Exit Route Map

TMS320F/C24x DSP Controllers Reference Guide

CPU and Instruction Set

Literature Number: SPRU160C
June 1999

IMPORTANT NOTICE

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

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

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

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

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

Read This First

About This Manual

The purpose of this user's guide is to assist you, the hardware or software engineer, in developing applications using the TMS320F/C240,F/C24x digital signal processors (DSPs). This book provides CPU, hardware, and instruction set details for these devices, and explains the DSP core that is common to all TMS320F/C24x DSP controllers; and consequently, is common to the devices described in this manual.

Throughout this book, the TMS320F/C240,F/C24x devices are generally referred to as '24x.

For information about '24x peripherals, see *TMS320F/C240 DSP Controllers, Peripheral Library and Specific Devices* (literature number SPRU161), or *TMS320F243/F241/C242 DSP Controllers System and Peripherals Reference Guide* (literature number SPRU276).

Notational Conventions

This document uses the following conventions:

- Program listings and program examples are shown in a `special typeface`.

  Here is a segment of a program listing:

```
OUTPUT  LDP     #6          ;select data page 6
        BLDD    #300, 20h   ;move data at address 300h to 320h
        RET
```

- Hexadecimal numbers are represented with a lowercase letter *h* following the number. For example, 7400h or 743Fh.

- In syntax descriptions, the instruction is in a **bold typeface** and parameters are in an *italic typeface*. Portions of a syntax in **bold** must 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:

  BLDD *source, destination*

  BLDD is the instruction and has two parameters, *source* and *destination*. When you use **BLDD**, the first parameter must be an actual data memory

source address and the second parameter must be a destination address. A comma and a space (optional) must separate the two addresses.

- ❏ Square brackets, [ ], identify an optional parameter. If you use an optional parameter, specify the information within the brackets; do not type the brackets themselves. When you specify more than one optional parameter from a list, you separate them with a comma and a space. Here is a sample syntax:

  BLDD *source*, *destination* [, **AR***n*]

  BLDD is the instruction. The two required operands are *source* and *destination*, and the optional operand is **AR***n*. **AR** is bold and *n* is italic; if you choose to use **AR***n*, you must type the letters A and R and then supply a chosen value for *n* (in this case, a value from 0 to 7). Here is an example:

Information About Cautions

This book contains cautions.

CAUTION

This is an example of a caution statement.

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

Related Documentation from Texas Instruments

The following books describe the ’C24x 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. Many of these documents are located on the internet at http://www.ti.com.

TMS320F/C240 DSP Controllers Peripheral Library and Specific Devices Reference Guide (literature number SPRU161) describes the peripherals available on the TMS320F/C240 digital signal processor controllers and their operation. Also described are specific device configurations of the ’C24x family.

TMS320F243/F241/C242 DSP Controllers System and Peripherals Reference Guide (literature number SPRU276) describes the architecture, system hardware, peripherals, and general operation of the TMS320F243, ’F241, and ’C242 digital signal processor (DSP) controllers.

TMS320C240, TMS320F240 DSP Controllers (literature number SPRS042) data sheet contains the electrical and timing specifications for these devices, as well as signal descriptions and pinouts for all of the available packages.

TMS320F20x/F24x Embedded Flash Memory Technical Reference (literature number SPRU282) Describes the operation of the embedded flash EEPROM module on the TMS320F20x/F24x digital signal processor (DSP) devices and provides sample code that you can use to develop your own software.

TMS320C1x/C2x/C2xx/C5x Code Generation Tools Getting Started Guide (literature number SPRU121) describes how to install the TMS320C1x, TMS320C2x, TMS320C2xx, and TMS320C5x assembly language tools and the C compiler for the ’C1x, ’C2x, ’C2xx, and ’C5x devices. The installations for MS-DOS™, OS/2™, SunOS™, and Solaris™ systems are covered.

TMS320C1x/C2x/C2xx/C5x 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, ’C2xx, and ’C5x generations of devices.

TMS320C2x/C2xx/C5x Optimizing C Compiler User's Guide (literature number SPRU024) describes the 'C2x/C2xx/C5x C compiler. This C compiler accepts ANSI standard C source code and produces TMS320 assembly language source code for the 'C2x, 'C2xx, and 'C5x generations of devices.

TMS320C2xx C Source Debugger User's Guide (literature number SPRU151) tells you how to invoke the 'C2xx emulator and simulator versions of the C source debugger interface. This book discusses various aspects of the debugger interface, including window management, command entry, code execution, data management, and breakpoints. It also includes a tutorial that introduces basic debugger functionality.

TMS320C2xx Simulator Getting Started (literature number SPRU137) describes how to install the TMS320C2xx simulator and the C source debugger for the 'C2xx. The installation for MS-DOS™, PC-DOS™, SunOS™, Solaris™, and HP-UX™ systems is covered.

TMS320C2xx Emulator Getting Started Guide (literature number SPRU209) tells you how to install the Windows™ 3.1 and Windows™ 95 versions of the 'C2xx emulator and C source debugger interface.

XDS51x Emulator Installation Guide (literature number SPNU070) describes the installation of the XDS510™, XDS510PP™, and XDS510WS™ emulator controllers. The installation of the XDS511™ emulator is also described.

XDS522/XDS522A Emulation System Installation Guide (literature number SPRU171) describes the installation of the emulation system. Instructions include how to install the hardware and software for the XDS522™ and XDS522A™.

XDS522A Emulation System User's Guide (literature number SPRU169) tells you how to use the XDS522A™ emulation system. This book describes the operation of the breakpoint, tracing, and timing functionality in the XDS522A emulation system. This book also discusses BTT software interface and includes a tutorial that uses step-by-step instructions to demonstrate how to use the XDS522A emulation system.

XDS522A Emulation System Online Help (literature number SPRC002) is an online help file that provides descriptions of the BTT software user interface, menus, and dialog boxes.

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

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

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

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

Related Technical Articles

The following technical articles contain useful information regarding designs, operations, and applications for signal-processing systems. These articles supplement the material in this book.

"A Greener World Through DSP Controllers", Panos Papamichalis, *DSP & Multimedia Technology*, September 1994.

"A Single-Chip Multiprocessor DSP for Image Processing—TMS320C80", Dr. Ing. Dung Tu, *Industrie Elektronik*, Germany, March 1995.

"Application Guide with DSP Leading-Edge Technology", Y. Nishikori, M. Hattori, T. Fukuhara, R.Tanaka, M. Shimoda, I. Kudo, A.Yanagitani, H. Miyaguchi, et al., *Electronics Engineering*, November 1995.

"Approaching the No-Power Barrier", Jon Bradley and Gene Frantz, *Electronic Design*, January 9, 1995.

"Beware of BAT: DSPs Add Brilliance to New Weapons Systems", Panos Papamichalis, *DSP & Multimedia Technology*, October 1994.

"Choose DSPs for PC Signal Processing", Panos Papamichalis, *DSP & Multimedia Technology*, January/February 1995.

"Developing Nations Take Shine to Wireless", Russell MacDonald, Kara Schmidt and Kim Higden, *EE Times*, October 2, 1995.

"Digital Signal Processing Solutions Target Vertical Application Markets", Ron Wages, *ECN*, September 1995.

"Digital Signal Processors Boost Drive Performance", Tim Adcock, *Data Storage*, September/October 1995.

"DSP and Speech Recognition, An Origin of the Species", Panos Papamichalis, *DSP & Multimedia Technology*, July 1994.

"DSP Design Takes Top-Down Approach", Andy Fritsch and Kim Asal, *DSP Series Part III*, *EE Times*, July 17, 1995.

"DSPs Advance Low-Cost 'Green' Control", Gregg Bennett, *DSP Series Part II*, *EE Times*, April 17, 1995.

"DSPs Do Best on Multimedia Applications", Doug Rasor, *Asian Computer World*, October 9–16, 1995.

"DSPs: Speech Recognition Technology Enablers", Gene Frantz and Gregg Bennett, *I&CS*, May 1995.

"Easing JTAG Testing of Parallel-Processor Projects", Tony Coomes, Andy Fritsch, and Reid Tatge, *Asian Electronics Engineer*, Manila, Philippines, November 1995.

"Fixed or Floating? A Pointed Question in DSPs", Jim Larimer and Daniel Chen, *EDN*, August 3, 1995.

"Function-Focused Chipsets: Up the DSP Integration Core", Panos Papamichalis, *DSP & Multimedia Technology*, March/April 1995.

"GSM: Standard, Strategien und Systemchips", Edgar Auslander, *Elektronik Praxis*, Germany, October 6, 1995.

"High Tech Copiers to Improve Images and Reduce Paperwork", Karl Guttag, *Document Management*, July/August 1995.

"Host-Enabled Multimedia: Brought to You by DSP Solutions", Panos Papamichalis, *DSP & Multimedia Technology*, September/October 1995.

"Integration Shrinks Digital Cellular Telephone Designs", Fred Cohen and Mike McMahan, *Wireless System Design*, November 1994.

"On-Chip Multiprocessing Melds DSPs", Karl Guttag and Doug Deao, *DSP Series Part III*, *EE Times*, July 18, 1994.

"Real-Time Control", Gregg Bennett, *Appliance Manufacturer*, May 1995.

"Speech Recognition", P.K. Rajasekaran and Mike McMahan, *Wireless Design & Development*, May 1995.

"Telecom Future Driven by Reduced Milliwatts per DSP Function", Panos Papamichalis, *DSP & Multimedia Technology*, May/June 1995.

"The Digital Signal Processor Development Environment", Greg Peake, *Embedded System Engineering*, United Kingdom, February 1995.

"The Growing Spectrum of Custom DSPs", Gene Frantz and Kun Lin, *DSP Series Part II*, *EE Times*, April 18, 1994.

"The Wide World of DSPs, " Jim Larimer, *Design News*, June 27, 1994.

"Third-Party Support Drives DSP Development for Uninitiated and Experts Alike", Panos Papamichalis, *DSP & Multimedia Technology*, December 1994/January 1995.

"Toward an Era of Economical DSPs", John Cooper, *DSP Series Part I*, *EE Times*, Jan. 23, 1995.

Trademarks

Contents

Figures

Tables

Examples

Chapter 1

Introduction

The TMS320C24x is a member of the TMS320 family of digital signal processors (DSPs). The ’C24x is designed to meet a wide range of digital motor control (DMC) and embedded control applications. This chapter provides an overview of the current TMS320 family, and describes the background and benefits of the ’C24x DSP controller products.

Topic **Page**

1.1 TMS320 Family Overview

The TMS320 family consists of fixed-point, floating-point, multiprocessor digital signal processors (DSPs), and fixed-point DSP controllers. TMS320 DSPs have an architecture designed specifically for real-time signal processing. The ’C24x series of DSP controllers combines this real-time processing capability with controller peripherals to create an ideal solution for control system applications. The following characteristics make the TMS320 family the right choice for a wide range of processing applications:

- ❏ Very flexible instruction set
- ❏ Inherent operational flexibility
- ❏ High-speed performance
- ❏ Innovative parallel architecture
- ❏ Cost effectiveness

In 1982, Texas Instruments introduced the TMS32010, the first fixed-point DSP in the TMS320 family. Before the end of the year, *Electronic Products* magazine awarded the TMS32010 the title “Product of the Year”. Today, the TMS320 family consists of the following generations shown in Figure 1–1: ’C1x, ’C2x, ’C24x, ’C5x, ’C54x, and ’C6x fixed-point DSPs; ’C3x and ’C4x floating-point DSPs; and ’C8x multiprocessor DSPs. The ’C24x is considered part of the ’C24x family of fixed-point DSPs, and a member of the ’C2000 platform.

Devices within a generation of the TMS320 family have the same CPU structure but different on-chip memory and peripheral configurations. Spin-off devices use new combinations of on-chip memory and peripherals to satisfy a wide range of needs in the worldwide electronics market. By integrating memory and peripherals onto a single chip, TMS320 devices reduce system costs and save circuit board space.

Figure 1–1. DSP Product Generation

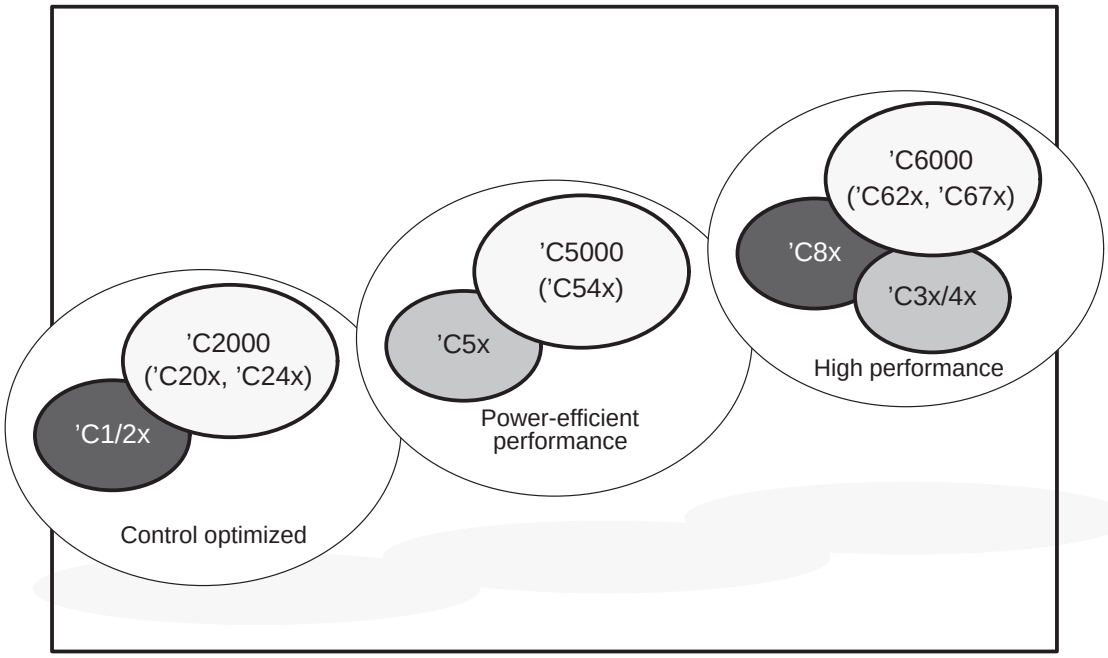
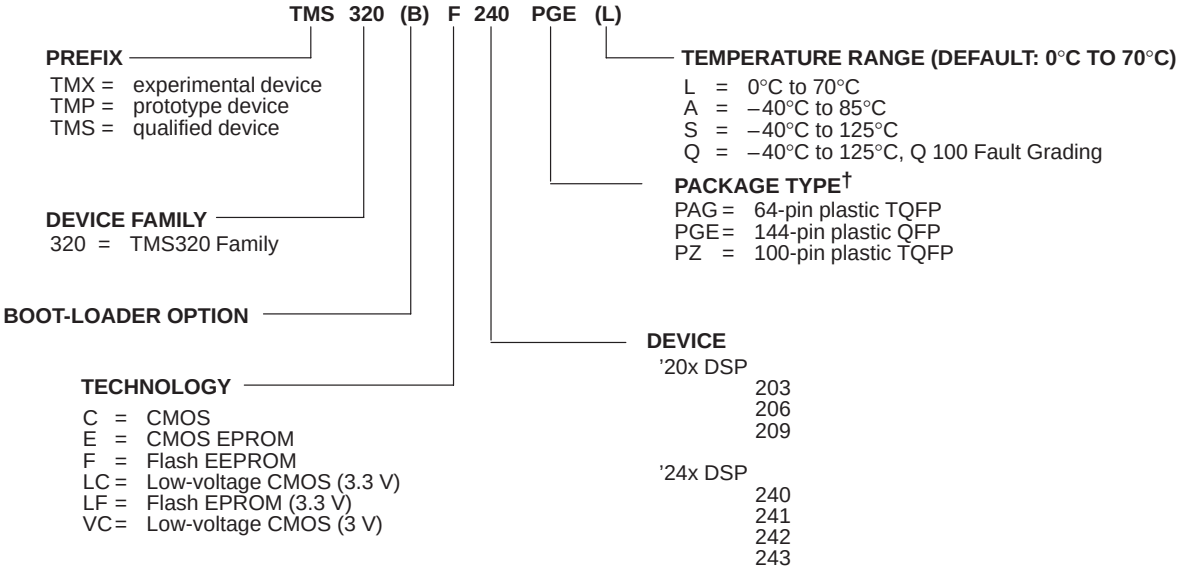


Figure 1–2. TMS320 Device Nomenclature



† PLCC = Plastic J-Leaded Chip Carrier
QFP = Quad Flatpack
TQFP = Thin Quad Flatpack

1.2 TMS320C24x Series of DSP Controllers

Designers have recognized the opportunity to redesign existing DMC systems to use advanced algorithms that yield better performance and reduce system component count. DSPs enable:

- ❑ Design of robust controllers for a new generation of inexpensive motors, such as AC induction, DC permanent magnet, and switched-reluctance motors
- ❑ Full variable-speed control of brushless motor types that have lower manufacturing cost and higher reliability
- ❑ Energy savings through variable-speed control, saving up to 25% of the energy used by fixed-speed controllers
- ❑ Increased fuel economy, improved performance, and elimination of hydraulic fluid in automotive electronic power steering (EPS) systems
- ❑ Reduced manufacturing and maintenance costs by eliminating hydraulic fluids in automotive electronic braking systems
- ❑ More efficient and quieter operation due to less generation of torque ripple, resulting in less loss of power, lower vibration, and longer life
- ❑ Elimination or reduction of memory lookup tables through real-time polynomial calculation, thereby reducing system cost
- ❑ Use of advanced algorithms that can reduce the number of sensors required in a system
- ❑ Control of power switching inverters, along with control algorithm processing
- ❑ Single-processor control of multimotor systems

The ’C24x DSP controllers are designed to meet the needs of control-based applications. By integrating the high performance of a DSP core and the on-chip peripherals of a microcontroller into a single-chip solution, the ’C24x series yields a device that is an affordable alternative to traditional microcontroller units (MCUs) and expensive multichip designs. At 20 million instructions per second (MIPS), the ’C24x DSP controllers offer significant performance over traditional 16-bit microcontrollers and microprocessors. Future derivatives of these devices will run at speeds higher than 20 MIPS.

The 16-bit, fixed-point DSP core of the ’C24x device provides analog designers a digital solution that does not sacrifice the precision and performance of

their systems. In fact, system performance can be enhanced through the use of advanced control algorithms for techniques such as adaptive control, Kalman filtering, and state control. The ’C24x DSP controllers offer reliability and programmability. Analog control systems, on the other hand, are hard-wired solutions and can experience performance degradation due to aging, component tolerance, and drift.

The high-speed central processing unit (CPU) allows the digital designer to process algorithms in real time rather than approximate results with look-up tables. When the instruction set of these DSP controllers (which incorporates both signal processing instructions and general-purpose control functions) is coupled with the extensive development support available for the ’C24x devices, it reduces development time and provides the same ease of use as traditional 8- and 16-bit microcontrollers. The instruction set also allows you to retain your software investment when moving from other general-purpose TMS320 fixed-point DSPs. It is source- and object-code compatible with the other members of the ’C24x generation, source code compatible with the ’C2x generation, and upwardly source code compatible with the ’C5x generation of DSPs from Texas Instruments.

The ’C24x architecture is also well-suited for processing control signals. It uses a 16-bit word length along with 32-bit registers for storing intermediate results, and has two hardware shifters available to scale numbers independently of the CPU. This combination minimizes quantization and truncation errors, and increases processing power for additional functions. Two examples of these additional functions are: a notch filter that cancels mechanical resonances in a system, and an estimation technique that eliminates state sensors in a system.

The ’C24x DSP controllers take advantage of an existing set of peripheral functions that allow Texas Instruments to quickly configure various series members for different price/performance points or for application optimization. This library of both digital and mixed-signal peripherals includes:

- ❏ Timers
- ❏ Serial communications ports (SCI, SPI)
- ❏ Analog-to-digital converters (ADC)
- ❏ Event manager
- ❏ System protection, such as watchdog timers
- ❏ CAN controller

The DSP controller peripheral library is continually growing and changing to suit the needs of tomorrow’s embedded control marketplace.

Chapter 2

Architectural Overview

This chapter provides an overview of the architectural structure and components of the ’C24x DSP CPU. The ’C24x DSP uses an advanced, modified Harvard architecture that maximizes processing power by maintaining separate bus structures for program memory and data memory.

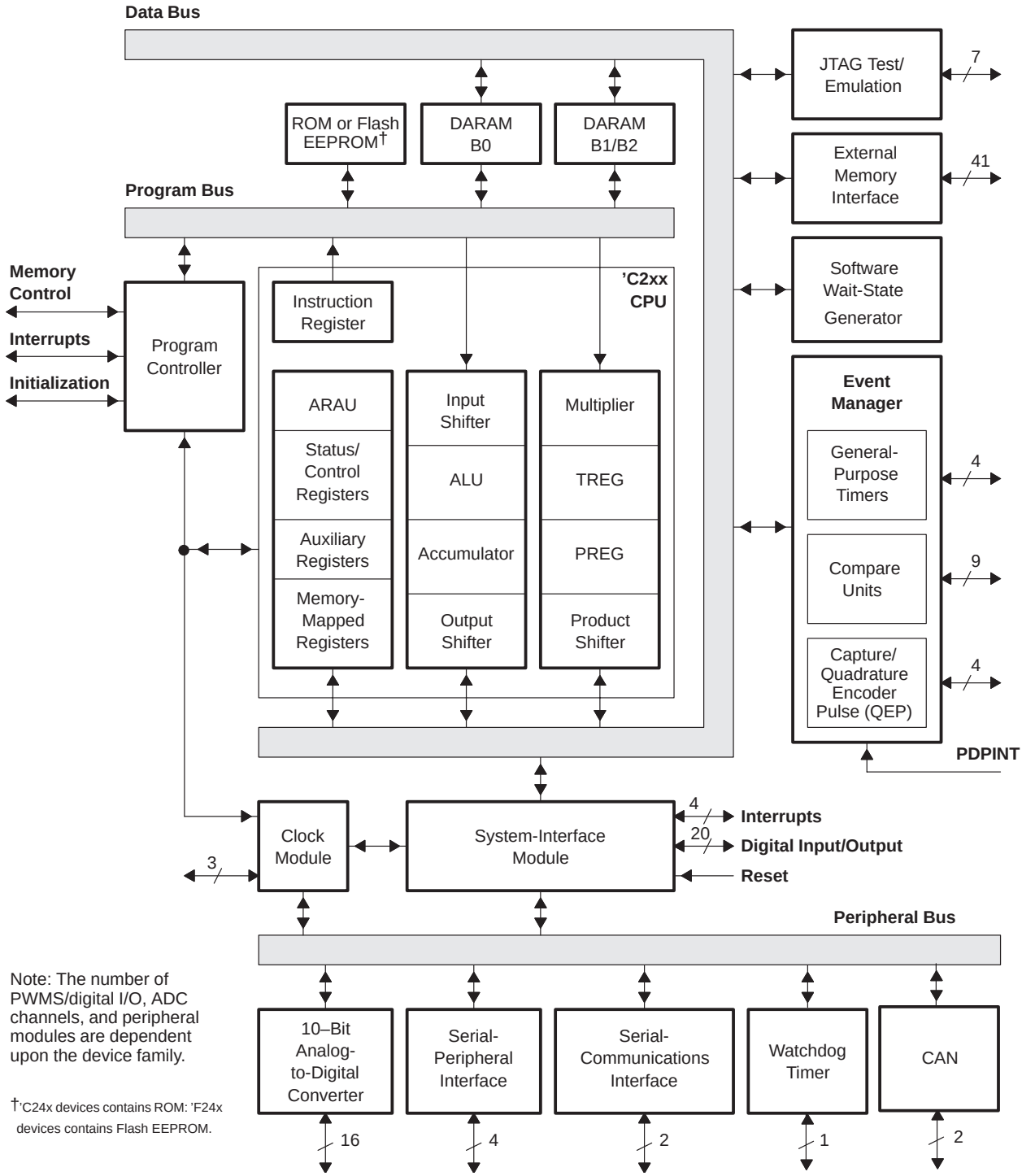
Topic **Page**

2.1 Architecture Summary

A functional block diagram of the ’C24x DSP controller architecture is shown in Figure 2–1 on page 2-3. The ’C24x DSP architecture is based on a modified Harvard architecture, which supports separate bus structures for program space and data space. A third space, the input/output (I/O) space, is also available and is accessible through the *external bus interface*. To support a large selection of peripherals, a peripheral bus is used. The peripheral bus is mapped to the data space and interfaced to the data bus through a special system module. Thus, all the instructions that operate on the data space also operate on all the peripheral registers.

Separate program and data spaces allow simultaneous access to program instructions and data. For example, while data is multiplied, a previous product can be added to the accumulator, and at the same time, a new address can be generated. Such parallelism supports a set of arithmetic, logic, and bit-manipulation operations that can all be performed in a single machine cycle. The ’C24x also includes control mechanisms to manage interrupts, repeated operations, and function/subroutine calls.

Figure 2–1. TMS320C24x DSP Controller Functional Block Diagram



Note: The number of PWMS/digital I/O, ADC channels, and peripheral modules are dependent upon the device family.

†'C24x devices contains ROM: 'F24x devices contains Flash EEPROM.

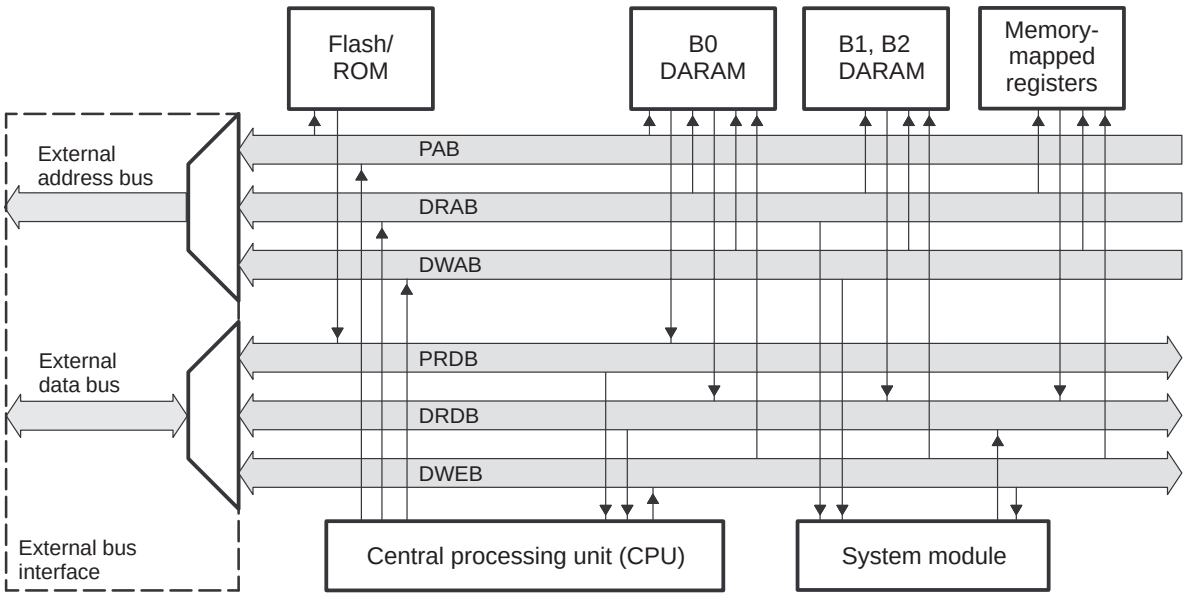
2.2 'C24x CPU Internal Bus Structure

The 'C24x DSP, a member of the TMS320 family of DSPs, includes a 'C2xx DSP core designed using the '2xLP ASIC core. The 'C2xx DSP core has an internal data and program bus structure that is divided into six 16-bit buses (see Figure 2–2). The six buses are:

- **PAB.** The *program address bus* provides addresses for both reads from and writes to program memory.
- **DRAB.** The *data-read address bus* provides addresses for reads from data memory.
- **DWAB.** The *data-write address bus* provides addresses for writes to data memory.
- **PRDB.** The *program read bus* carries instruction code and immediate operands, as well as table information, from program memory to the CPU.
- **DRDB.** The *data-read bus* carries data from data memory to the central arithmetic logic unit (CALU) and the auxiliary register arithmetic unit (ARAU).
- **DWEB.** The *data-write bus* carries data to both program memory and data memory.

Having separate address buses for data reads (DRAB) and data writes (DWAB) allows the CPU to read and write in the same machine cycle.

Figure 2–2. 'C24x Address and Data Bus Structure



2.3 Memory

The ’C24x contains the following types of on-chip memory:

- Dual-access RAM (DARAM)
- Flash EEPROM or ROM (masked)

The ’C24x memory is organized into four individually-selectable spaces:

- Program (64K words)
- Local data (64K words)
- Global data (32K words)
- Input/Output (64K words)

These spaces form an address range of 224K words.

2.3.1 On-Chip Dual-Access RAM (DARAM)

The ’C24x has 544 words of on-chip DARAM, which can be accessed twice per machine cycle. This memory is primarily intended to hold data, but when needed, can also be used to hold programs. The memory can be configured in one of two ways, depending on the state of the CNF bit in status register ST1.

- When CNF = 0, all 544 words are configured as data memory.
- When CNF = 1, 288 words are configured as data memory and 256 words are configured as program memory.

Because DARAM can be accessed twice per cycle, it improves the speed of the CPU. The CPU operates within a 4-cycle pipeline. In this pipeline, the CPU reads data on the third cycle and writes data on the fourth cycle. However, DARAM allows the CPU to write and read in one cycle; the CPU writes to DARAM on the master phase of the cycle and reads from DARAM on the slave phase. For example, suppose two instructions, A and B, store the accumulator value to DARAM and load the accumulator with a new value from DARAM. Instruction A stores the accumulator value during the master phase of the CPU cycle, and instruction B loads the new value in the accumulator during the slave phase. Because part of the dual-access operation is a write, it only applies to RAM.

2.3.2 Flash EEPROM

Flash EEPROM provides an attractive alternative to masked program ROM. Like ROM, flash is a nonvolatile memory type; however, it has the advantage of *in-target* reprogrammability. The ’F24x incorporates one 16K/8K × 16-bit flash EEPROM module in program space. This type of memory expands the capabilities of the ’F24x in the areas of prototyping, early field testing, and single-chip applications.

Unlike most discrete flash memory, the ’F24x flash does not require a dedicated state machine because the algorithms for programming and erasing the flash are executed by the DSP core. This enables several advantages, including reduced chip size and sophisticated adaptive algorithms. For production programming, the IEEE Standard 1149.1 (JTAG) scan port provides easy access to on-chip RAM for downloading the algorithms and flash code. Other key features of the flash include zero-wait-state access rate and single 5-V power supply.

An erased bit in the ’24x flash is read as a logic one, and a programmed bit is read as a logic zero. The flash requires a block-erase of the entire 16K/8K module; however, any combination of bits can be programmed. The following four algorithms are required for flash operations: clear, erase, flash-write, and program. For an explanation of these algorithms and a complete description of the flash EEPROM, see *TMS320F20x/F24x DSPs Embedded Flash Memory Technical Reference* (Literature number SPRU282).

2.3.3 Flash Serial Loader

Most of the on-chip flash devices are shipped with a serial bootloader code programmed at the following addresses: 0x0000 – 0x00FFh. All other flash addresses are in an erased state. The serial bootloader can be used to program the on-chip flash memory with user’s code. During the flash programming sequence, the on-chip data RAM is used to load and execute the clear, erase, and program algorithms.

2.3.4 Factory-Masked ROM

For large-volume applications consisting of stable software free of bugs, low-cost, masked ROM is available and supported up to 16K or 4K words. If you want a custom ROM, you can provide the code or data to be programmed into the ROM in object-file format, and Texas Instruments will generate the appropriate process mask to program the ROM. For details, see Appendix B, *Submitting ROM Codes to TI*.

A small portion of the ROM (128 or 64 words) is reserved by Texas Instruments for test purposes. These reserved locations are at addresses 0x3F80 or 3FC0 through 0x3FFF. This leaves about 16K words available for your code.

Figure 2–3. ’C24x ROM Memory Map

2.3.5 External Memory Interface Module

In addition to full, on-chip memory support, some of the ’C24x devices provide access to external memory by way of the *External Memory Interface Module*. This interface provides 16 external address lines, 16 external data lines, and relevant control signals to select data, program, and I/O spaces. An on-chip wait-state generator allows interfacing with slower off-chip memory and peripherals.

2.4 Central Processing Unit

The ’C24x is based on TI’s ’C2xx CPU. It contains:

- A 32-bit central arithmetic logic unit (CALU)
- A 32-bit accumulator
- Input and output data-scaling shifters for the CALU
- A 16-bit × 16-bit multiplier
- A product-scaling shifter
- Data-address generation logic, which includes eight auxiliary registers and an auxiliary register arithmetic unit (ARAU)
- Program-address generation logic

2.4.1 Central Arithmetic Logic Unit (CALU) and Accumulator

The ’C24x performs 2s-complement arithmetic using the 32-bit CALU. The CALU uses 16-bit words taken from data memory, derived from an immediate instruction, or from the 32-bit multiplier result. In addition to arithmetic operations, the CALU can perform Boolean operations.

The accumulator stores the output from the CALU; it can also provide a second input to the CALU. The accumulator is 32 bits wide and is divided into a high-order word (bits 31 through 16) and a low-order word (bits 15 through 0). Assembly language instructions are provided for storing the high- and low-order accumulator words to data memory.

2.4.2 Scaling Shifters

The ’C24x has three 32-bit shifters that allow for scaling, bit extraction, extended arithmetic, and overflow-prevention operations:

- **Input data-scaling shifter (input shifter).** This shifter left-shifts 16-bit input data by 0 to 16 bits to align the data to the 32-bit input of the CALU.

- **Output data-scaling shifter (output shifter).** This shifter left-shift output from the accumulator by 0 to 7 bits before the output is stored to data memory. The content of the accumulator remains unchanged.

- **Product-scaling shifter (product shifter)**. The product register (PREG) receives the output of the multiplier. The product shifter shifts the output of the PREG before that output is sent to the input of the CALU. The product shifter has four product shift modes (no shift, left shift by one bit, left shift by four bits, and right shift by six bits), which are useful for performing multiply/accumulate operations, performing fractional arithmetic, or justifying fractional products.

2.4.3 Multiplier

The on-chip multiplier performs 16-bit × 16-bit 2s-complement multiplication with a 32-bit result. In conjunction with the multiplier, the ’C24x uses the 16-bit temporary register (TREG) and the 32-bit product register (PREG); TREG always supplies one of the values to be multiplied, and PREG receives the result of each multiplication.

Using the multiplier, TREG, and PREG, the ’C24x efficiently performs fundamental DSP operations such as convolution, correlation, and filtering. The effective execution time of each multiplication instruction can be as short as one CPU cycle.

2.4.4 Auxiliary Register Arithmetic Unit (ARAU) and Auxiliary Registers

The ARAU generates data memory addresses when an instruction uses indirect addressing (see Chapter 6, *Addressing Modes*) to access data memory. The ARAU is supported by eight auxiliary registers (AR0 through AR7), each of which can be loaded with a 16-bit value from data memory or directly from an instruction word. Each auxiliary register value can also be stored in data memory. The auxiliary registers are referenced by a 3-bit auxiliary register pointer (ARP) embedded in status register ST0.

2.5 Program Control

Several hardware and software mechanisms provide program control:

- Program control logic decodes instructions, manages the 4-level pipeline, stores the status of operations, and decodes conditional operations. Hardware elements included in the program control logic are the program counter, the status registers, the stack, and the address-generation logic.
- Software mechanisms used for program control include branches, calls, conditional instructions, a repeat instruction, reset, interrupts, and power-down modes.

Table 2–1 shows where you can find detailed information about these program control features.

Table 2–1. Where to Find Information About Program Control Features

For information about	See
Address-generation logic	Chapter 5, *Program Control*
Address-generation data memory	Chapter 6, *Addressing Modes*
Branches, calls, and returns	Chapter 5, *Program Control*
Conditional operations	Chapter 5, *Program Control*
Interrupts	Chapter 5, *Program Control*
Pipeline	Chapter 5, *Program Control*
Program counter	Chapter 5, *Program Control*
Repeat instruction	Chapter 5, *Program Control*
Reset	Chapter 5, *Program Control*
Stack	Chapter 5, *Program Control*
Status registers	Chapter 4, *Central Processing Unit*

2.6 Serial-Scan Emulation

The ’C24x has seven pins dedicated to the serial scan emulation port (JTAG port). This port allows for non-intrusive emulation of ’C24x devices, and is supported by Texas Instruments emulation tools and by many third party debugger tools. For documentation on these emulation and debugger tools, see *Related Documentation From Texas Instruments* in the preface section of this book, and *Design Considerations for Using XDS510 Emulator* in Appendix C.

Chapter 3

Memory and I/O Spaces

The ’C24x has a 16-bit address line that accesses four individually selectable spaces (224K words total):

- A 64K-word program space
- A 64K-word local data space
- A 32K-word global data space
- A 64K-word I/O space

This chapter describes these four spaces and shows memory maps for program, data, and I/O spaces. It also describes available ’C24x memory configuration options.

Topic **Page**

3.1 Overview of Memory and I/O Spaces

The ’C24x design is based on an enhanced Harvard architecture. The ’C24x has multiple memory spaces accessible on three parallel buses: a program address bus (PAB), a data-read address bus (DRAB), and a data-write address bus (DWAB). Each of the three buses access different memory spaces for different phases of the device’s operation. Because the bus operations are independent, it is possible to access both the program and data spaces simultaneously. Within a given machine cycle, the CALU can execute as many as three concurrent memory operations.

The ’C24x address map is organized into four individually selectable spaces:

- **Program memory** (64K words) contains the instructions to be executed, as well as data used during program execution.
- **Data memory** (64K words) holds data used by the instructions.
- **Global data memory** (32K words) shares data with other devices or serves as additional data space.
- **Input/output (I/O) space** (64K words) interfaces to external peripherals and may contain on-chip registers.

These spaces provide a total address space of 224K words. The ’C24x includes on-chip memory to aid in system performance and integration, and numerous addresses that can be used for external memory and I/O devices.

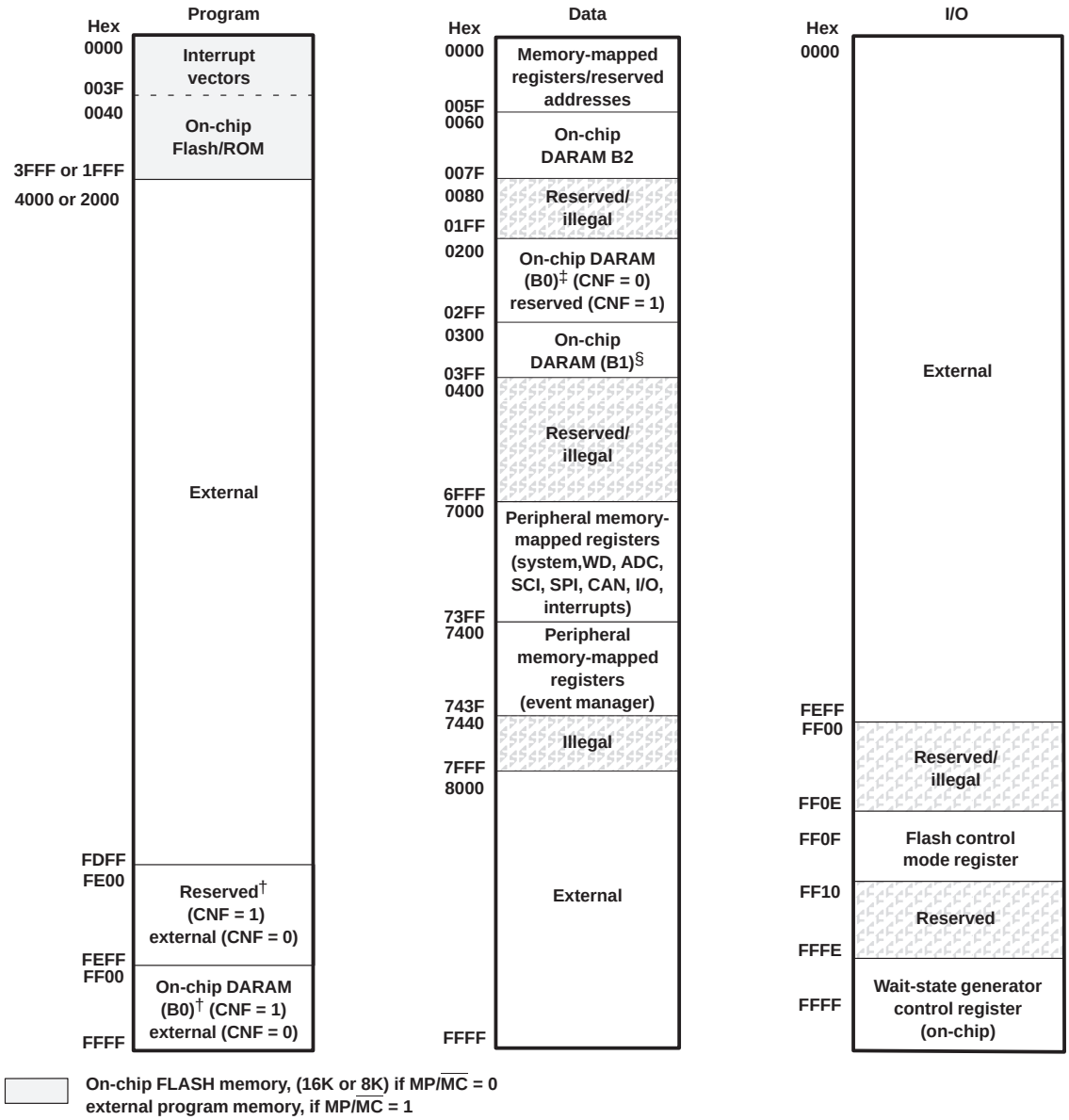
The advantages of operating from on-chip memory are:

- Higher performance than external memory (because the wait states required for slower external memories are avoided)
- Lower cost than external memory
- Lower power consumption than external memory

The advantage of operating from external memory is the ability to access a larger address space.

The memory maps shown in Figure 3–1 are generic for all ’C24x devices; however, each device has its own set of memory maps. ’C24x devices are available with different combinations of on-chip memory and peripherals. You should refer to the appropriate data sheet for details about a specific device.

Figure 3–1. Generic Memory Maps for ’C24x DSP Controllers



† When CNF = 1, addresses FE00h–FEFFh and FF00h–FFFFh are mapped to the same physical block (B0) in program-memory space. For example, a write to FE00h will have the same effect as a write to FF00h. For simplicity, addresses FE00h–FEFFh are referred to as reserved when CNF = 1.

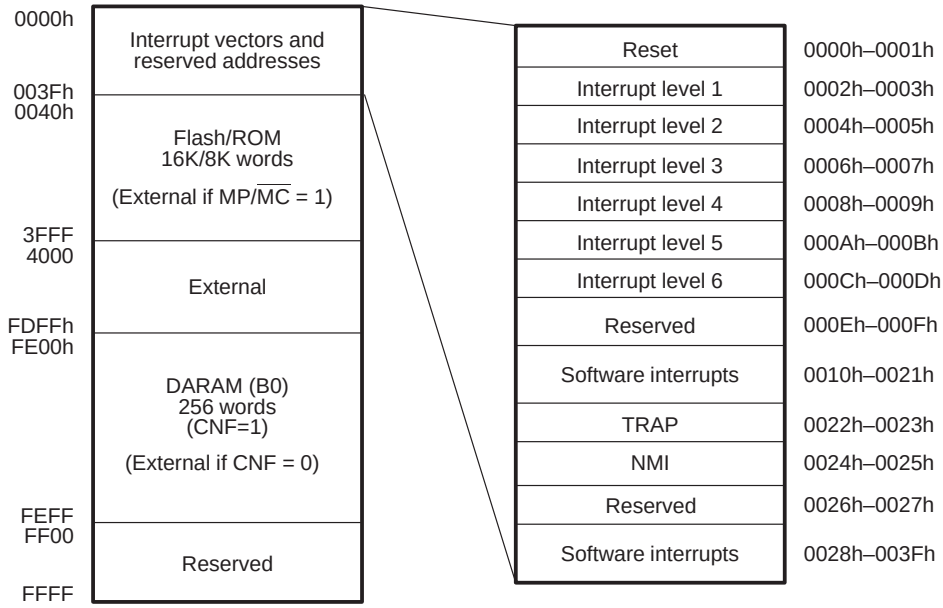
‡ When CNF = 0, addresses 0100h–01FFh and 0200h–02FFh are mapped to the same physical block (B0) in data-memory space. For example, a write to 0100h will have the same effect as a write to 0200h. For simplicity, addresses 0100h–01FFh are referred to as reserved.

§ Addresses 0300h–03FFh and 0400h–04FFh are mapped to the same physical block (B1) in data-memory space. For example, a write to 0400h has the same effect as a write to 0300h. For simplicity, addresses 0400h–04FFh are referred to as reserved.

3.2 Program Memory

The program-memory space is where the application program code resides; it can also hold table information and immediate operands. The program-memory space addresses up to 64K 16-bit words. On the ’C24x device, these words include on-chip DARAM and on-chip ROM/flash EEPROM. When the ’C24x generates an address outside the set of addresses configured to on-chip program memory, the device automatically generates an external access, asserting the appropriate control signals (if an external memory interface is present). Figure 3–2 shows the ’C24x program memory map.

Figure 3–2. Program Memory Map for ’C24x



Note: Flash/ROM memory includes the address range 0000h–003Fh.

3.2.1 Program Memory Configuration

Depending on which types of memory are included in a particular ’C24x device, two factors contribute to the configuration of program memory:

- **CNF bit.** The CNF bit (bit 12) of status register ST1 determines whether the addresses for DARAM B0 are available for program space:
  - **CNF = 0.** There is no addressable on-chip program DARAM.
  - **CNF = 1.** The 256 words of DARAM B0 are configured for program use. At reset, any words of program/data DARAM are mapped into local data space (CNF = 0).
- **MP/$\overline{\text{MC}}$ pin.** The level on the MP/$\overline{\text{MC}}$ pin determines whether program instructions are read from on-chip ROM or flash EEPROM (if available) after reset:
  - **MP/$\overline{\text{MC}}$ = 0.** The device is configured as a microcomputer. The on-chip ROM/flash EEPROM is accessible. The device fetches the reset vector from on-chip memory.
  - **MP/$\overline{\text{MC}}$ = 1.** The device is configured as a microprocessor. The device fetches the reset vector from external memory.

  Regardless of the value of MP/$\overline{\text{MC}}$, the ’C24x fetches its reset vector at location 0000h of program memory.

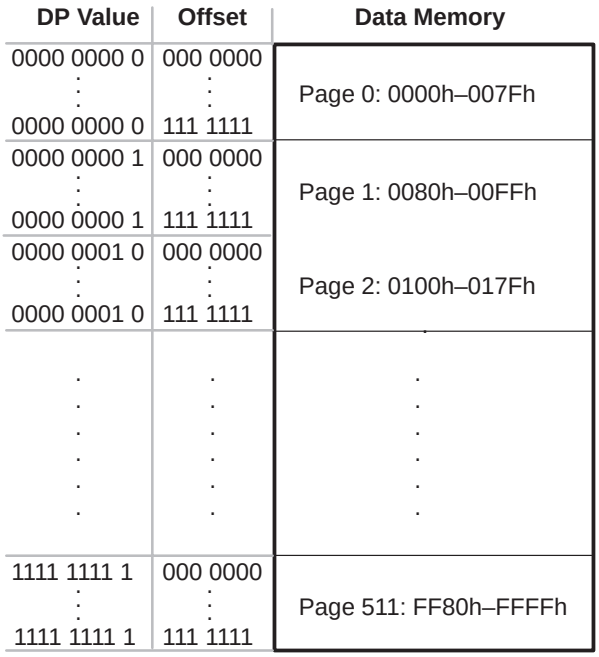
3.3 Data Memory

Data-memory space addresses up to 64K 16-bit words. Each ’C24x device has three on-chip DARAM blocks: B0, B1, and B2. Block B0 is configurable as either data memory or program memory. Blocks B1 and B2 are available for data memory only.

Data memory can be addressed with either of two addressing modes: direct-addressing or indirect-addressing. Addressing modes are described in detail in Chapter 6.

When direct addressing is used, data memory is addressed in blocks of 128 words called data pages. The entire 64K of data memory consists of 512 data pages labeled 0 through 511. The current data page is determined by the value in the 9-bit data page pointer (DP) in status register ST0. Each of the 128 words on the current page is referenced by a 7-bit offset, which is taken from the instruction that is using direct addressing. Therefore, when an instruction uses direct addressing, you must specify both the data page (with a preceding instruction) and the offset (in the instruction that accesses data memory).

Figure 3–3. Pages of Data Memory



3.3.1 Data Page 0 Address Map

The data memory also includes the device's memory-mapped registers (MMR), which reside at the top of data page 0 (addresses 0000h–007Fh). Note the following:

- ❑ The three registers that can be accessed with zero wait states are:
  - Interrupt mask register (IMR)
  - Global memory allocation register (GREG)
  - Interrupt flag register (IFR)
- ❑ The test/emulation reserved area is used by the test and emulation systems for special information transfers.

Do Not Write to Test/Emulation Addresses

Writing to the test/emulation addresses can cause the device to change its operating mode and, therefore, affect the operation of an application.

- ❑ The scratch-pad RAM block (B2) includes 32 words of DARAM that provide for variable storage without fragmenting the larger RAM blocks, whether internal or external. This RAM block supports dual-access operations and can be addressed via any data-memory addressing mode.

Table 3–1 shows the address map of data page 0.

Table 3–1. Data Page 0 Address Map

Address	Name	Description
0000h–0003h	–	Reserved
0004h	IMR	Interrupt mask register
0005h	GREG	Global memory allocation register
0006h	IFR	Interrupt flag register
0023h–0027h	–	Reserved
002Bh–002Fh	–	Reserved for test/emulation
0060h–007Fh	B2	Scratch-pad RAM (DARAM B2)

3.3.2 Data Memory Configuration

The following contributes to the configuration of data memory:

- **CNF bit.** The CNF bit (bit 12) of status register ST1 determines whether the on-chip DARAM B0 is mapped into local data space or into program space.
  - **CNF = 1.** DARAM B0 is used for program space.
  - **CNF = 0.** B0 is used for data space.

  At reset, B0 is mapped into local data space (CNF = 0).

3.4 Global Data Memory

Addresses in the upper 32K words (8000h–FFFFh) of local data memory can be used for global data memory. The global memory allocation register (GREG) determines the size of the global data-memory space, which is between 256 and 32K words. The GREG is connected to the eight LSBs of the internal data bus and is memory-mapped to data-memory location 0005h. Table 3–2 shows the allowable GREG values and shows the corresponding address range set aside for global data memory. Any remaining addresses within 8000h–FFFFh are available for local data memory.

Note:

Choose only the GREG values listed in Table 3–2. Other values lead to fragmented memory maps.

Table 3–2. Global Data Memory Configurations

GREG Value		Local Memory		Global Memory	
High Byte	Low Byte	Range	Words	Range	Words
XXXX XXXX	0000 0000	0000h–FFFFh	65 536	–	0
XXXX XXXX	1000 0000	0000h–7FFFh	32 768	8000h–FFFFh	32 768
XXXX XXXX	1100 0000	0000h–BFFFh	49 152	C000h–FFFFh	16 384
XXXX XXXX	1110 0000	0000h–DFFFh	57 344	E000h–FFFFh	8 192
XXXX XXXX	1111 0000	0000h–EFFFh	61 440	F000h–FFFFh	4 096
XXXX XXXX	1111 1000	0000h–F7FFh	63 488	F800h–FFFFh	2 048
XXXX XXXX	1111 1100	0000h–FBFFh	64 512	FC00h–FFFFh	1 024
XXXX XXXX	1111 1110	0000h–FDFFh	65 024	FE00h–FFFFh	512
XXXX XXXX	1111 1111	0000h–FEFFh	65 280	FF00h–FFFFh	256

Note: X = Don't care

When a program accesses any data address, the 'C24x drives the $\overline{DS}$ signal low. If that address is within the range defined by the GREG as a global address, $\overline{BR}$ signal is also asserted. Because $\overline{BR}$ differentiates local and global accesses, the addresses configured by the GREG value are an additional data space. The external data-address range is extended by the selected amount of global space (up to 32K words).

Global memory is available only on devices with an external memory interface. In other devices, this space and the GREG register are reserved.

As an example of configuring global memory, suppose you want to designate 8K data-memory addresses as global addresses. To do this, you write the 8-bit value 11100000 to the GREG (see Figure 3–4). This designates addresses E000h–FFFFh of data memory as global data addresses (see Figure 3–5).

Figure 3–4. GREG Register Set to Configure 8K for Global Data Memory

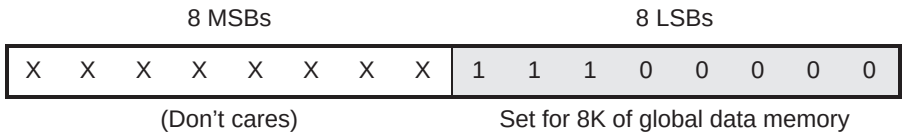
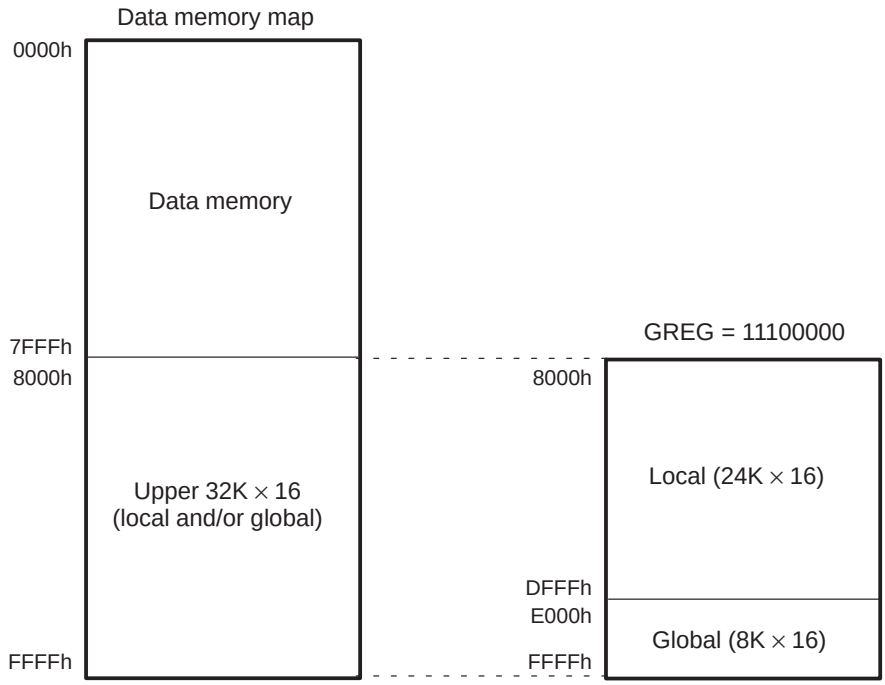


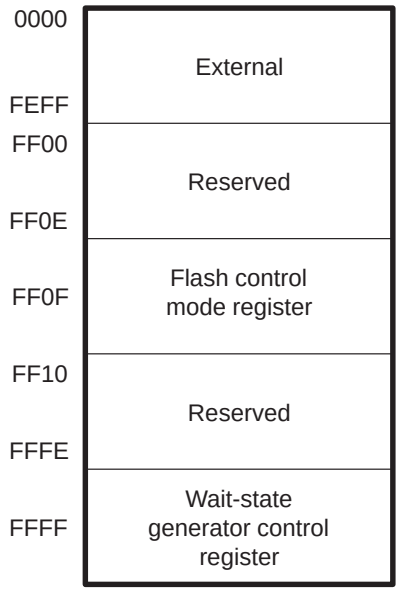
Figure 3–5. Global and Local Data Memory for GREG = 11100000



3.5 I/O Space

The I/O space memory addresses up to 64K 16-bit words. Figure 3–6 shows the I/O-space address map for the ’C24x.

Figure 3–6. I/O-Space Address Map for ’C24x



The I/O space is useful for mapping external peripherals and flash control registers. This I/O space is a generic space available for the ’C24x core. Depending on the specific device within the ’C24x family, the I/O space is partially available or disabled. You should refer to the specific data sheet for exact details.

External I/O space is available only in ’24x devices that have an external memory interface; otherwise, this space is reserved.

Chapter 4

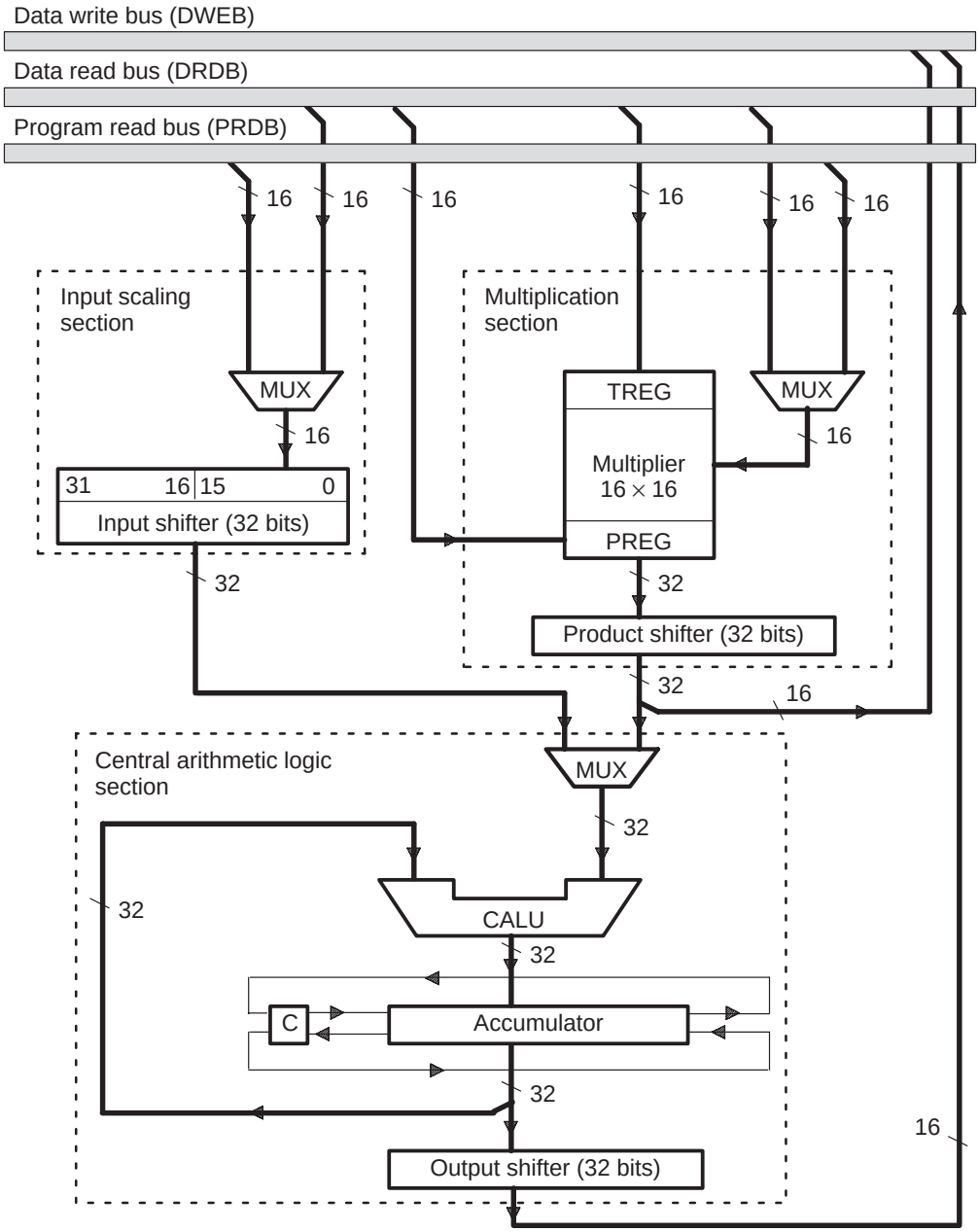
Central Processing Unit

This chapter describes the ’C24x central processing unit (CPU) and its operations. Because of its parallel architectural design, the ’C24x CPU can perform high-speed arithmetic operations within one instruction cycle.

Three fundamental sections of the CPU are presented (see Figure 4–1) along with a description of the auxiliary register arithmetic unit (ARAU), which performs arithmetic operations independently of the central arithmetic logic section. The chapter concludes with a description of status registers ST0 and ST1. These registers contain bits for determining processor modes, addressing pointer values, and indicating various processor conditions and arithmetic logic results.

Topic **Page**

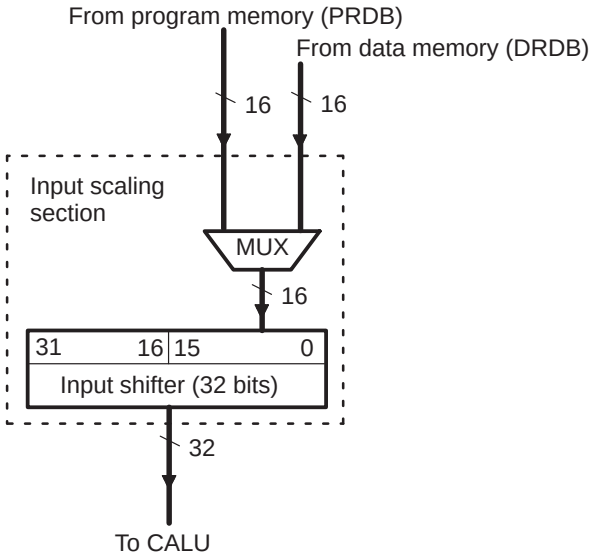
Figure 4–1. Block Diagram of the Input Scaling, Central Arithmetic Logic, and Multiplication Sections of the CPU



4.1 Input Scaling Section

A 32-bit input data-scaling shifter (input shifter) aligns the 16-bit value from memory to the 32-bit central arithmetic logic unit (CALU). This data alignment is necessary for data-scaling arithmetic, as well as aligning masks for logical operations. The input shifter operates as part of the data path between program or data space and the CALU; and therefore, requires no cycle overhead. Described below are the input, output, and shift count of the input shifter. Figure 4–2, *Block Diagram of the Input Scaling Section*, can be used as a reference throughout the discussion.

Figure 4–2. Block Diagram of the Input Scaling Section



Input. Bits 15 through 0 of the input shifter accept a 16-bit input from either of two sources (see Figure 4–2):

- *The data read bus (DRDB).* This input is a value from a data memory location referenced in an instruction operand.
- *The program read bus (PRDB).* This input is a constant value given as an instruction operand.

Output. After a value has been accepted into bits 15 through 0, the input shifter aligns the16-bit value to the 32-bit bus of the CALU as shown in Figure 4–2. The shifter shifts the value left 0 to 16 bits and then sends the 32-bit result to the CALU.

During the left shift, unused LSBs in the shifter are filled with 0s, and unused MSBs in the shifter are either filled with 0s or sign extended, depending on the value of the sign-extension mode bit (SXM) of status register ST1.

Shift count. The shifter can left shift a 16-bit value by 0 to 16 bits. The size of the shift (or the shift count) is obtained from one of two sources:

- *A constant embedded in the instruction word.* Putting the shift count in the instruction word allows you to use specific data-scaling or alignment operations customized for your program code.
- *The four LSBs of the temporary register (TREG).* The TREG-based shift allows the data-scaling factor to be determined dynamically so that it can be adapted to the system's performance.

Sign-extension mode bit. For many (but not all) instructions, the sign-extension mode bit (SXM), bit 10 of status register ST1, determines whether the CALU uses sign extension during its calculations. If SXM = 0, sign extension is suppressed. If SXM = 1, the output of the input shifter is sign extended. Figure 4–3 shows an example of an input value shifted left by eight bits for SXM = 0. The MSBs of the value passed to the CALU are zero filled. Figure 4–4 shows the same shift but with SXM = 1. The value is sign extended during the shift.

Figure 4–3. Operation of the Input Shifter for SXM = 0

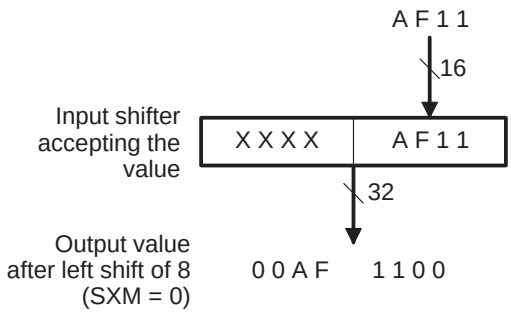
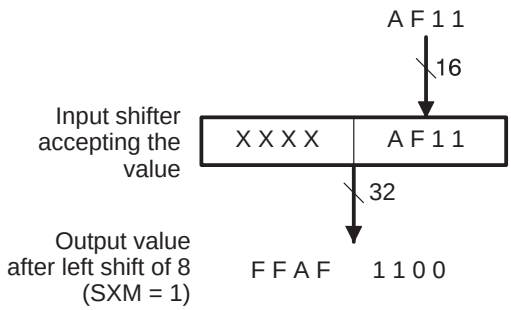


Figure 4–4. Operation of the Input Shifter for SXM = 1

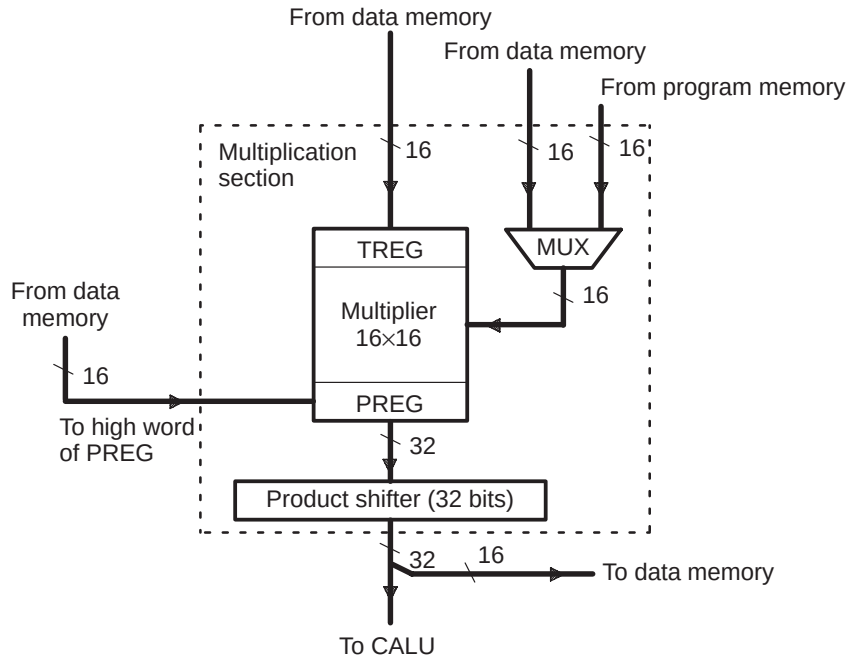


4.2 Multiplication Section

The ’C24x uses a 16-bit × 16-bit hardware multiplier that can produce a signed or unsigned 32-bit product in a single machine cycle. As shown in Figure 4–5, the multiplication section consists of:

- ❑ The 16-bit temporary register (TREG), which holds one of the multiplicands
- ❑ The multiplier, which multiplies the TREG value by a second value from data memory or program memory
- ❑ The 32-bit product register (PREG), which receives the result of the multiplication
- ❑ The product shifter, which scales the PREG value before passing it to the CALU

Figure 4–5. Block Diagram of the Multiplication Section



4.2.1 Multiplier

The 16-bit × 16-bit hardware multiplier can produce a signed or unsigned 32-bit product in a single machine cycle. The two numbers being multiplied are treated as 2s-complement numbers, except during unsigned multiplication (MPYU instruction). Descriptions of the inputs to, and output of, the multiplier follow.

Inputs. The multiplier accepts two 16-bit inputs:

- One input is always from the 16-bit temporary register (TREG). The TREG is loaded before the multiplication with a data-value from the data read bus (DRDB).

- The other input is one of the following:
  - A data-memory value from the data read bus (DRDB)
  - A program memory value from the program read bus (PRDB)

Output. After the two 16-bit inputs are multiplied, the 32-bit result is stored in the product register (PREG). The output of the PREG is connected to the 32-bit product-scaling shifter. Through this shifter, the product is transferred from the PREG to the CALU or to data memory (by the SPH and SPL instructions).

4.2.2 Product-Scaling Shifter

The product-scaling shifter (product shifter) facilitates scaling of the product register (PREG) value. The shifter has a 32-bit input connected to the output of the PREG and a 32-bit output connected to the input of the CALU.

Input. The shifter has a 32-bit input connected to the output of the PREG.

Output. After the shifter completes the shift, all 32 bits of the result can be passed to the CALU, or 16 bits of the result can be stored to data memory.

Shift Modes. This shifter uses one of four product shift modes, summarized in Table 4–1. As shown in the table, these modes are determined by the product shift mode (PM) bits of status register ST1. In the first shift mode (PM = 00), the shifter does not shift the product at all before giving it to the CALU or to data memory. The next two modes cause left shifts (of one or four), which are useful for implementing fractional arithmetic or justifying products. The right-shift mode shifts the product by six bits, enabling the execution of up to 128 consecutive multiply-and-accumulate operations without causing the accumulator to overflow. Note that the content of the PREG remains unchanged; the value is copied to the product shifter and shifted there.

Note:

The right shift in the product shifter is always sign extended, regardless of the value of the sign-extension mode bit (SXM) of status register ST1.

Table 4–1. Product Shift Modes for the Product-Scaling Shifter

PM	Shift	Comments†
00	No shift	Product sent to CALU or data write bus (DWEB) with no shift
01	Left 1	Removes the extra sign bit generated in a 2s-complement multiply to produce a Q31 product
10c	Left 4	Removes the extra four sign bits generated in a 16-bit × 13-bit 2s-complement multiply to produce a Q31 product when multiplying by a 13-bit constant
11	Right 6	Scales the product to allow up to 128 product accumulations without overflowing the accumulator. The right shift is always sign extended, regardless of the value of the sign-extension mode bit (SXM) of status register ST1.

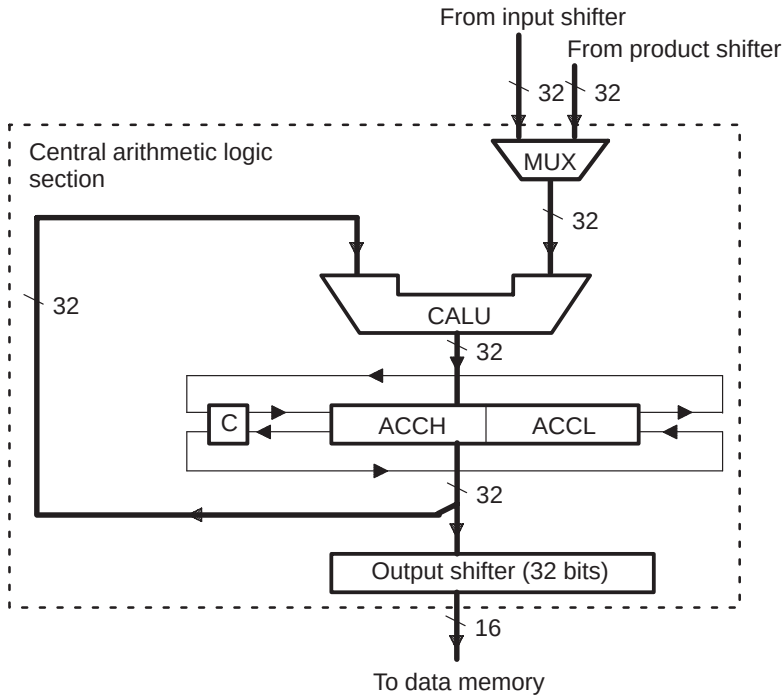
† A Q31 number is a binary fraction in which there are 31 digits to the right of the binary point (the base 2 equivalent of the base 10 decimal point).

4.3 Central Arithmetic Logic Section

The main components of the central arithmetic logic section shown in Figure 4–6 are:

- ❑ The central arithmetic logic unit (CALU), which implements a wide range of arithmetic and logic functions
- ❑ The 32-bit accumulator (ACC), which receives the output of the CALU and is capable of performing bit shifts on its contents with the help of the carry bit (C). Figure 4–6 shows the accumulator's high word (ACCH) and low word (ACCL).
- ❑ The output shifter, which can shift a copy of either the high word or low word of the accumulator before sending it to data memory for storage

Figure 4–6. Block Diagram of the Central Arithmetic Logic Section



4.3.1 Central Arithmetic Logic Unit (CALU)

The CALU implements a wide range of arithmetic and logic functions, most of which execute in a single clock cycle. These functions can be grouped into four categories:

- ❑ 16-bit addition
- ❑ 16-bit subtraction
- ❑ Boolean logic operations
- ❑ Bit testing, shifting, and rotating

Because the CALU can perform Boolean operations, you can perform bit manipulation. For bit shifting and rotating, the CALU uses the accumulator. The CALU is referred to as central because there is an independent arithmetic unit, the auxiliary register arithmetic unit (ARAU), which is described in Section 4.4. A description of the inputs, the output, and an associated status bit of the CALU follows.

Inputs. The CALU has two inputs (see Figure 4–6):

- ❑ One input is always provided by the 32-bit accumulator.
- ❑ The other input is provided by one of the following:
  - ■ The product-scaling shifter (see section 4.2.2)
  - ■ The input data-scaling shifter (see Section 4.1)

Output. Once the CALU performs an operation, it transfers the result to the 32-bit accumulator, which is capable of performing bit shifts of its contents. The output of the accumulator is connected to the 32-bit output data-scaling shifter. Through the output shifter, the accumulator's upper and lower 16-bit words can be individually shifted and stored to data memory.

Sign-extension mode bit. For many but not all instructions, the sign-extension mode bit (SXM), bit 10 of status register ST1, determines whether the CALU uses sign extension during its calculations. If SXM = 0, sign extension is suppressed. If SXM = 1, sign extension is enabled.

4.3.2 Accumulator

Once the CALU performs an operation, it transfers the result to the 32-bit accumulator, which can then perform single-bit shifts or rotations on its contents. Each of the accumulator's upper and lower 16-bit words can be passed to the output data-scaling shifter, where it can be shifted and then stored in data memory. The following describes the status bits and branch instructions associated with the accumulator.

Status bits. Four status bits are associated with the accumulator:

- *Carry bit (C).* C (bit 9 of status register ST1) is affected during:
  - Additions to and subtractions from the accumulator:

    C = 0 When the result of a subtraction generates a borrow

    When the result of an addition does not generate a carry (Exception: When the ADD instruction is used with a shift of 16 and no carry is generated, the ADD instruction has no effect on C.)

    C = 1 When the result of an addition generates a carry

    When the result of a subtraction does not generate a borrow (Exception: When the SUB instruction is used with a shift of 16 and no borrow is generated, the SUB instruction has no effect on C.)

  - Single-bit shifts and rotations of the accumulator value. During a left shift or rotation, the MSB of the accumulator is passed to C; during a right shift or rotation, the LSB is passed to C.

- *Overflow mode bit (OVM).* OVM (bit 11 of status register ST0) determines how the accumulator reflects arithmetic overflows. When the processor is in overflow mode (OVM = 1) and an overflow occurs, the accumulator is filled with one of two specific values:
  - If the overflow is in the positive direction, the accumulator is filled with its most positive value (7FFF FFFFh).
  - If the overflow is in the negative direction, the accumulator is filled with its most negative value (8000 0000h).

- *Overflow flag bit (OV).* OV is bit 12 of status register ST0. When no accumulator overflow is detected, OV is latched at 0. When overflow (positive or negative) occurs, OV is set to 1 and latched.

- *Test/control flag bit (TC).* TC (bit 11 of status register ST1) is set to 0 or 1 depending on the value of a tested bit. In the case of the NORM instruction, if the exclusive-OR of the two MSBs of the accumulator is true, TC is set to 1.

A number of branch instructions are implemented, based on the status of bits C, OV, and TC, and on the value in the accumulator (as compared to 0). For more information about these instructions, see Section 5.4, *Conditional Branches, Calls, and Returns*, on page 5-10.

4.3.3 Output Data-Scaling Shifter

The output data-scaling shifter (output shifter) has a 32-bit input connected to the 32-bit output of the accumulator and a 16-bit output connected to the data bus. The shifter copies all 32 bits of the accumulator and then performs a left shift on its content; it can be shifted from zero to seven bits, as specified in the corresponding store instruction. The upper word (SACH instruction) or lower word (SACL instruction) of the shifter is then stored to data memory. The content of the accumulator remains unchanged.

When the output shifter performs the shift, the MSBs are lost and the LSBs are zero filled. Figure 4–7 shows an example in which the accumulator value is shifted left by four bits and the shifted high word is stored to data memory. Figure 4–8 shows the same accumulator value shifted left by six bits and the shifted low word stored.

Figure 4–7. Shifting and Storing the High Word of the Accumulator

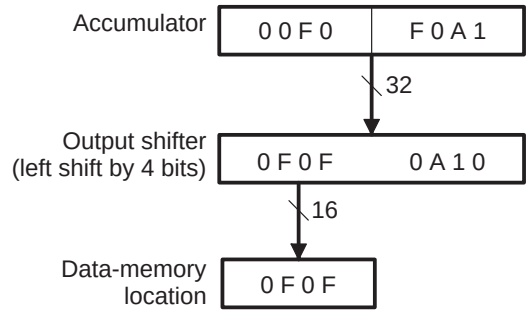
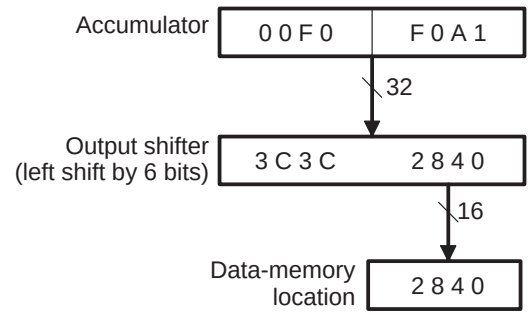


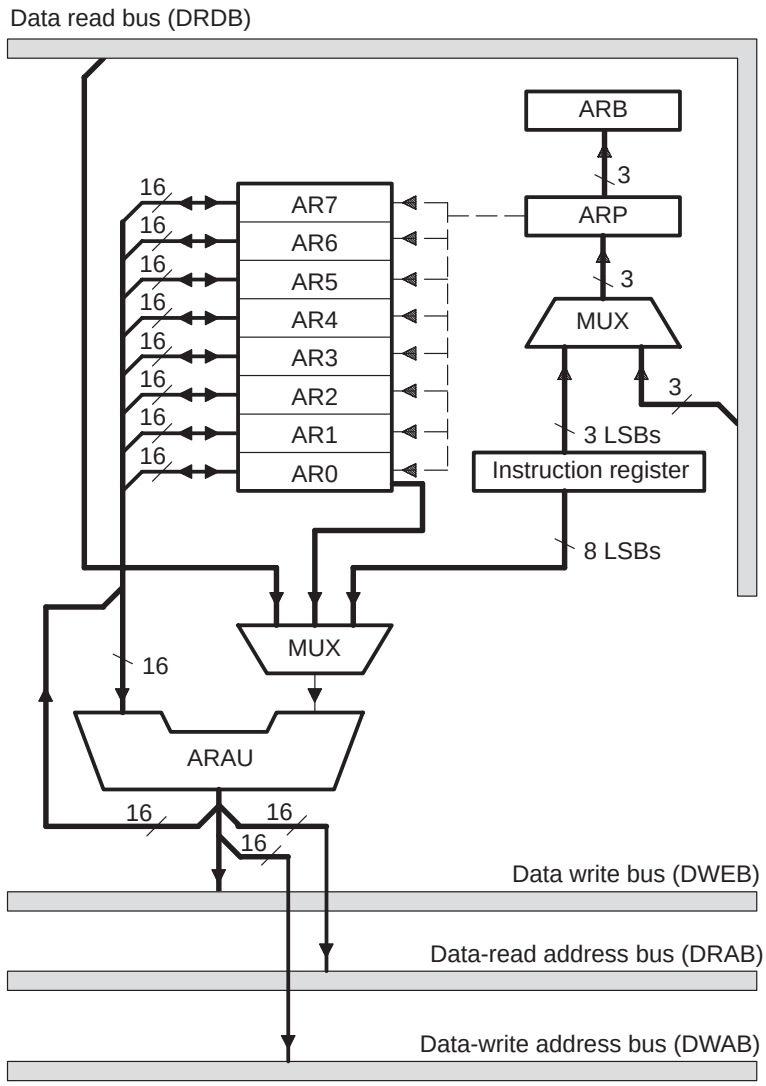
Figure 4–8. Shifting and Storing the Low Word of the Accumulator



4.4 Auxiliary Register Arithmetic Unit (ARAU)

The CPU also contains the ARAU, an arithmetic unit independent of the CALU. The main function of the ARAU is to perform arithmetic operations on eight auxiliary registers (AR7 through AR0) in parallel with operations occurring in the CALU. Figure 4–9 shows the ARAU and related logic.

Figure 4–9. ARAU and Related Logic



The eight auxiliary registers (AR7–AR0) provide flexible and powerful indirect addressing. Any location in the 64K data memory space can be accessed using a 16-bit address contained in an auxiliary register. For details of indirect addressing, see Section 6.3 on page 6-9.

To select a specific auxiliary register, load the 3-bit auxiliary register pointer (ARP) of status register ST0 with a value from 0 through 7. The ARP can be loaded as a primary operation by the MAR instruction (which only performs modifications to the auxiliary registers and the ARP), or by the LST instruction (which can load a data-memory value to ST0 by way of the data read bus, DRDB). The ARP can be loaded as a secondary operation by any instruction that supports indirect addressing.

The register pointed to by the ARP is referred to as the *current auxiliary register* or *current AR*. During the processing of an instruction, the content of the current auxiliary register is used as the address where the data-memory access will take place. The ARAU passes this address to the data-read address bus (DRAB) if the instruction requires a read from data memory; or, it passes the address to the data-write address bus (DWAB) if the instruction requires a write to data memory. After the instruction uses the data value, the contents of the current auxiliary register can be incremented or decremented by the ARAU, which implements unsigned 16-bit arithmetic.

4.4.1 ARAU Functions

The ARAU performs the following operations:

- ❏ Increments or decrements an auxiliary register value by 1 or by an index amount (by way of any instruction that supports indirect addressing)
- ❏ Adds a constant value to an auxiliary register value (ADRK instruction) or subtracts a constant value from an auxiliary register value (SBRK instruction). The constant is an 8-bit value taken from the eight LSBs of the instruction word.
- ❏ Compares the content of AR0 with the content of the current AR and puts the result in the test/control flag bit (TC) of status register ST1 (CMPR instruction). The result is passed to TC by way of the data write bus (DWEB).

Normally, the ARAU performs its arithmetic operations in the decode phase of the pipeline (when the instruction specifying the operations is being decoded). This allows the address to be generated before the decode phase of the next instruction. There is an exception to this rule: During processing of the NORM instruction, the auxiliary register and/or ARP modification is done during the execute phase of the pipeline. For information on the operation of the pipeline, see Section 5.2 on page 5-7.

4.4.2 Auxiliary Register Functions

In addition to using the auxiliary registers to reference data-memory addresses, you can use them for other purposes. For example, you can:

- ❑ Use the auxiliary registers to support conditional branches, calls, and returns by using the CMPR instruction. This instruction compares the content of AR0 with the content of the current AR and puts the result in the test/control flag bit (TC) of status register ST1.
- ❑ Use the auxiliary registers for temporary storage by using the LAR instruction to load values into the registers and the SAR instruction to store AR values to data memory
- ❑ Use the auxiliary registers as software counters, incrementing or decrementing them as necessary

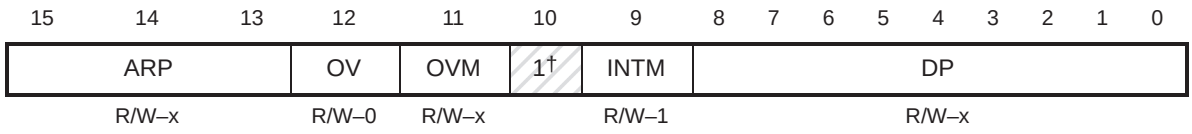
4.5 Status Registers ST0 and ST1

The ’C24x has two status registers, ST0 and ST1, which contain status and control bits. These registers can be stored to, and loaded from, data memory. This allows the status of the machine to be saved and restored for subroutines.

The LST (load status register) instruction writes to ST0 and ST1, and the SST (store status register) instruction reads from ST0 and ST1 (with the exception of the INTM bit, which is not affected by the LST instruction). Many of the individual bits of these registers can be set and cleared using the SETC and CLRC instructions. For example, the sign-extension mode is set with SETC SXM and cleared with CLRC SXM.

Figure 4–10 and Figure 4–11 show the organization of status registers ST0 and ST1, respectively. Several bits in the status registers are reserved; they are always read as logic 1s. The other bits are described in alphabetical order in Table 4–2.

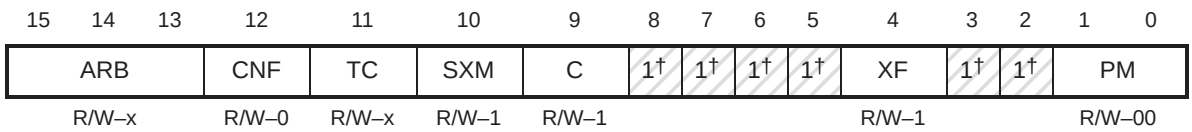
Figure 4–10. Status Register ST0



Note: R = Read access; W = Write access; value following dash (–) is value after reset (x means value not affected by reset).

† This reserved bit is always read as 1. Writes have no effect on it.

Figure 4–11. Status Register ST1



Note: R = Read access; W = Write access; value following dash (–) is value after reset (x means value not affected by reset).

† These reserved bits are always read as 1s. Writes have no effect on them.

Table 4–2. Bit Fields of Status Registers ST0 and ST1

Name	Description
ARB	**Auxiliary register pointer buffer.** Whenever the auxiliary register pointer (ARP) is loaded, the previous ARP value is copied to the ARB, except during an LST (load status register) instruction. When the ARB is loaded by an LST instruction, the same value is also copied to the ARP.
ARP	**Auxiliary register pointer.** This 3-bit field selects which auxiliary register (AR) to use in indirect addressing. When the ARP is loaded, the previous ARP value is copied to the ARB register, except during an LST (load status register) instruction. The ARP may be modified by memory-reference instructions using indirect addressing, and by the MAR (modify auxiliary register) and LST instructions. When the ARB is loaded by an LST instruction, the same value is also copied to the ARP. For more details on the use of ARP in indirect addressing, see Section 6.3, *Indirect Addressing Mode*, on page 6-9.
C	**Carry bit.** This bit is set to 1 if the result of an addition generates a carry, or cleared to 0 if the result of a subtraction generates a borrow. Otherwise, it is cleared after an addition or set after a subtraction, except if the instruction is ADD or SUB with a 16-bit shift. In these cases, ADD can only set and SUB only clear the carry bit, but cannot affect it otherwise. The single-bit shift and rotate instructions also affect this bit, as well as the SETC, CLRC, and LST instructions. The conditional branch, call, and return instructions can execute, based on the status of C. C is set to 1 on reset.
CNF	**On-chip DARAM configuration bit.** This bit determines whether reconfigurable dual-access RAM blocks are mapped to data space or to program space. The CNF bit may be modified by the SETC CNF, CLRC CNF, and LST instructions. Reset clears the CNF bit to 0. For more information about CNF and the dual-access RAM blocks, see Chapter 3, *Memory and I/O Space*s. CNF = 0 Reconfigurable dual-access RAM blocks are mapped to data space. CNF = 1 Reconfigurable dual-access RAM blocks are mapped to program space.
DP	**Data page pointer.** When an instruction uses direct addressing, the 9-bit DP field is concatenated with the seven LSBs of the instruction word to form a full 16-bit data-memory address. For more details, see Section 6.2, *Direct Addressing Mode*, on page 6-4. The LST and LDP (load DP) instructions can modify the DP field.
INTM	**Interrupt mode bit.** This bit enables or disables all maskable interrupts. INTM is set and cleared by the SETC INTM and CLRC INTM instructions, respectively. INTM has no effect on the nonmaskable interrupts $\overline{RS}$ and $\overline{NMI}$ or on interrupts initiated by software. INTM is unaffected by the LST (load status register) instruction. INTM is set to 1 when an interrupt trap is taken (except in the case of the TRAP instruction) and at reset. INTM = 0 All unmasked interrupts are enabled. INTM = 1 All maskable interrupts are disabled.
OV	**Overflow flag bit.** This bit holds a latched value that indicates whether overflow has occurred in the CALU. OV is set to 1 when an overflow occurs in the CALU. Once an overflow occurs, the OV bit remains set until it is cleared by a reset, a conditional branch on overflow (OV) or no overflow (NOV), or an LST instruction.

Table 4–2. Bit Fields of Status Registers ST0 and ST1 (Continued)

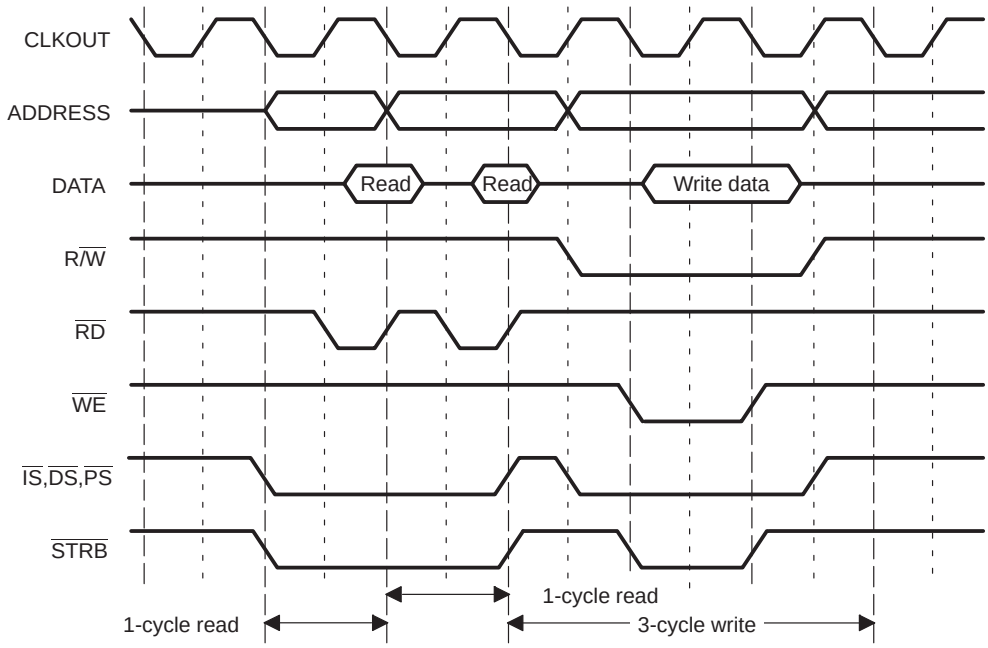
Name	Description
OVM	**Overflow mode bit.** OVM determines how overflows in the CALU are handled. The SETC and CLRC instructions set and clear this bit, respectively. An LST instruction can also be used to modify OVM. OVM = 0 Results overflow normally in the accumulator. OVM = 1 The accumulator is set to either its most positive or negative value upon encountering an overflow. (See subsection 4.3.2, *Accumulator*, on page 4-9.)
PM	**Product shift mode.** PM determines the amount that the PREG value is shifted on its way to the CALU or to data memory. Note that the content of the PREG remains unchanged; the value is copied to the product shifter and shifted there. PM is loaded by the SPM and LST instructions. The PM bits are cleared by reset. PM = 00 The multiplier's 32-bit product is passed to the CALU or to data memory with no shift. PM = 01 The output of the PREG is left shifted one place (with the LSBs zero filled) before being passed to the CALU or to data memory. PM = 10 The output of the PREG is left shifted four bits (with the LSBs zero filled) before being passed to the CALU or to data memory. PM = 11 This mode produces a right shift of six bits, sign extended.
SXM	**Sign-extension mode bit.** SXM does not affect the basic operation of certain instructions. For example, the ADDS instruction suppresses sign extension regardless of SXM. This bit is set by the SETC SXM instruction, cleared by the CLRC SXM instruction, and may be loaded by the LST instruction. SXM is set to 1 by reset. SXM = 0 This mode suppresses sign extension. SXM = 1 In this mode, data values that are shifted in the input shifter are sign extended before they are passed to the CALU.
TC	**Test/control flag bit.** The TC bit is set to 1 if a bit tested by BIT or BITT is a 1, if a compare condition tested by CMPR exists between the current auxiliary register and AR0, or if the exclusive-OR function of the two MSBs of the accumulator is true when tested by a NORM instruction. The conditional branch, call, and return instructions can execute, based on the condition of the TC bit. The TC bit is affected by the BIT, BITT, CMPR, LST, and NORM instructions.
XF	**XF pin status bit.** This bit determines the state of the XF pin, which is a general-purpose output pin. XF is set by the SETC XF instruction and cleared by the CLRC XF instruction. XF can also be modified with an LST instruction. XF is set to 1 by reset.

4.6 External Memory Interface Operation

Some of the ’24x DSP controller devices have an external memory interface. This section explains the behavior of the ’24x external memory timings based on the PS, DS, IS, BR, STRB, RD, LR, and R/W signals.

All bus cycles comprise integral numbers of CLKOUT cycles. One CLKOUT cycle is defined to be from one falling edge of CLKOUT to the next falling edge of CLKOUT. For full-speed, 0-wait-state operation, reads require one cycle. A write immediately preceded by a read or immediately followed by a read requires three cycles. (Refer to Figure 4–12 on page 4-18, Figure 4–13 on page 4-20, and Figure 4–14 on page 4-20 for read and write timing cycles.) These timing models explain CPU and external memory signal behavior during back-to-back write, without wait state, and with one wait state. External READY timings in ’C24x devices can be satisfied only if internal wait state is at least one. Refer to the respective datasheet for specific timing values.

Figure 4–12. External Interface Operation for Read-Read-Write (Zero Wait States)



For read cycles, $\overline{\text{STRB}}$ goes low and ADDRESS becomes valid with the falling edge of CLKOUT. For 0-wait-state read cycles, the $\overline{\text{RD}}$† signal goes low with the rising edge of CLKOUT and then goes high at the next falling edge of CLKOUT. For 1-wait-state (multicycle) read cycles, the $\overline{\text{RD}}$ stays low but goes high with the falling edge of CLKOUT before the next cycle, even if the cycles are contiguous. Read data is sampled at the rising edge of $\overline{\text{RD}}$.

† In some devices (F240), the external $\overline{\text{RD}}$ is replaced with an inverted R/W signal.

The R/$\overline{W}$ signal goes high at least one-half cycle of CLKOUT before any read cycle; for contiguous read cycles, $\overline{STRB}$ stays low. At the end of a read cycle or sequence of reads, $\overline{STRB}$ and $\overline{RD}$ go high on the falling edge of CLKOUT.

Write cycles always have at least one inactive (pad) cycle of CLKOUT before and after the actual write operation, including contiguous writes. This allows a smooth transition between the write and any adjacent bus operations or other writes. For this pad cycle, $\overline{STRB}$ and $\overline{WE}$ are always high. The R/$\overline{W}$ signal always changes state on the rising edge of CLKOUT during the pad cycle before and after a write or series of writes. This prevents bus contention during a transition between read and write operations. Note that for a series of writes, R/$\overline{W}$ 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, ADDRESS becomes valid on the rising edge of CLKOUT. If a read immediately follows a write or series of writes, ADDRESS becomes valid for that read cycle one half-cycle of CLKOUT early — that is, on the rising edge, rather than on the falling edge, of CLKOUT. This is an exception to the usual read cycle address timing.

For the actual write operation, $\overline{STRB}$ and $\overline{WE}$ both go low on the falling edge of CLKOUT and stay low until the next falling edge of CLKOUT (for 0-wait-state write cycles). For 1-wait-state (multicycle) writes, $\overline{STRB}$ and $\overline{WE}$ remain low but go high again on the falling edge of CLKOUT at the beginning of the pad cycle. *Write data* is driven approximately at the falling edge of $\overline{STRB}$ and $\overline{WE}$ and is held for approximately one half-cycle of CLKOUT after $\overline{STRB}$ and $\overline{WE}$ go high (refer to the TMS320C24x data sheet for actual timing specifications).

Transitions on the external parallel interface control outputs (CLKOUT, $\overline{STRB}$, $\overline{WE}$, and $\overline{RD}$) are all initiated by the same internal clocks. Since these signals also use the same output buffer circuitry, they all switch within close tolerances of each other, as specified in the TMS320C24x data sheet.

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 that generates these outputs is different from the circuitry used for the control outputs. Therefore, transitions on the address bus and related outputs typically occur later than control-line transitions.

Figure 4–13. External Interface Operation for Write-Write-Read (Zero Wait States)

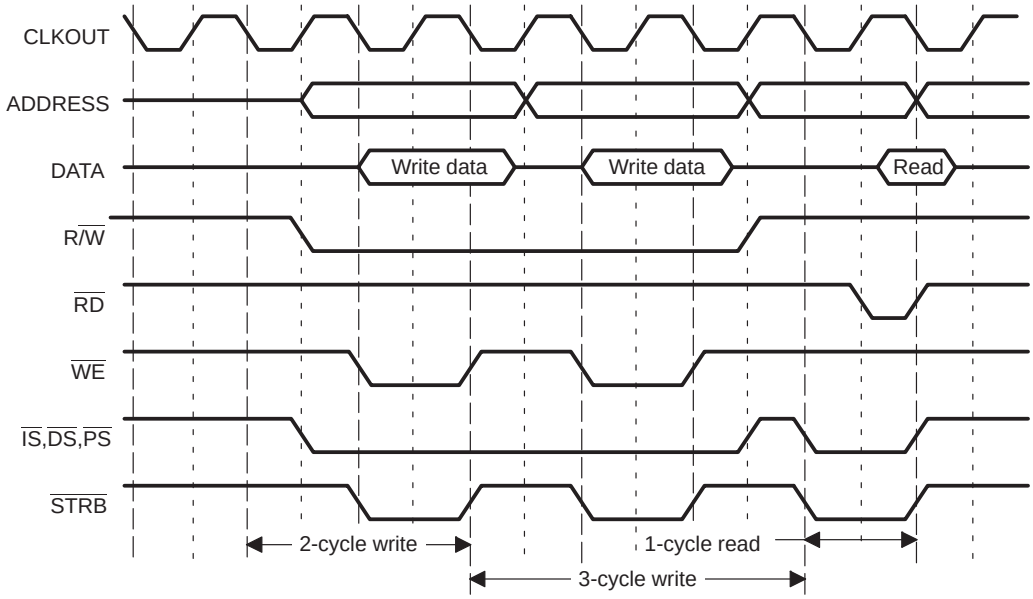
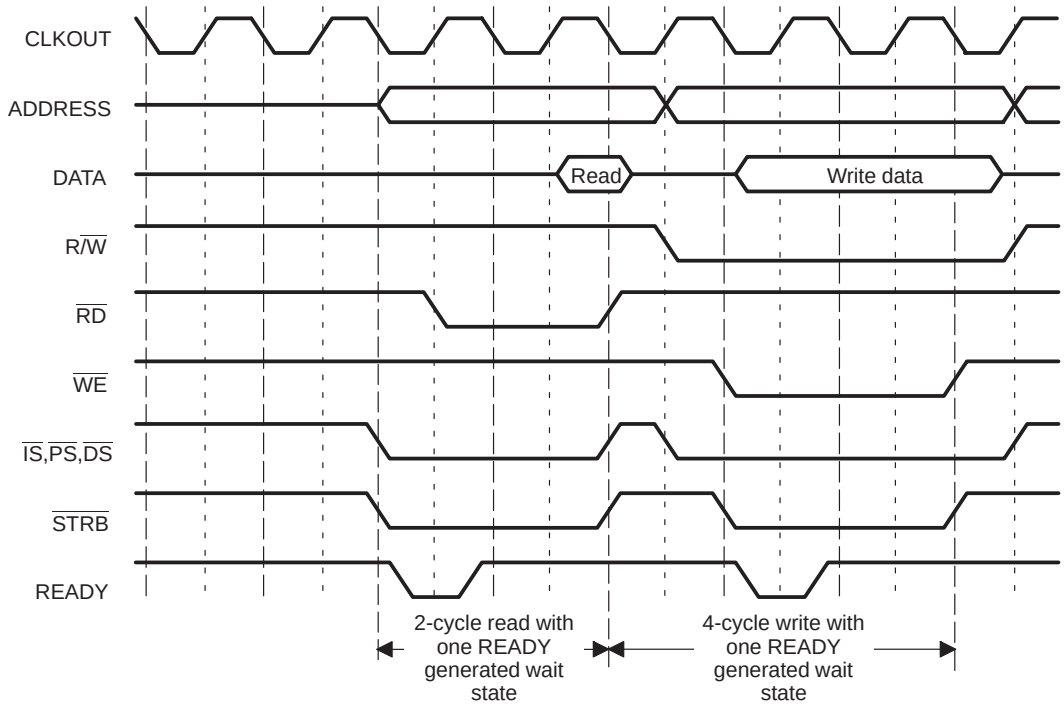


Figure 4–14. External Interface Operation for Read-Write (One Wait State)



Chapter 5

Program Control

This chapter discusses the processes and features involved in controlling the flow of a program on the ’C24x.

Program control involves controlling the order in which one or more blocks of instructions are executed. Normally, the flow of a program is sequential; the ’C24x executes instructions at consecutive program-memory addresses. At times, a program must branch to a nonsequential address and then execute instructions sequentially at that new location. For this purpose, the ’C24x supports branches, calls, returns, repeats, and interrupts.

Topic **Page**

5.1 Program-Address Generation

Program flow requires the processor to generate the next program address (sequential or nonsequential) while executing the current instruction. Program-address generation is illustrated in Figure 5–1 and summarized in Table 5–1.

Figure 5–1. Program-Address Generation Block Diagram

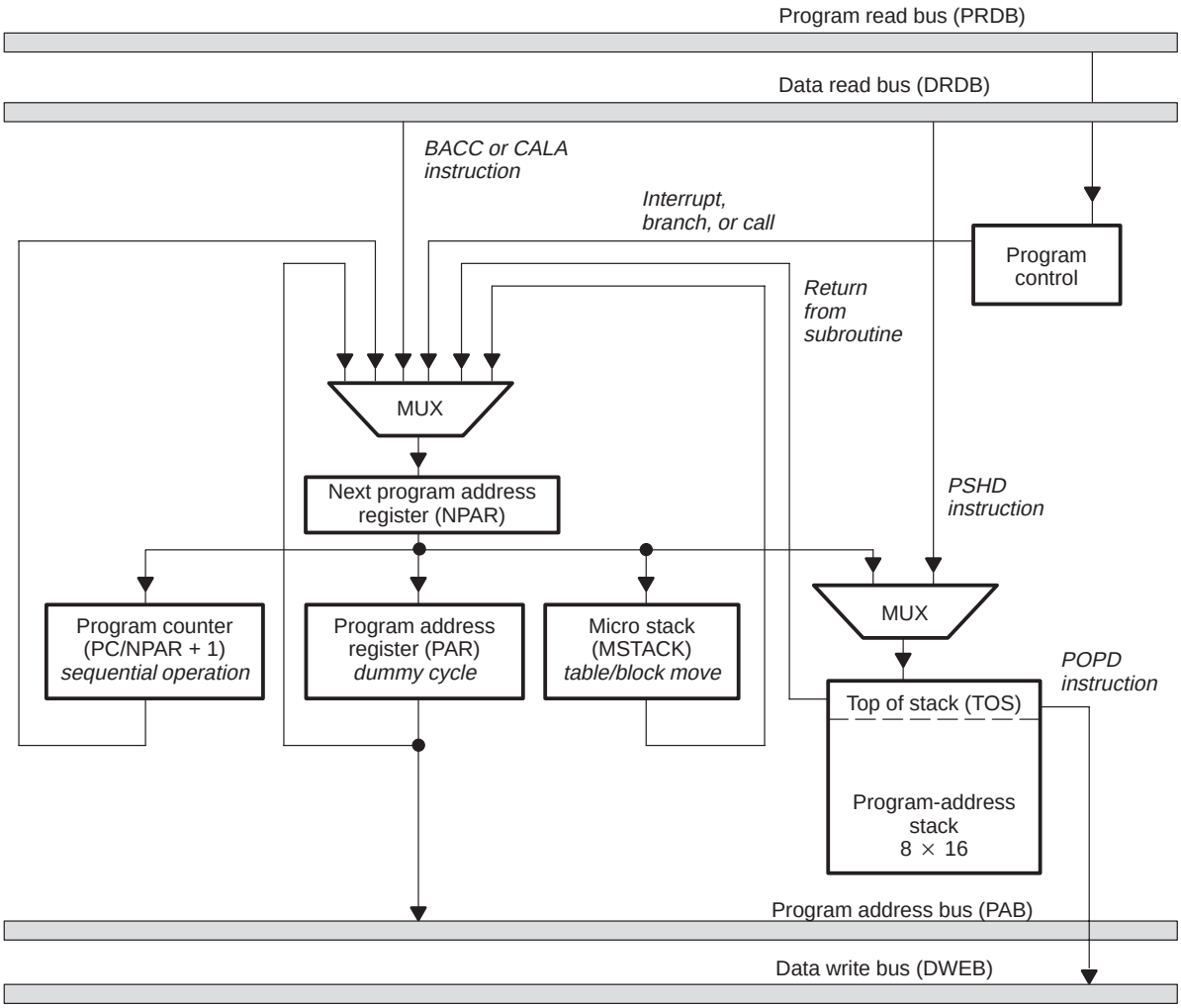


Table 5–1. Program-Address Generation Summary

Operation	Program-Address Source
Sequential operation	PC (contains program address +1)
Dummy cycle	PAR (contains program address)
Return from subroutine	Top of the stack (TOS)
Return from table move or block move	Microstack (MSTACK)
Branch or call to address specified in instruction	Branch or call instruction by way of the program read bus (PRDB)
Branch or call to address specified in lower half of the accumulator	Low accumulator by way of the data read bus (DRDB)
Branch to interrupt service routine	Interrupt vector location by way of the program read bus (PRDB)

The ’C24x program-address generation logic uses the following hardware:

- **Program counter (PC).** The ’C24x has a 16-bit program counter (PC) that addresses internal and external program memory when fetching instructions.
- **Program address register (PAR).** The PAR drives the program address bus (PAB). The PAB is a 16-bit bus that provides program addresses for both reads and writes.
- **Stack.** The program-address generation logic includes a 16-bit-wide, 8-level hardware stack for storing up to eight return addresses. In addition, you can use the stack for temporary storage.
- **Microstack (MSTACK).** Occasionally, the program-address generation logic uses the 16-bit-wide, 1-level MSTACK to store one return address.
- **Repeat counter (RPTC).** The 16-bit RPTC is used with the repeat (RPT) instruction to determine how many times the instruction following RPT is repeated.

5.1.1 Program Counter (PC)

The program-address generation logic uses the 16-bit program counter (PC) to address internal and external program memory. The PC holds the address of the next instruction to be executed. Through the program address bus (PAB), an instruction is fetched from that address in program memory and loaded into the instruction register. When the instruction register is loaded, the PC holds the next address.

The ’C24x can load the PC in a number of ways, to accommodate sequential and nonsequential program flow. Table 5–2 shows what is loaded to the PC according to the code operation performed.

Table 5–2. Address Loading to the Program Counter

Code Operation	Address Loaded to the PC
Sequential execution	The PC is loaded with PC + 1 if the current instruction has one word or PC + 2 if the current instruction has two words.
Branch	The PC is loaded with the long immediate value directly following the branch instruction.
Subroutine call and return	For a call, the address of the next instruction is pushed from the PC onto the stack, and then the PC is loaded with the long immediate value directly following the call instruction. A return instruction pops the return address back into the PC to return to the calling sequence of code.
Software or hardware interrupt	The PC is loaded with the address of the appropriate interrupt vector location. At this location is a branch instruction that loads the PC with the address of the corresponding interrupt service routine.
Computed GOTO	The content of the lower 16 bits of the accumulator is loaded into the PC. Computed GOTO operations can be performed using the BACC (branch to address in accumulator) or CALA (call subroutine at location specified by the accumulator) instructions.

5.1.2 Stack

The ’C24x has a 16-bit-wide, 8-level-deep hardware stack. The program-address generation logic uses the stack for storing return addresses when a subroutine call or interrupt occurs. When an instruction forces the CPU into a subroutine or an interrupt forces the CPU into an interrupt service routine, the return address is loaded to the top of the stack automatically, without the need for additional cycles. When the subroutine or interrupt service routine is com-

plete, a return instruction transfers the return address from the top of the stack to the program counter.

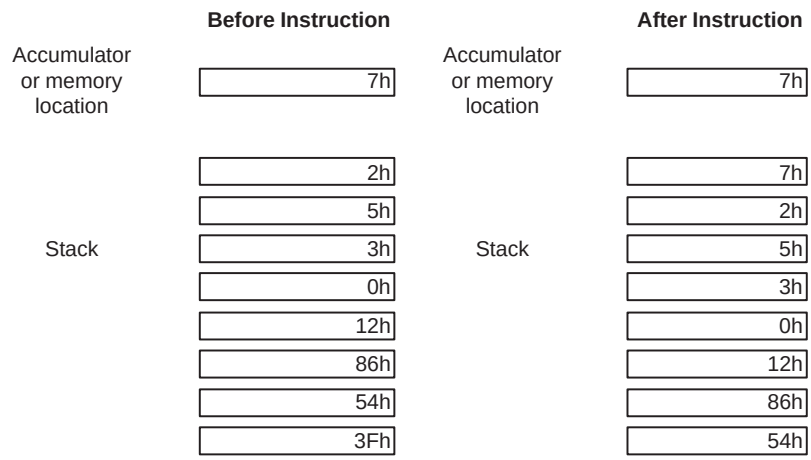
When the eight levels are not used for return addresses, the stack may be used for saving context data during a subroutine or interrupt service routine or for other storage purposes.

You can access the stack with two sets of instructions:

- **PUSH and POP.** The PUSH instruction copies the 16 LSBs of the accumulator to the top of the stack. The POP instruction copies the value on the top of the stack to the 16 LSBs of the accumulator.
- **PSHD and POPD.** These instructions allow you to build a stack in data memory for the nesting of subroutines or interrupts beyond eight levels. The PSHD instruction pushes a data-memory value onto the top of the stack. The POPD instruction pops a value from the top of the stack to data memory.

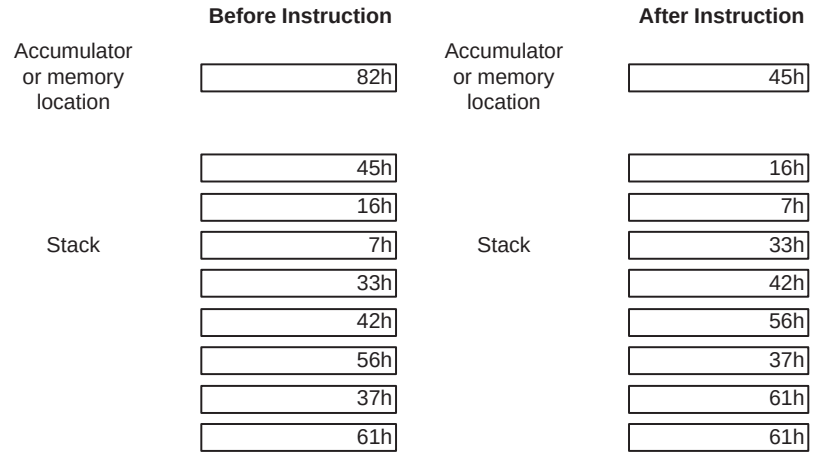
Whenever a value is pushed onto the top of the stack (by an instruction or by the address-generation logic), the content of each level is pushed down one level, and the bottom (eighth) location of the stack is lost. Therefore, data is lost (stack overflow occurs) if more than eight successive pushes occur before a pop. Figure 5–2 shows a push operation.

Figure 5–2. A Push Operation



Pop operations are the reverse of push operations. A pop operation copies the value at each level to the next higher level. Any pop after seven sequential pops yields the value that was originally at the bottom of the stack because, by then, the bottom value has been copied upward to all of the stack levels. Figure 5–3 shows a pop operation.

Figure 5–3. A Pop Operation



5.1.3 Microstack (MSTACK)

The program-address generation logic uses the 16-bit-wide, 1-level-deep MSTACK to store a return address before executing certain instructions. These instructions use the program-address generation logic to provide a second address in a 2-operand instruction. These instructions are: BLDD, BLPD, MAC, MACD, TBLR, and TBLW. When repeated, these instructions use the PC to increment the first operand address and can use the auxiliary register arithmetic unit (ARAU) to generate the second operand address. When these instructions are used, the return address (the address of the next instruction to be fetched) is pushed onto the MSTACK. Upon completion of the repeated instruction, the MSTACK value is popped back into the program-address generation logic.

The MSTACK operations are not visible to you. Unlike the stack, the MSTACK can be used only by the program-address generation logic; there are no instructions that allow you to use the MSTACK for storage.

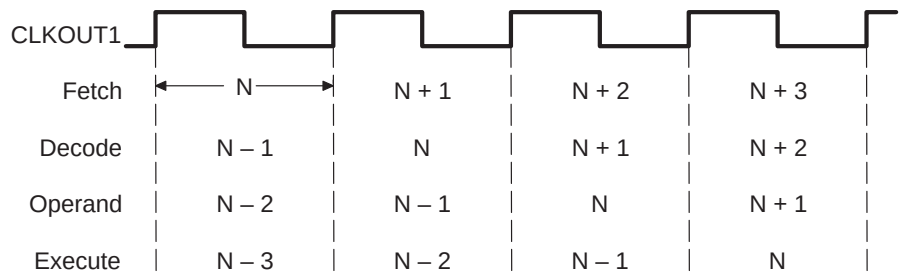
5.2 Pipeline Operation

Instruction pipelining consists of a sequence of bus operations that occur during the execution of an instruction. The ’C24x pipeline has four independent stages: instruction-fetch, instruction-decode, operand-fetch, and instruction-execute. Because the four stages are independent, these operations can overlap. During any given cycle, one to four different instructions can be active, each at a different stage of completion. Figure 5–4 shows the operation of the 4-level-deep pipeline for single-word, single-cycle instructions executing with no wait states.

The pipeline is essentially invisible to you, except in the following cases:

- A single-word, single-cycle instruction immediately following a modification of the global-memory allocation register (GREG) uses the previous global map.
- The NORM instruction modifies the auxiliary register pointer (ARP) and uses the current auxiliary register (the one pointed to by the ARP) during the execute phase of the pipeline. If the next two instruction words change the values in the current auxiliary register or the ARP, they will do so during the instruction decode phase of the pipeline (before the execution of NORM). This would cause NORM to use the wrong auxiliary register value and the following instructions to use the wrong ARP value.

Figure 5–4. Four-Level Pipeline Operation



The CPU is implemented using 2-phase static logic. The 2-phase operation of the ’C24x CPU consists of a master phase in which all commutation logic is executed, and a slave phase in which results are latched. Therefore, sequential operations require sequential master cycles. Although sequential operations require a deeper pipeline, 2-phase operation provides more time for the computational logic to execute. This allows the ’C24x to run at faster clock rates, despite having a deeper pipeline that imposes a penalty on branches and subroutine calls.

5.3 Branches, Calls, and Returns

Branches, calls, and returns break the sequential flow of instructions by transferring control to another location in program memory. A *branch* only transfers control to the new location. A *call* also saves the return address (the address of the instruction following the call) to the top of the hardware stack. Every called subroutine or interrupt service routine is concluded with a *return* instruction, which pops the return address off the stack and back into the program counter (PC).

The ’C24x has two types of branches, calls, and returns:

- **Unconditional.** An unconditional branch, call, or return is always executed. The unconditional branch, call, and return instructions are described in sections 5.3.1, 5.3.2, and 5.3.3, respectively.
- **Conditional.** A conditional branch, call, or return is executed only if certain specified conditions are met. The conditional branch, call, and return instructions are described in detail in Section 5.4, *Conditional Branches, Calls, and Returns*, on page 5-10.

5.3.1 Unconditional Branches

When an unconditional branch is encountered, it is always executed. During the execution, the PC is loaded with the specified program-memory address and program execution begins at that address. The address loaded into the PC may come from either the second word of the branch instruction or the lower sixteen bits of the accumulator.

By the time the branch instruction reaches the execute phase of the pipeline, the next two instruction words have already been fetched. These two instruction words are flushed from the pipeline so that they are not executed, and then execution continues at the branched-to address. The unconditional branch instructions are B (branch) and BACC (branch to location specified by accumulator).

5.3.2 Unconditional Calls

When an unconditional call is encountered, it is always executed. When the call is executed, the PC is loaded with the specified program-memory address and program execution begins at that address. The address loaded into the PC may come from either the second word of the call instruction or the lower 16 bits of the accumulator. Before the PC is loaded, the return address is saved in the stack. After the subroutine or function is executed, a return instruction loads the PC with the return address from the stack, and execution resumes at the instruction following the call.

By the time the unconditional call instruction reaches the execute phase of the pipeline, the next two instruction words have already been fetched. These two instruction words are flushed from the pipeline with the following results:

- ❑ They are not executed.
- ❑ The return address is stored to the stack.
- ❑ Execution continues at the beginning of the called function.

The unconditional call instructions are CALL and CALA (call subroutine at location specified by accumulator).

5.3.3 Unconditional Returns

When an unconditional return (RET) instruction is encountered, it is always executed. When the return is executed, the PC is loaded with the value at the top of the stack, and execution resumes at that address.

By the time the unconditional return instruction reaches the execute phase of the pipeline, the next two instruction words have already been fetched. The two instruction words are flushed from the pipeline with the following results:

- ❑ They are not executed.
- ❑ The return address is taken from the stack.
- ❑ Execution continues in the calling function.

5.4 Conditional Branches, Calls, and Returns

The ’C24x provides branch, call, and return instructions that execute only if one or more conditions are met. You specify the conditions as operands of the conditional instruction. Table 5–3 lists the conditions that you can use with these instructions and their corresponding operand symbols.

Table 5–3. Conditions for Conditional Calls and Returns

Operand Symbol	Condition	Description
EQ	ACC = 0	Accumulator equal to 0
NEQ	ACC ≠ 0	Accumulator not equal to 0
LT	ACC < 0	Accumulator less than 0
LEQ	ACC ≤ 0	Accumulator less than or equal to 0
GT	ACC > 0	Accumulator greater than 0
GEQ	ACC ≥ 0	Accumulator greater than or equal to 0
C	C = 1	Carry bit set to 1
NC	C = 0	Carry bit cleared to 0
OV	OV = 1	Accumulator overflow detected
NOV	OV = 0	No accumulator overflow detected
BIO	$\overline{\text{BIO}}$ low	$\overline{\text{BIO}}$ pin is low
TC	TC = 1	Test/control flag set to 1
NTC	TC = 0	Test/control flag cleared to 0

5.4.1 Using Multiple Conditions

Multiple conditions can be listed as operands of the conditional instructions. If multiple conditions are listed, all conditions must be met for the instruction to execute. Note that only certain combinations of conditions are meaningful. See Table 5–4 on page 5-11. For each combination, the conditions must be selected from Group 1 and Group 2 as follows:

- **Group 1.** You can select up to two conditions. Each of these conditions must be from a different category (A or B); you cannot have two conditions from the same category. For example, you can test EQ and OV at the same time, but you cannot test GT and NEQ at the same time.

- **Group 2.** You can select up to three conditions. Each of these conditions must be from a different category (A, B, or C); you cannot have two conditions from the same category. For example, you can test TC, C, and BIO at the same time, but you cannot test C and NC at the same time.

Table 5–4. Groupings of Conditions

Group 1		Group 2		
Category A	**Category B**	**Category A**	**Category B**	**Category C**
EQ	OV	TC	C	BIO
NEQ	NOV	NTC	NC	
LT				
LEQ				
GT				
GEQ				

5.4.2 Stabilization of Conditions

A conditional instruction must be able to test the most recent values of the status bits. Therefore, the conditions cannot be considered stable until the fourth, or execution stage of the pipeline, one cycle after the previous instruction has been executed. The pipeline controller stops the decoding of any instructions following the conditional instruction until the conditions are stable.

5.4.3 Conditional Branches

A branch instruction transfers program control to any location in program memory. Conditional branch instructions are executed only when one or more user-specified conditions are met (see Table 5–3 on page 5-10). If all the conditions are met, the PC is loaded with the second word of the branch instruction, which contains the address to branch to, and execution continues at this address.

By the time the conditions have been tested, the two instruction words following the conditional branch instruction have already been fetched in the pipeline. If all the conditions are met, these two instruction words are flushed from the pipeline so that they are not executed, and then execution continues at the branched-to address. If the conditions are *not* met, the two instruction words are executed instead of the branch. Because conditional branches use conditions determined by the execution of the previous instructions, a conditional branch takes one more cycle than an unconditional one.

The conditional branch instructions are BCND (branch conditionally) and BANZ (branch if currently selected auxiliary register is not equal to 0). The BANZ instruction is useful for implementing loops.

5.4.4 Conditional Calls

The conditional call (CC) instruction is executed only when the specified condition or conditions are met (see Table 5–3 on page 5-10). This allows your program to choose among multiple subroutines based on the data being processed. If all the conditions are met, the PC is loaded with the second word of the call instruction, which contains the starting address of the subroutine. Before branching to the subroutine, the processor stores the address of the instruction following the call instruction (the return address) to the stack. The function must end with a return instruction, which takes the return address off the stack and forces the processor to resume execution of the calling program.

By the time the conditions of the conditional call instruction have been tested, the two instruction words following the call instruction have already been fetched in the pipeline. If all the conditions are met, these two instruction words are flushed from the pipeline so that they are not executed, and then execution continues at the beginning of the called function. If the conditions are *not* met, the two instructions are executed instead of the call. Because there is a wait cycle for conditions to become stable, the conditional call takes one more cycle than the unconditional one.

5.4.5 Conditional Returns

Returns are used in conjunction with calls and interrupts. A call or interrupt stores a return address to the stack and then transfers program control to a new location in program memory. The called subroutine or the interrupt service routine concludes with a return instruction, which pops the return address off the top of the stack and into the program counter (PC).

The conditional return instruction (RETC) is executed only when one or more conditions are met (see Table 5–3 on page 5-10). By using the RETC instruction, you can give a subroutine or interrupt service routine more than one possible return path. The path chosen then depends on the data being processed. In addition, you can use a conditional return to avoid conditionally branching to/around the return instruction at the end of the subroutine or interrupt service routine.

If all the conditions are met for execution of the RETC instruction, the processor loads the return address from the stack to the PC and resumes execution of the calling or interrupted program.

RETC, like RET, is a single-word instruction. However, because of the potential PC discontinuity, it operates with the same effective execution time as the conditional branch (BCND) and the conditional call (CC). By the time the conditions of the conditional return instruction have been tested, the two instruction words following the return instruction have already been fetched in the pipeline. If all the conditions are met, these two instruction words are flushed from the pipeline so that they are not executed, and then execution of the calling program continues. If the conditions are *not* met, the two instructions are executed instead of the return. Because there is a wait cycle for conditions to become stable, the conditional return takes one more cycle than the unconditional one.

5.5 Repeating a Single Instruction

The 'C24x repeat instruction (RPT) allows the execution of a single instruction N + 1 times, where N is specified as an operand of the RPT instruction. When RPT is executed, the repeat counter (RPTC) is loaded with N. RPTC is then decremented every time the repeated instruction is executed, until RPTC equals 0. RPTC can be used as a 16-bit counter when the count value is read from a data-memory location; if the count value is specified as a constant operand, it is in an 8-bit counter.

The repeat feature is useful with instructions such as NORM (normalize contents of accumulator), MACD (multiply and accumulate with data move), and SUBC (conditional subtract). When instructions are repeated, the address and data buses for program memory are free to fetch a second operand in parallel with the address and data buses for data memory. This allows instructions such as MACD and BLPD to effectively execute in a single cycle when repeated.

5.6 Interrupts

The ’C24x DSP supports both hardware and software interrupts. The hardware interrupts INT1 – INT6, along with NMI, TRAP, and RS, provide a flexible interrrupt scheme. The software interrupts offer flexibility to access interrupt vectors using software instructions. Table 5–5, *’C24x Interrupt Locations and Priorities*, on page 5-15 shows the vectors supported by the DSP core. Since most of the ’C24x DSPs come with multiple peripherals, the core interrupts (INT1–IN6) are expanded using additional system or peripheral interrupt logic. Although the core interrupts are the same, the peripheral interrupt structure varies slightly among ’C240 and ’C24x class of DSP controllers. For details on how these core interrupts are multiplexed to meet peripheral interrupt requirements, refer to *TMS320F/C240 DSP Controllers, Peripheral Library and Specific Devices,* (literature number SPRU161), and *TMS320F241,C242,F243 DSP Controllers, System, and Peripherals*, (literature number SPRU276).

The maskable core interrupt structure is supported by two registers, IFR and IMR. The core interrupt logic has a global interrupt enable bit in the ST register. Details of these registers are presented in the tables and sections that follow.

Table 5–5. ’C24x Interrupt Locations and Priorities

K†	Vector Location	Name	Priority	Function
0	0h	$\overline{\text{RS}}$	1 (highest)	Hardware reset (nonmaskable)
1	2h	INT1	4	Maskable interrupt level #1‡
2	4h	INT2	5	Maskable interrupt level #2‡
3	6h	INT3	6	Maskable interrupt level #3‡
4	8h	INT4	7	Maskable interrupt level #4‡
5	Ah	INT5	8	Maskable interrupt level #5‡
6	Ch	INT6	9	Maskable interrupt level #6‡
7	Eh		10	Reserved
8	10h	INT8	–	User-defined software interrupt

† The K value is the operand used in an INTR instruction that branches to the corresponding interrupt vector location.

‡ Maskable interrupts are customized for each ’C24x DSP device with additional interrupt expansion logic.

Table 5–5. ’C24x Interrupt Locations and Priorities (Continued)

K†	Vector Location	Name	Priority	Function
9	12h	INT9	–	User-defined software interrupt
10	14h	INT10	–	User-defined software interrupt
11	16h	INT11	–	User-defined software interrupt
12	18h	INT12	–	User-defined software interrupt
13	1Ah	INT13	–	User-defined software interrupt
14	1Ch	INT14	–	User-defined software interrupt
15	1Eh	INT15	–	User-defined software interrupt
16	20h	INT16	–	User-defined software interrupt
17	22h	TRAP	–	TRAP instruction vector
18	24h	NMI	3	Nonmaskable interrupt
19	26h		2	Reserved
20	28h	INT20	–	User-defined software interrupt
21	2Ah	INT21	–	User-defined software interrupt
22	2Ch	INT22	–	User-defined software interrupt
23	2Eh	INT23	–	User-defined software interrupt
24	30h	INT24	–	User-defined software interrupt
25	32h	INT25	–	User-defined software interrupt
26	34h	INT26	–	User-defined software interrupt
27	36h	INT27	–	User-defined software interrupt
28	38h	INT28	–	User-defined software interrupt
29	3Ah	INT29	–	User-defined software interrupt
30	3Ch	INT30	–	User-defined software interrupt
31	3Eh	INT31	–	User-defined software interrupt

† The K value is the operand used in an INTR instruction that branches to the corresponding interrupt vector location.

5.7 CPU Interrupt Registers

There are two CPU registers for controlling interrupts:

- ❑ The interrupt flag register (IFR) contains flag bits that indicate when maskable interrupt requests have reached the CPU on levels INT1 through INT6.
- ❑ The interrupt mask register (IMR) contains mask bits that enable or disable each of the interrupt levels (INT1 through INT6).

5.7.1 Interrupt Flag Register (IFR)

The interrupt flag register (IFR), a 16-bit, memory-mapped register at address 0006h in data-memory space, is used to identify and clear pending interrupts. The IFR contains flag bits for all the maskable interrupts.

When a maskable interrupt is requested, the flag bit in the corresponding control register is set to 1. If the mask bit in that same control register is also 1, the interrupt request is sent to the CPU, setting the corresponding flag in the IFR. This indicates that the interrupt is pending, or waiting for, acknowledgement.

You can read the IFR to identify pending interrupts and write to the IFR to clear pending interrupts. To clear a single interrupt, write a 1 to the corresponding IFR bit. All pending interrupts can be cleared by writing the current contents of the IFR back into the IFR. A device reset clears all IFR bits.

The following events also clear an IFR flag:

- ❑ The CPU acknowledges the interrupt.
- ❑ The ’C24x is reset.

Notes:

1) To clear an IFR bit, you must write a 1 to it, not a 0.

2) When a maskable interrupt is acknowledged, *only* the IFR bit is cleared automatically. The flag bit in the corresponding control register is *not* cleared. If an application requires that the control register flag be cleared, the bit must be cleared by software.

3) When an interrupt is requested by an INTR instruction and the corresponding IFR bit is set, the CPU does not clear the bit automatically. If an application requires that the IFR bit be cleared, the bit must be cleared by software.

The IFR is shown in Figure 5–5; descriptions of the bits follow the figure.

Figure 5–5. Interrupt Flag Register (IFR) — Address 0006h

15–6	5	4	3	2	1	0
Reserved	INT6	INT5	INT4	INT3	INT2	INT1
0	R/W–x	R/W1C–x	R/W1C–x	R/W1C–x	R/W1C–x	R/W1C–x

Note: 0 = Always read as zeros, R = Read access, W1C = Write 1 to this bit to clear it, –n = Value after reset, x = Value unchanged by reset

Bits 15–6 **Reserved**. These bits are always read as 0s.

Bit 5 **INT6.** Interrupt 6 flag. This bit is the flag for interrupts connected to interrupt level INT6.

0 = No INT6 interrupt is pending.
1 = At least one INT6 interrupt is pending. Write a 1 to this bit to clear it to 0 and clear the interrupt request.

Bit 4 **INT5.** Interrupt 5 flag. This bit is the flag for interrupts connected to interrupt level INT5.

0 = No INT5 interrupt is pending.
1 = At least one INT5 interrupt is pending. Write a 1 to this bit to clear it to 0 and clear the interrupt request.

Bit 3 **INT4.** Interrupt 4 flag. This bit is the flag for interrupts connected to interrupt level INT4.

0 = No INT4 interrupt is pending.
1 = At least one INT4 interrupt is pending. Write a 1 to this bit to clear it to 0 and clear the interrupt request.

Bit 2 **INT3.** Interrupt 3 flag. This bit is the flag for interrupts connected to interrupt level INT3.

0 = No INT3 interrupt is pending.
1 = At least one INT3 interrupt is pending. Write a 1 to this bit to clear it to 0 and clear the interrupt request.

Bit 1 **INT2.** Interrupt 2 flag. This bit is the flag for interrupts connected to interrupt level INT2.

0 = No INT2 interrupt is pending.
1 = At least one INT2 interrupt is pending. Write a 1 to this bit to clear it to 0 and clear the interrupt request.

Bit 0 **INT1.** Interrupt 1 flag. This bit is the flag for interrupts connected to interrupt level INT1.

0 = No INT1 interrupt is pending.
1 = At least one INT1 interrupt is pending. Write a 1 to this bit to clear it to 0 and clear the interrupt request.

5.7.2 Interrupt Mask Register (IMR)

The IMR is a 16-bit, memory-mapped register located at address 0004h in data memory space. The IMR contains mask bits for all the maskable interrupt levels (INT1–INT6). Neither NMI nor $\overline{\text{RS}}$ is included in the IMR; thus, IMR has no effect on these interrupts.

You can read the IMR to identify masked or unmasked interrupt levels, and you can write to the IMR to mask or unmask interrupt levels. To unmask an interrupt level, set its corresponding IMR bit to 1. To mask an interrupt level, set its corresponding IMR bit to 0. When an interrupt is masked, it is not acknowledged, regardless of the value of the INTM bit. When an interrupt is unmasked, it is acknowledged if the corresponding IFR bit is 1 and the INTM bit is 0. At reset, the IMR bits are all set to 0, masking all the maskable interrupts.

The IMR is shown in Figure 5–6, *Interrupt Mask Register (IMR)* on page 5-20. Bit descriptions follow the figure.

Figure 5–6. Interrupt Mask Register (IMR) — Address 0004h

15–6	5	4	3	2	1	0
Reserved	INT6	INT5	INT4	INT3	INT2	INT1
0	R/W–x	R/W–x	R/W–x	R/W–x	R/W–x	R/W–x

Note: 0 = Always read as zeros, R = Read access, W = Write access, –n = Value after reset, x = value unchanged by reset

Bits 15–6 **Reserved**. These bits are always read as 0s.

Bit 5 **INT6.** Interrupt 6 mask. This bit masks or unmasks interrupt level INT6.

0 = Level INT6 is masked.
1 = Level INT6 is unmasked.

Bit 4 **INT5.** Interrupt 5 mask. This bit masks or unmasks interrupt level INT5.

0 = Level INT5 is masked.
1 = Level INT5 is unmasked.

Bit 3 **INT4.** Interrupt 4 mask. This bit masks or unmasks interrupt level INT4.

0 = Level INT4 is masked.
1 = Level INT4 is unmasked.

Bit 2 **INT3.** Interrupt 3 mask. This bit masks or unmasks interrupt level INT3.

0 = Level INT3 is masked.
1 = Level INT3 is unmasked.

Bit 1 **INT2.** Interrupt 2 mask. This bit masks or unmasks interrupt level INT2.

0 = Level INT2 is masked.
1 = Level INT2 is unmasked.

Bit 0 **INT1.** Interrupt 1 mask. This bit masks or unmasks interrupt level INT1.

0 = Level INT1 is masked.
1 = Level INT1 is unmasked.

Chapter 6

Addressing Modes

This chapter explains the three basic memory addressing modes used by the ’C24x instruction set. The three modes are:

- Immediate addressing mode
- Direct addressing mode
- Indirect addressing mode

In the immediate addressing mode, a constant to be manipulated by the instruction is supplied directly as an operand of that instruction. The ’C24x supports two types of immediate addressing, long and short, described in section 6.1, *Immediate Addressing Mode*, on page 6-2.

When you need to access data memory, you can use the direct or indirect addressing mode. Direct addressing concatenates seven bits of the instruction word with the nine bits of the data-memory page pointer (DP) to form a 16-bit data memory address. Indirect addressing accesses data memory through one of eight 16-bit auxiliary registers.

Topic **Page**

6.1 Immediate Addressing Mode

In the immediate addressing mode, the instruction word contains a constant to be manipulated by the instruction. The two types of immediate addressing modes are:

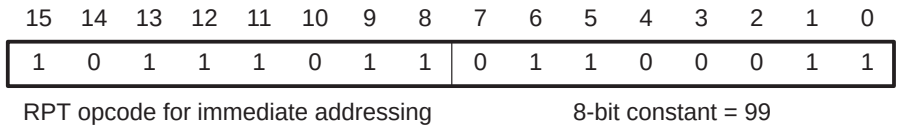
- **Short-immediate addressing.** Instructions that use short-immediate addressing have an 8-bit, 9-bit, or 13-bit constant as an operand. Short-immediate instructions require a single instruction word, with the constant embedded in that word.
- **Long-immediate addressing.** Instructions that use long-immediate addressing have a 16-bit constant as an operand and require two instruction words. The constant is sent as the second instruction word. This 16-bit value can be used as an absolute constant or as a 2s-complement value.

In Example 6–1, the immediate operand is contained as a part of the RPT instruction word. For this RPT instruction, the instruction register will be loaded with the value shown in Figure 6–1. Immediate operands are preceded by the symbol #.

Example 6–1. RPT Instruction Using Short-Immediate Addressing

```
RPT #99      ;Execute the instruction that follows RPT
             ;100 times.
```

Figure 6–1. Instruction Register Contents for Example 6–1

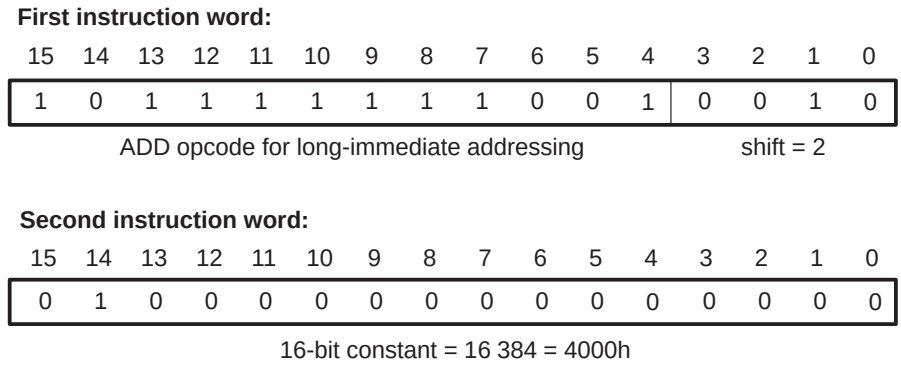


In Example 6–2, the immediate operand is contained in the second instruction word. The instruction register receives, consecutively, the two 16-bit values shown in Figure 6–2.

Example 6–2. ADD Instruction Using Long-Immediate Addressing

```
ADD   #16384,2  ;Shift the value 16384 left by two bits
                ;and add the result to the accumulator.
```

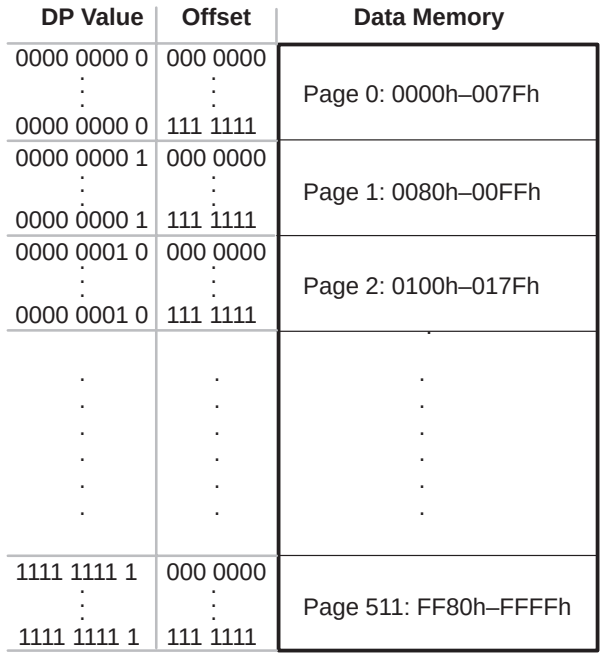
Figure 6–2. Two Words Loaded Consecutively to the Instruction Register in Example 6–2



6.2 Direct Addressing Mode

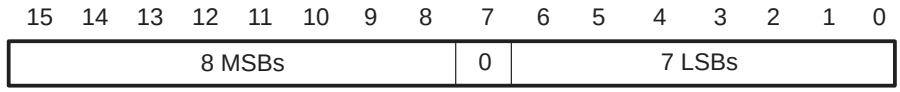
In the direct addressing mode, data memory is addressed in blocks of 128 words called data pages. The entire 64K of data memory consists of 512 data pages labeled 0 through 511, as shown in Figure 6–3. The current data page is determined by the value in the 9-bit data page pointer (DP) in status register ST0. For example, if the DP value is 0 0000 0000_2, the current data page is 0. If the DP value is 0 0000 0010_2, the current data page is 2.

Figure 6–3. Pages of Data Memory



In addition to the data page, the processor must know the particular word being referenced on that page. This is determined by a 7-bit offset (see Figure 6–3). The offset is supplied by the seven least significant bits (LSBs) of the IR register shown in Figure 6–4, *Instruction Register (IR) Contents in Direct Addressing Mode* instruction register, on page 6-5, which holds the opcode for the next instruction to be executed. In direct addressing mode, the contents of the instruction register has the format.

Figure 6–4. Instruction Register (IR) Contents in Direct Addressing Mode



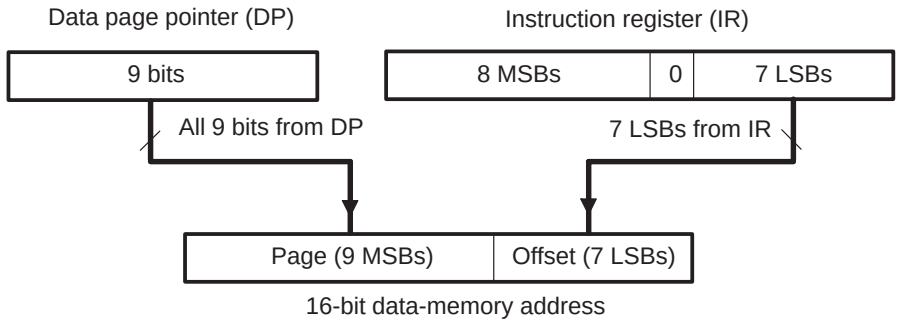
8 MSBs Bits 15 through 8 indicate the instruction type (for example, ADD) and also contain any information regarding a shift of the data value to be accessed by the instruction.

0 **Direct/indirect indicator.** Bit 7 contains a 0 to define the addressing mode as direct.

7 LSBs Bits 6 through 0 indicate the offset for the data-memory address referenced by the instruction.

To form a complete 16-bit address, the processor concatenates the DP value and the seven LSBs of the instruction register, as shown in Figure 6–5. The DP supplies the nine most significant bits (MSBs) of the address (the page number), and the seven LSBs of the instruction register supply the seven LSBs of the address (the offset). For example, to access data address 003Fh, you specify data page 0 (DP = 0000 0000 0) and an offset of 011 1111. Concatenating the DP and the offset produces the 16-bit address 0000 0000 0011 1111, which is 003Fh or decimal 63.

Figure 6–5. Generation of Data Addresses in Direct Addressing Mode



Initialize the DP in All Programs

It is critical that all programs initialize the DP. The DP is not initialized by reset and is undefined after power up. The ’C24x development tools use default values for many parameters, including the DP. However, programs that do not explicitly initialize the DP can execute improperly, depending on whether they are executed on a ’C24x device or with a development tool.

6.2.1 Using Direct Addressing Mode

When you use direct addressing mode, the processor uses the DP to find the data page and uses the seven LSBs of the instruction register to find a particular address on that page. Always do the following:

1) **Set the data page.** Load the appropriate value (from 0 to 511) into the DP. The DP register can be loaded by the LDP instruction or by any instruction that can load a value to ST0. The LDP instruction loads the DP directly without affecting the other bits of ST0, and it clearly indicates the value loaded into the DP. For example, to set the current data page to 32 (addresses 1000h–107Fh), you can use:

```
LDP #32   ;Initialize data page pointer
```

2) **Specify the offset.** Supply the 7-bit offset as an operand of the instruction. For example, if you want the ADD instruction to use the value at the second address of the current data page, you would write:

```
ADD 1h ;Add to accumulator the value in the current
   ;data page, offset of 1.
```

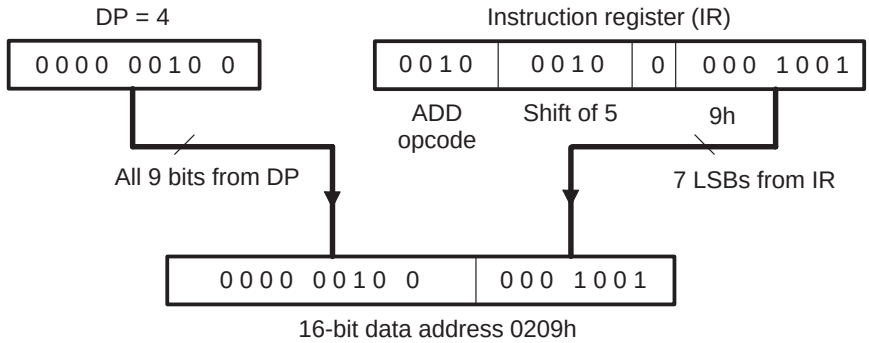
You do not have to set the data page prior to every instruction that uses direct addressing. If all the instructions in a block of code access the same data page, you can simply load the DP at the front of the block. However, if various data pages are being accessed throughout the block of code, be sure the DP is changed whenever a new data page should be accessed.

6.2.2 Examples of Direct Addressing

In Example 6–3, the first instruction loads the DP with 0 0000 0100_2 to set the current data page to 4. The ADD instruction then references a data memory address that is generated as shown following the program code. Before the ADD instruction is executed, the opcode is loaded into the instruction register. Together, the DP and the seven LSBs of the instruction register form the complete 16-bit address, 0000 0010 0000 1001_2 (0209h).

Example 6–3. Using Direct Addressing with ADD (Shift of 0 to 15)

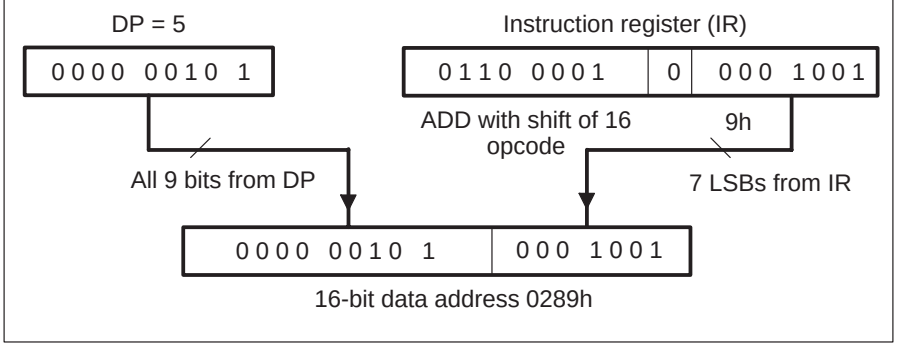
```
LDP #4   ;Set data page to 4 (addresses 0200h–027Fh).
ADD 9h,5 ;The contents of data address 0209h are
         ;left shifted 5 bits and added to the
         ;contents of the accumulator.
```



In Example 6–4, the ADD instruction references a data memory address that is generated as shown following the program code. For any instruction that performs a shift of 16, the shift value is not embedded directly in the instruction word; instead, all eight MSBs contain an opcode that not only indicates the instruction type, but also a shift of 16. The eight MSBs of the instruction word indicate an ADD with a shift of 16.

Example 6–4. Using Direct Addressing with ADD (Shift of 16)

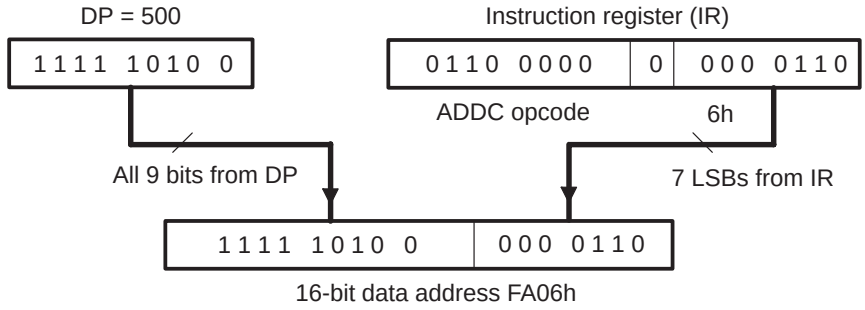
```
LDP #5     ;Set data page to 5 (addresses 0280h–02FFh).
ADD 9h,16  ;The contents of data address 0289h are
           ;left shifted 16 bits and added to the
           ;contents of the accumulator.
```



In Example 6–5, the ADDC instruction references a data memory address that is generated as shown following the program code. You should note that if an instruction does not perform shifts (such as the ADDC instruction), all eight MSBs of the instruction contain the opcode for the instruction type.

Example 6–5. Using Direct Addressing with ADDC

```
LDP #500  ;Set data page to 500 (addresses FA00h–FA7Fh).
ADDC 6h   ;The contents of data address FA06h
          ;and the value of the carry bit (C) are
          ;added to the contents of the accumulator.
```



6.3 Indirect Addressing Mode

Eight auxiliary registers (AR0–AR7) provide flexible and powerful indirect addressing. Any location in the 64K data memory space can be accessed using a 16-bit address contained in an auxiliary register.

6.3.1 Current Auxiliary Register

To select a specific auxiliary register, load the 3-bit auxiliary register pointer (ARP) of status register ST0 with a value from 0 to 7. The ARP can be loaded as a primary operation by the MAR instruction or by the LST instruction. The ARP can be loaded as a secondary operation by any instruction that supports indirect addressing.

The register pointed to by the ARP is referred to as the *current auxiliary register* or *current AR*. During the processing of an instruction, the content of the current auxiliary register is used as the address at which the data-memory access occurs. The ARAU passes this address to the data-read address bus (DRAB) if the instruction requires a read from data memory, or it passes the address to the data-write address bus (DWAB) if the instruction requires a write to data memory. After the instruction uses the data value, the contents of the current auxiliary register can be incremented or decremented by the ARAU, which implements unsigned 16-bit arithmetic.

Normally, the ARAU performs its arithmetic operations in the decode phase of the pipeline (when the instruction specifying the operations is being decoded). This allows the address to be generated before the decode phase of the next instruction. There is an exception to this rule: during processing of the NORM instruction, the auxiliary register and/or ARP modification is done during the execute phase of the pipeline. For information on the pipeline operation, see Chapter 5, Section 5.2, *Pipeline Operation*, on page 5-7.

6.3.2 Indirect Addressing Options

The ’C24x provides four types of indirect addressing options:

- **No increment or decrement.** The instruction uses the content of the current auxiliary register as the data memory address but neither increments nor decrements the content of the current auxiliary register.
- **Increment or decrement by 1.** The instruction uses the content of the current auxiliary register as the data memory address and then increments or decrements the content of the current auxiliary register by one.
- **Increment or decrement by an index amount.** The value in AR0 is the index amount. The instruction uses the content of the current auxiliary register as the data memory address and then increments or decrements the content of the current auxiliary register by the index amount.

- **Increment or decrement by an index amount using reverse carry.** The value in AR0 is the index amount. After the instruction uses the content of the current auxiliary register as the data-memory address, that content is incremented or decremented by the index amount. The addition and subtraction process is accomplished with the carry propagation reversed for fast Fourier transforms (FFTs).

These four option types provide the seven indirect addressing options listed in Table 6–1. The table also shows the instruction operand that corresponds to each indirect addressing option and gives an example of how each option is used.

Table 6–1. Indirect Addressing Operands

Operand	Option	Example
*	No increment or decrement	**LT *** loads the temporary register (TREG) with the content of the data memory address referenced by the current AR.
*+	Increment by 1	**LT *+** loads the temporary register (TREG) with the content of the data memory address referenced by the current AR and then adds 1 to the content of the current AR.
*–	Decrement by 1	**LT *–** loads the temporary register (TREG) with the content of the data memory address referenced by the current AR and then subtracts 1 from the content of the current AR.
*0+	Increment by index amount	**LT *0+** loads the temporary register (TREG) with the content of the data memory address referenced by the current AR and then adds the content of AR0 to the content of the current AR.
*0–	Decrement by index amount	**LT *0–** loads the temporary register (TREG) with the content of the data memory address referenced by the current AR and then subtracts the content of AR0 from the content of the current AR.
*BR0+	Increment by index amount, adding with reverse carry	**LT *BR0+** loads the temporary register (TREG) with the content of the data memory address referenced by the current AR and then adds the content of AR0 to the content of the current AR, adding with reverse carry propagation.
*BR0–	Decrement by index amount, subtracting with reverse carry	**LT *BR0–** loads the temporary register (TREG) with the content of the data memory address referenced by the current AR and then subtracts the content of AR0 from the content of the current AR, subtracting with bit reverse carry propagation.

All increments or decrements are performed by the auxiliary register arithmetic unit (ARAU) in the same cycle during which the instruction is being decoded in the pipeline.

The bit-reversed indexed addressing allows efficient I/O operations by resequencing the data points in a radix-2 FFT program. The direction of carry propagation in the ARAU is reversed when the address is selected, and AR0 is added to or subtracted from the current auxiliary register. A typical use of this addressing mode requires that AR0 be set initially to a value corresponding to half of the array's size, and further, that the current AR value be set to the base address of the data (the first data point).

6.3.3 Next Auxiliary Register

In addition to updating the current auxiliary register, a number of instructions can also specify the *next auxiliary register* or *next AR*. This register will be the current auxiliary register when the instruction execution is complete. The instructions that allow you to specify the next auxiliary register load the ARP with a new value. When the ARP is loaded with that value, the previous ARP value is loaded into the auxiliary register pointer buffer (ARB).

Example 6–6 illustrates the selection of a next auxiliary register and other indirect addressing features.

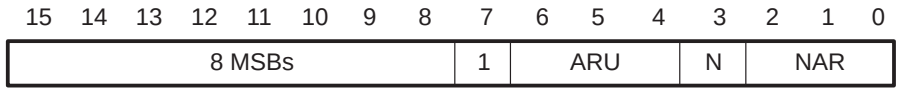
Example 6–6. Selecting a New Current Auxiliary Register

```
MAR*,AR1     ;Load the ARP with 1 to make AR1 the
             ;current auxiliary register.
LT  *+,AR2   ;AR2 is the next auxiliary register.
             ;Load the TREG with the content of the
             ;address referenced by AR1, add one to
             ;the content of AR1, then make AR2 the
             ;current auxiliary register.
MPY*         ;Multiply TREG by content of address
             ;referenced by AR2.
```

6.3.4 Indirect Addressing Opcode Format

Figure 6–6 shows the format of the instruction word loaded into the instruction register when you use indirect addressing. The opcode fields are described following Figure 6–6.

Figure 6–6. Instruction Register Content in Indirect Addressing



8 MSBs Bits 15 through 8 indicate the instruction type (for example, LT) and also contain any information regarding data shifts.

1 **Direct/indirect indicator.** Bit 7 contains a 1 to define the addressing mode as indirect.

ARU **Auxiliary register update code.** Bits 6 through 4 determine whether and how the current auxiliary register is incremented or decremented. See Table 6–2 below.

N **Next auxiliary register indicator.** Bit 3 specifies whether the instruction changes the ARP value.

- N = 0 The content of the ARP remains unchanged.
- N = 1 The content of NAR is loaded into the ARP, and the old ARP value is loaded into the auxiliary register buffer (ARB) of status register ST1.

NAR **Next auxiliary register value.** Bits 2 through 0 contain the value of the next auxiliary register. NAR is loaded into the ARP if N = 1.

Table 6–2. Effects of the ARU Code on the Current Auxiliary Register

ARU Code			
6	**5**	**4**	**Arithmetic Operation Performed on Current AR**
0	0	0	No operation on current AR
0	0	1	Current AR – 1 → current AR
0	1	0	Current AR + 1 → current AR
0	1	1	Reserved
1	0	0	Current AR – AR0 → current AR [reverse carry propagation]
1	0	1	Current AR – AR0 → current AR
1	1	0	Current AR + AR0 → current AR
1	1	1	Current AR + AR0 → current AR [reverse carry propagation]

Table 6–3 shows the opcode field bits and the notation used for indirect addressing. It also shows the corresponding operations performed on the current auxiliary register and the ARP.

Table 6–3. Field Bits and Notation for Indirect Addressing

Instruction Opcode Bits												
15	**–**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**	**Operand(s)**	**Operation**
←	8 MSBs	→	1	0	0	0	0	←NAR→			*	No manipulation of current AR
←	8 MSBs	→	1	0	0	0	1	←NAR→			*,AR*n*	NAR → ARP
←	8 MSBs	→	1	0	0	1	0	←NAR→			***–**	Current AR – 1 → current AR
←	8 MSBs	→	1	0	0	1	1	←NAR→			*–,AR*n*	Current AR – 1 → current AR NAR → ARP
←	8 MSBs	→	1	0	1	0	0	←NAR→			*+	Current AR + 1 → current AR
←	8 MSBs	→	1	0	1	0	1	←NAR→			*+,AR*n*	Current AR + 1 → current AR NAR → ARP
←	8 MSBs	→	1	1	0	0	0	←NAR→			*BR0–	Current AR – *rc*AR0 → current AR †
←	8 MSBs	→	1	1	0	0	1	←NAR→			*BR0–,AR*n*	Current AR – *rc*AR0 → current AR NAR → ARP †
←	8 MSBs	→	1	1	0	1	0	←NAR→			*0–	Current AR – AR0 → current AR
←	8 MSBs	→	1	1	0	1	1	←NAR→			*0–,AR*n*	Current AR – AR0 → current AR NAR → ARP
←	8 MSBs	→	1	1	1	0	0	←NAR→			*0+	Current AR + AR0 → current AR
←	8 MSBs	→	1	1	1	0	1	←NAR→			*0+,AR*n*	Current AR + AR0 → current AR NAR → ARP
←	8 MSBs	→	1	1	1	1	0	←NAR→			*BR0+	Current AR + *rc*AR0 → current AR †
←	8 MSBs	→	1	1	1	1	1	←NAR→			*BR0+,AR*n*	Current AR + *rc*AR0 → current AR NAR → ARP †

† Bit-reversed addressing mode

Legend:

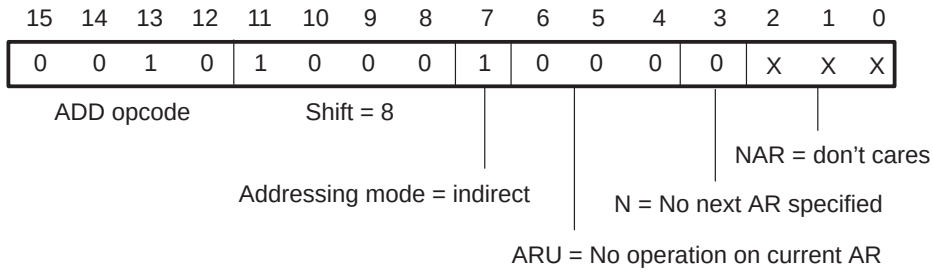
rc	Reverse carry propagation
NAR	Next AR
n	0, 1, 2, ..., or 7
8 MSBs	Eight bits determined by instruction type and (sometimes) shift information
→	Is loaded into

6.3.5 Examples of Indirect Addressing

Example 6–7 illustrates how the instruction register is loaded with the value shown when the ADD instruction is fetched from program memory.

Example 6–7. Indirect Addressing—No Increment or Decrement

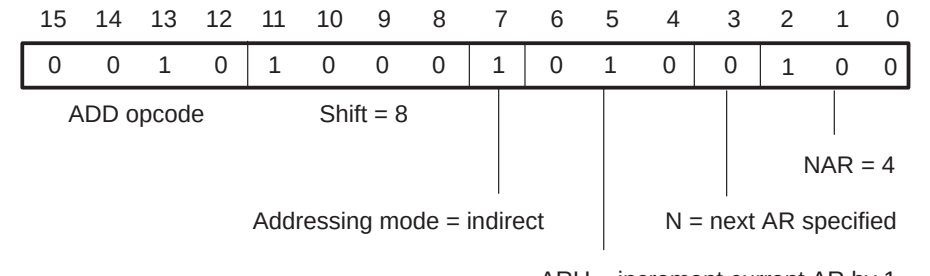
```
ADD *,8      ;Add to the accumulator the content of the
             ;data-memory address referenced by the
             ;current auxiliary register. The data
             ;is left shifted 8 bits before being added.
```



Example 6–8, illustrates how the instruction register is loaded with the value shown when the ADD instruction is fetched from program memory.

Example 6–8. Indirect Addressing—Increment by 1

```
ADD *+,8,AR4    ;Operates as in Example 6–7, but
                ;in addition, the current auxiliary
                ;register is incremented by one, and
                ;AR4 is chosen as the next auxiliary
                ;register.
```



Example 6–9. Indirect Addressing—Decrement by 1

```
ADD *–,8     ;Operates as in Example 6–7, but in
             ;addition, the current auxiliary register
             ;is decremented by one.
```

Example 6–10. Indirect Addressing—Increment by Index Amount

```
ADD *0+,8   ;Operates as in Example 6–7, but in
            ;addition, the content of register AR0
            ;is added to the current auxiliary
            ;register.
```

Example 6–11. Indirect Addressing—Decrement by Index Amount

```
ADD *0–,8    ;Operates as in Example 6–7, but in
             ;addition, the content of register AR0
             ;is subtracted from the current auxiliary
             ;register.
```

Example 6–12. Indirect Addressing—Increment by Index Amount With Reverse Carry Propagation

```
ADD *BR0+,8 ;Operates as in Example 6–10, except that
            ;the content of register AR0 is added to
            ;the current auxiliary register with
            ;reverse carry propagation.
```

Example 6–13. Indirect Addressing—Decrement by Index Amount With Reverse Carry Propagation

```
ADD *BR0–,8 ;Operates as in Example 6–11, except that
            ;the content of register AR0 is subtracted
            ;from the current auxiliary register with
            ;reverse carry propagation.
```

6.3.6 Modifying Auxiliary Register Content

The LAR, ADRK, SBRK, and MAR instructions are specialized instructions for changing the content of an auxiliary register (AR):

- The LAR instruction loads an AR.
- The ADRK instruction adds an immediate value to an AR; SBRK subtracts an immediate value.
- The MAR instruction can increment or decrement an AR value by 1 or by an index amount.

When modifying auxiliary register content, you are not limited to these four instructions. Auxiliary registers can be modified by any instruction that supports indirect addressing operands. (Indirect addressing can be used with all instructions except those that have immediate operands or no operands.)

Chapter 7

Assembly Language Instructions

Note:

The instruction set for the TMS320C24x is identical to that of the TMS320C2xx core.

This chapter describes the ’C24x assembly language instructions. This instruction set supports numerically intensive signal-processing operations as well as general-purpose applications, such as multiprocessing and high-speed control. The ’C24x instruction set is compatible with the ’C2x instruction set; code written for the ’C2x can be reassembled to run on the ’C24x. The ’C5x instruction set is a superset of that of the ’C24x; thus, code written for the ’C24x can be upgraded to run on a ’C5x.

Topic **Page**

7.1 Instruction Set Summary

This section provides six tables (Table 7–1 to Table 7–6) that summarize the instruction set according to the following functional headings:

- ❑ Accumulator, arithmetic, and logic instructions (see Table 7–1 on page 7-5)
- ❑ Auxiliary register and data page pointer instructions (see Table 7–2 on page 7-7)
- ❑ TREG, PREG, and multiply instructions (see Table 7–3 on page 7-8)
- ❑ Branch instructions (see Table 7–4 on page 7-9)
- ❑ Control instructions (see Table 7–5 on page 7-10)
- ❑ I/O and memory operations (see Table 7–6 on page 7-11)

Within each table, the instructions are arranged alphabetically. The number of words that an instruction occupies in program memory is specified in column three of each table; the number of cycles that an instruction requires to execute is in column four. 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 about each instruction is presented in the individual instruction descriptions in Section 7.2 on page 7-12.

For your reference, here are the definitions of the symbols used in the six summary tables:

ACC The accumulator

AR The auxiliary register

ARX A 3-bit value used in the LAR and SAR instructions to designate which auxiliary register will be loaded (LAR) or have its contents stored (SAR)

BITX A 4-bit value (called the bit code) that determines which bit of a designated data memory value will be tested by the BIT instruction

CM A 2-bit value. The CMPR instruction performs a comparison specified by the value of CM:

If CM = 00, test whether current AR = AR0
If CM = 01, test whether current AR < AR0
If CM = 10, test whether current AR > AR0
If CM = 11, test whether current AR ≠ AR0

IAAA AAAA (One I followed by seven As) The I at the left represents a bit that reflects whether direct addressing (I = 0) or indirect addressing (I = 1) is being used. When direct addressing is used, the seven As are the seven least significant bits (LSBs) of a data memory address. For indirect addressing, the seven As are bits that control auxiliary register manipulation (see Section 6.3, *Indirect Addressing Mode*, on page 6-9).

IIII IIII (Eight Is) An 8-bit constant used in short immediate addressing

I IIII IIII (Nine Is) A 9-bit constant used in short immediate addressing for the LDP instruction

I IIII IIII IIII (Thirteen Is) A 13-bit constant used in short immediate addressing for the MPY instruction

I NTR# A 5-bit value representing a number from 0 to 31. The INTR instruction uses this number to change program control to one of the 32 interrupt vector addresses.

PM A 2-bit value copied into the PM bits of status register ST1 by the SPM instruction

SHF A 3-bit left-shift value

SHFT A 4-bit left-shift value

TP A 2-bit value used by the conditional execution instructions to represent four conditions:

$\overline{\text{BIO}}$ pin low	TP = 00
TC bit =1	TP = 01
TC bit = 0	TP = 10
No condition	TP = 11

ZLVC ZLVC Two 4-bit fields — each representing the following conditions:

ACC = 0	Z
ACC < 0	L
Overflow	V
Carry	C

A conditional instruction contains two of these 4-bit fields. The 4-LSB field of the instruction is a mask field. A 1 in the corresponding mask bit indicates that condition is being tested. For example, to test for ACC ≥ 0, the Z and L fields are set, and the V and C fields are not set. The Z field is set to test the condition ACC = 0, and the L field is reset to test the condition ACC ≥ 0.The second 4-bit field (bits 4 – 7) indicates the state of the conditions to test. The conditions possible with these eight bits are shown in the descriptions for the BCND, CC, and RETC instructions.

+ 1 word The second word of a 2-word opcode. This second word contains a 16-bit constant. Depending on the instruction, this constant is a long immediate value, a program memory address, or an address for an I/O port or an I/O-mapped register.

Table 7–1. Accumulator, Arithmetic, and Logic Instructions

Mnemonic	Description	Words	Cycles	Opcode
ABS	Absolute value of ACC	1	1	1011 1110 0000 0000
ADD	Add to ACC with shift of 0 to 15, direct or indirect	1	1	0010 SHFT IAAA AAAA
	Add to ACC with shift 0 to 15, long immediate	2	2	1011 1111 1001 SHFT + 1 word
	Add to ACC with shift of 16, direct or indirect	1	1	0110 0001 IAAA AAAA
	Add to ACC, short immediate	1	1	1011 1000 IIII IIII
ADDC	Add to ACC with carry, direct or indirect	1	1	0110 0000 IAAA AAAA
ADDS	Add to low ACC with sign-extension suppressed, direct or indirect	1	1	0110 0010 IAAA AAAA
ADDT	Add to ACC with shift (0 to 15) specified by TREG, direct or indirect	1	1	0110 0011 IAAA AAAA
AND	AND ACC with data value, direct or indirect	1	1	0110 1110 IAAA AAAA
	AND with ACC with shift of 0 to 15, long immediate	2	2	1011 1111 1011 SHFT + 1 word
	AND with ACC with shift of 16, long immediate	2	2	1011 1110 1000 0001 + 1 word
CMPL	Complement ACC	1	1	1011 1110 0000 0001
LACC	Load ACC with shift of 0 to 15, direct or indirect	1	1	0001 SHFT IAAA AAAA
	Load ACC with shift of 0 to 15, long immediate	2	2	1011 1111 1000 SHFT + 1 word
	Load ACC with shift of 16, direct or indirect	1	1	0110 1010 IAAA AAAA
LACL	Load low word of ACC, direct or indirect	1	1	0110 1001 IAAA AAAA
	Load low word of ACC, short immediate	1	1	1011 1001 IIII IIII
LACT	Load ACC with shift (0 to 15) specified by TREG, direct or indirect	1	1	0110 1011 IAAA AAAA
NEG	Negate ACC	1	1	1011 1110 0000 0010
NORM	Normalize the contents of ACC, indirect	1	1	1010 0000 IAAA AAAA

Table 7–1. Accumulator, Arithmetic, and Logic Instructions (Continued)

Mnemonic	Description	Words	Cycles	Opcode
OR	OR ACC with data value, direct or indirect	1	1	0110 1101 IAAA AAAA
	OR with ACC with shift of 0 to 15, long immediate	2	2	1011 1111 1100 SHFT + 1 word
	OR with ACC with shift of 16, long immediate	2	2	1011 1110 1000 0010 + 1 word
ROL	Rotate ACC left	1	1	1011 1110 0000 1100
ROR	Rotate ACC right	1	1	1011 1110 0000 1101
SACH	Store high ACC with shift of 0 to 7, direct or indirect	1	1	1001 1SHF IAAA AAAA
SACL	Store low ACC with shift of 0 to 7, direct or indirect	1	1	1001 0SHF IAAA AAAA
SFL	Shift ACC left	1	1	1011 1110 0000 1001
SFR	Shift ACC right	1	1	1011 1110 0000 1010
SUB	Subtract from ACC with shift of 0 to 15, direct or indirect	1	1	0011 SHFT IAAA AAAA
	Subtract from ACC with shift of 0 to 15, long immediate	2	2	1011 1111 1010 SHFT + 1 word
	Subtract from ACC with shift of 16, direct or indirect	1	1	0110 0101 IAAA AAAA
	Subtract from ACC, short immediate	1	1	1011 1010 IIII IIII
SUBB	Subtract from ACC with borrow, direct or indirect	1	1	0110 0100 IAAA AAAA
SUBC	Conditional subtract, direct or indirect	1	1	0000 1010 IAAA AAAA
SUBS	Subtract from ACC with sign-extension suppressed, direct or indirect	1	1	0110 0110 IAAA AAAA
SUBT	Subtract from ACC with shift (0 to 15) specified by TREG, direct or indirect	1	1	0110 0111 IAAA AAAA

Table 7–1. Accumulator, Arithmetic, and Logic Instructions (Continued)

Mnemonic	Description	Words	Cycles	Opcode
XOR	Exclusive OR ACC with data value, direct or indirect	1	1	0110 1100 IAAA AAAA
	Exclusive OR with ACC with shift of 0 to 15, long immediate	2	2	1011 1111 1101 SHFT + 1 word
	Exclusive OR with ACC with shift of 16, long immediate	2	2	1011 1110 1000 0011 + 1 word
ZALR	Zero low ACC and load high ACC with rounding, direct or indirect	1	1	0110 1000 IAAA AAAA

Table 7–2. Auxiliary Register Instructions

Mnemonic	Description	Words	Cycles	Opcode
ADRK	Add constant to current AR, short immediate	1	1	0111 1000 IIII IIII
BANZ	Branch on current AR not 0, indirect	2	4 (condition true) 2 (condition false)	0111 1011 1AAA AAAA + 1 word
CMPR	Compare current AR with AR0	1	1	1011 1111 0100 01CM
LAR	Load specified AR from specified data location, direct or indirect	1	2	0000 0ARX IAAA AAAA
	Load specified AR with constant, short immediate	1	2	1011 0ARX IIII IIII
	Load specified AR with constant, long immediate	2	2	1011 1111 0000 1ARX + 1 word
MAR	Modify current AR and/or ARP, indirect (performs no operation when direct)	1	1	1000 1011 IAAA AAAA
SAR	Store specified AR to specified data location, direct or indirect	1	1	1000 0ARX IAAA AAAA
SBRK	Subtract constant from current AR, short immediate	1	1	0111 1100 IIII IIII

Table 7–3. TREG, PREG, and Multiply Instructions

Mnemonic	Description	Words	Cycles	Opcode
APAC	Add PREG to ACC	1	1	1011 1110 0000 0100
LPH	Load high PREG, direct or indirect	1	1	0111 0101 IAAA AAAA
LT	Load TREG, direct or indirect	1	1	0111 0011 IAAA AAAA
LTA	Load TREG and accumulate previous product, direct or indirect	1	1	0111 0000 IAAA AAAA
LTD	Load TREG, accumulate previous product, and move data, direct or indirect	1	1	0111 0010 IAAA AAAA
LTP	Load TREG and store PREG in accumulator, direct or indirect	1	1	0111 0001 IAAA AAAA
LTS	Load TREG and subtract previous product, direct or indirect	1	1	0111 0100 IAAA AAAA
MAC	Multiply and accumulate, direct or indirect	2	3	1010 0010 IAAA AAAA + 1 word
MACD	Multiply and accumulate with data move, direct or indirect	2	3	1010 0011 IAAA AAAA + 1 word
MPY	Multiply TREG by data value, direct or indirect	1	1	0101 0100 IAAA AAAA
	Multiply TREG by 13-bit constant, short immediate	1	1	110I IIII IIII IIII
MPYA	Multiply and accumulate previous product, direct or indirect	1	1	0101 0000 IAAA AAAA
MPYS	Multiply and subtract previous product, direct or indirect	1	1	0101 0001 IAAA AAAA
MPYU	Multiply unsigned, direct or indirect	1	1	0101 0101 IAAA AAAA
PAC	Load ACC with PREG	1	1	1011 1110 0000 0011
SPAC	Subtract PREG from ACC	1	1	1011 1110 0000 0101
SPH	Store high PREG, direct or indirect	1	1	1000 1101 IAAA AAAA
SPL	Store low PREG, direct or indirect	1	1	1000 1100 IAAA AAAA
SPM	Set product shift mode	1	1	1011 1111 0000 00PM
SQRA	Square and accumulate previous product, direct or indirect	1	1	0101 0010 IAAA AAAA
SQRS	Square and subtract previous product, direct or indirect	1	1	0101 0011 IAAA AAAA

Table 7–4. Branch Instructions

Mnemonic	Description	Words	Cycles	Opcode
B	Branch unconditionally, indirect	2	4	0111 1001 1AAA AAAA + 1 word
BACC	Branch to address specified by ACC	1	4	1011 1110 0010 0000
BANZ	Branch on current AR not 0, indirect	2	4 (condition true) 2 (condition false)	0111 1011 1AAA AAAA + 1 word
BCND	Branch conditionally	2	4 (conditions true) 2 (any condition false)	1110 00TP ZLVC ZLVC + 1 word
CALA	Call subroutine at location specified by ACC	1	4	1011 1110 0011 0000
CALL	Call subroutine, indirect	2	4	0111 1010 1AAA AAAA + 1 word
CC	Call conditionally	2	4 (conditions true) 2 (any condition false)	1110 10TP ZLVC ZLVC + 1 word
INTR	Soft interrupt	1	4	1011 1110 011I NTR#
NMI	Nonmaskable interrupt	1	4	1011 1110 0101 0010
RET	Return from subroutine	1	4	1110 1111 0000 0000
RETC	Return conditionally	1	4 (conditions true) 2 (any condition false)	1110 11TP ZLVC ZLVC
TRAP	Software interrupt	1	4	1011 1110 0101 0001

Table 7–5. Control Instructions

Mnemonic	Description	Words	Cycles	Opcode
BIT	Test bit, direct or indirect	1	1	0100 BITX IAAA AAAA
BITT	Test bit specified by TREG, direct or indirect	1	1	0110 1111 IAAA AAAA
CLRC	Clear C bit	1	1	1011 1110 0100 1110
	Clear CNF bit	1	1	1011 1110 0100 0100
	Clear INTM bit	1	1	1011 1110 0100 0000
	Clear OVM bit	1	1	1011 1110 0100 0010
	Clear SXM bit	1	1	1011 1110 0100 0110
	Clear TC bit	1	1	1011 1110 0100 1010
	Clear XF bit	1	1	1011 1110 0100 1100
IDLE	Idle until interrupt	1	1	1011 1110 0010 0010
LDP	Load data page pointer, direct or indirect	1	2	0000 1101 IAAA AAAA
	Load data page pointer, short immediate	1	2	1011 110I IIII IIII
LST	Load status register ST0, direct or indirect	1	2	0000 1110 IAAA AAAA
	Load status register ST1, direct or indirect	1	2	0000 1111 IAAA AAAA
NOP	No operation	1	1	1000 1011 0000 0000
POP	Pop top of stack to low ACC	1	1	1011 1110 0011 0010
POPD	Pop top of stack to data memory, direct or indirect	1	1	1000 1010 IAAA AAAA
PSHD	Push data memory value on stack, direct or indirect	1	1	0111 0110 IAAA AAAA
PUSH	Push low ACC onto stack	1	1	1011 1110 0011 1100
RPT	Repeat next instruction, direct or indirect	1	1	0000 1011 IAAA AAAA
	Repeat next instruction, short immediate	1	1	1011 1011 IIII IIII
SETC	Set C bit	1	1	1011 1110 0100 1111
	Set CNF bit	1	1	1011 1110 0100 0101
	Set INTM bit	1	1	1011 1110 0100 0001
	Set OVM bit	1	1	1011 1110 0100 0011
	Set SXM bit	1	1	1011 1110 0100 0111
	Set TC bit	1	1	1011 1110 0100 1011
	Set XF bit	1	1	1011 1110 0100 1101
SPM	Set product shift mode	1	1	1011 1111 0000 00PM
SST	Store status register ST0, direct or indirect	1	1	1000 1110 IAAA AAAA
	Store status register ST1, direct or indirect	1	1	1000 1111 IAAA AAAA

Table 7–6. I/O and Memory Instructions

Mnemonic	Description	Words	Cycles	Opcode
BLDD	Block move from data memory to data memory, direct/indirect with long immediate source	2	3	1010 1000 IAAA AAAA + 1 word
	Block move from data memory to data memory, direct/indirect with long immediate destination	2	3	1010 1001 IAAA AAAA + 1 word
BLPD	Block move from program memory to data memory, direct/indirect with long immediate source	2	3	1010 0101 IAAA AAAA + 1 word
DMOV	Data move in data memory, direct or indirect	1	1	0111 0111 IAAA AAAA
IN	Input data from I/O location, direct or indirect	2	2	1010 1111 IAAA AAAA + 1 word
OUT	Output data to port, direct or indirect	2	3	0000 1100 IAAA AAAA + 1 word
SPLK	Store long immediate to data memory location, direct or indirect	2	2	1010 1110 IAAA AAAA + 1 word
TBLR	Table read, direct or indirect	1	3	1010 0110 IAAA AAAA
TBLW	Table write, direct or indirect	1	3	1010 0111 IAAA AAAA

7.2 How To Use the Instruction Descriptions

Section 7.3 contains detailed information on the instruction set. The description for each instruction presents the following categories of information:

- ❏ Syntax
- ❏ Operands
- ❏ Opcode
- ❏ Execution
- ❏ Status Bits
- ❏ Description
- ❏ Words
- ❏ Cycles
- ❏ Examples

7.2.1 Syntax

Each instruction begins with a list of the available assembler syntax expressions and the addressing mode type(s) for each expression. For example, the description for the ADD instruction begins with:

ADD *dma* [, *shift*]	Direct addressing
ADD *dma*, **16**	Direct with left shift of 16
ADD *ind* [, *shift* [, **AR***n*]]	Indirect addressing
ADD *ind*, **16** [, **AR***n*]	Indirect with left shift of 16
ADD #*k*	Short immediate addressing
ADD #*lk* [, *shift*]	Long immediate addressing

These are the notations used in the syntax expressions:

italic symbols

Italic symbols in an instruction syntax represent variables.

Example: For the syntax
ADD *dma*
you may use a variety of values for *dma*. Samples with this syntax follow:

```
ADD DAT
ADD 15
```

boldface characters

Boldface characters in an instruction syntax must be typed as shown.

Example: For the syntax
ADD *dma*, **16**
you may use a variety of values for *dma*, but the word ADD and the number 16 must be typed as shown. Samples with this syntax follow:

```
ADD 7h, 16
ADD X, 16
```

[, x] Operand x is optional.

Example: For the syntax
ADD *dma*, [, *shift*]
you must supply *dma*, as in the instruction:
`ADD 7h`
and you have the option of adding a *shift* value, as in the instruction:
`ADD 7h, 5`

[, x1 [, x2]] Operands x1 and x2 are optional, but you cannot include x2 without also including x1.

Example: For the syntax
ADD *ind*, [, *shift* [, **AR***n*]]
you must supply *ind*, as in the instruction:
`ADD *+`
You have the option of including *shift*, as in the instruction:
`ADD *+, 5`
If you wish to include **AR***n*, you must also include *shift*, as in:
`ADD *+, 0, AR2`

The # symbol is a prefix for constants used in immediate addressing. For short- or long- immediate operands, it is used in instructions where there is ambiguity with other addressing modes.

Example: `RPT #15` uses short immediate addressing. It causes the next instruction to be repeated 16 times. But `RPT 15` uses direct addressing. The number of times the next instruction repeats is determined by a value stored in memory.

Finally, consider this code example:

```
MoveData BLDD  DAT5, #310h  ;move data at address
                            ;referenced by DAT5 to address
                            ;310h.
```

Note the optional MoveData label is used as a reference in front of the instruction mnemonic. Place labels either before the instruction mnemonic on the same line or on the preceding line in the first column. (Be sure there are no spaces in your labels.) An optional comment field can conclude the syntax expression. At least one space is required between fields (label, mnemonic, operand, and comment).

7.2.2 Operands

Operands can be constants, or assembly-time expressions referring to memory, I/O ports, register addresses, pointers, shift counts, and a variety of other constants. The operands category for each instruction description defines the variables used for and/or within operands in the syntax expressions. For example, for the ADD instruction, the syntax category gives these syntax expressions:

ADD *dma* [, *shift*]	Direct addressing
ADD *dma*, **16**	Direct with left shift of 16
ADD *ind* [, *shift* [, **AR***n*]]	Indirect addressing
ADD *ind*, **16** [, **AR***n*]	Indirect with left shift of 16
ADD #*k*	Short immediate addressing
ADD #*lk* [, *shift*]	Long immediate addressing

The operands category defines the variables *dma*, *shift*, *ind*, *n*, *k*, and *lk*. For *ind*, an indirect addressing variable, you supply one of the following seven symbols:

* *+ *– *0+ *0– *BR0+ *BR0–

These symbols are defined in subsection 6.3.2, *Indirect Addressing Options*, on page 6-9.

7.2.3 Opcode

The opcode category breaks down the various bit fields that make up each instruction word. When one of the fields contains a constant value derived directly from an operand, it has the same name as that operand. The contents of fields that do not directly relate to operands have other names; the opcode category either explains these names directly or refers you to a section of this book that explains them in detail. For example, these opcodes are given for the ADDC instruction:

ADDC *dma*

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

ADDC *ind* [, **AR***n*]

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

Note: ARU, N, and NAR are defined in Section 6.3, *Indirect Addressing Mode* (page 6-9).

The field called dma contains the value *dma*, which is defined in the operands category. The contents of the fields ARU, N, and NAR are derived from the operands *ind* and *n* but do not directly correspond to those operands; therefore, a note directs you to the appropriate section for more details.

7.2.4 Execution

The execution category presents an instruction operation sequence that describes the processing that takes place when the instruction is executed. If the execution event or events depend on the addressing mode used, the execution category specifies which events are associated with which addressing modes. Here are notations used in the execution category:

(r)	The content of register or location r. *Example:* (ACC) represents the value in the accumulator.
x → y	Value x is assigned to register or location y. *Example:* (data-memory address) → ACC means: The content of the specified data-memory address is put into the accumulator.
r(n:m)	Bits n through m of register or location r. *Example:* ACC(15:0) represents bits 15 through 0 of the accumulator.
(r(n:m))	The content of bits n through m of register or location r. *Example:* (ACC(31:16)) represents the content of bits 31 through 16 of the accumulator.
nnh	Indicates that nn represents a hexadecimal number.

7.2.5 Status Bits

The bits in status registers ST0 and ST1 affect the operation of certain instructions and are affected by certain instructions. The status bits category of each instruction description states which of the bits (if any) affect the execution of the instruction and which of the bits (if any) are affected by the instruction.

7.2.6 Description

The description category explains what happens during instruction execution and its effect on the rest of the processor or on memory contents. It also discusses any constraints on the operands imposed by the processor or the assembler. This description parallels and supplements the information given in the execution category.

7.2.7 Words

The words category specifies the number of memory words required to store the instruction (one or two). When the number of words depends on the addressing mode used for an instruction, the words category specifies which addressing modes require one word and which require two words.

7.2.8 Cycles

The cycles category of each instruction description contains tables showing the number of processor machine cycles (CLKOUT1 periods) required for the instruction to execute in a given memory configuration when executed as a single instruction or when repeated with the RPT instruction. For example:

Cycles for a Single Instruction

	Program			
Operand	**ROM**	**DARAM**	**SARAM**	**External**
DARAM	1	1	1	1+p
SARAM	1	1	1	1+p
External	1+d	1+d	1+d	2+d+p

Cycles for a Repeat (RPT) Execution of an Instruction

	Program			
Operand	**ROM**	**DARAM**	**SARAM**	**External**
DARAM	n	n	n	n+p
SARAM	n	n	n	n+p
External	n+nd	n+nd	n+nd	n+1+p+nd

The column headings in these tables indicate the program source location, defined as follows:

ROM	The instruction executes from internal program ROM.
DARAM	The instruction executes from internal dual-access program RAM.
SARAM	The instruction executes from internal single-access program RAM.
External	The instruction executes from external program memory.

If an instruction requires memory operand(s), the rows in the table indicate the location(s) of the operand(s), as defined here:

DARAM	The operand is in internal dual-access RAM.
SARAM	The operand is in internal single-access RAM.
External	The operand is in external memory.

For the RPT mode execution, *n* indicates the number of times a given instruction is repeated by an RPT instruction. Additional cycles (wait states) can be generated for program-memory, data-memory, and I/O accesses by the wait-state generator or by the external READY signal. These additional wait states are represented in the tables by the following variables:

p	Program-memory wait states. Represents the number of additional clock cycles the device waits for external program memory to respond to a single access.
d	Data-memory wait states. Represents the number of additional clock cycles the device waits for external data memory to respond to a single access.
io	I/O wait states. Represents the number of additional clock cycles the device waits for an external I/O device to respond to a single access.
n	Number of repetitions (where $n > 2$ to fill the pipeline). Represents the number of times a repeated instruction is executed.

If there are multiple accesses to one of the spaces, the variable is preceded by the appropriate integer multiple. For example, two accesses to external program memory would require 2p wait states. The above variables may also use the subscripts *src, dst,* and *code* to indicate source, destination, and code, respectively.

The internal single-access memory on each ’C240 processor is divided into 2K-word blocks contiguous in address space. All ’C240 processors support parallel accesses to these internal single-access RAM blocks. Furthermore, one single access block allows only one access per cycle. Thus, the processor can read/write on single-access RAM block while accessing another single-access RAM block at the same time.

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. If the wait state generator or the READY pin is used to add *m* (*m* > 0) wait states to an external access, then external reads require *m* + 1 cycles, and external write accesses require *m* + 2 cycles.

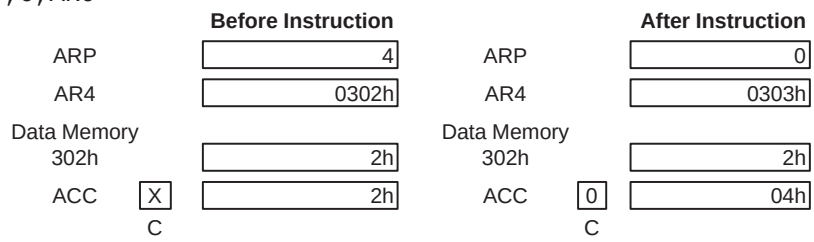
The instruction-cycle timings are based on the following assumptions:

- ❑ 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 the case of PC discontinuity instructions, such as 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 Section 5.2, *Pipeline Operation*, on page 5-7 for more information about pipeline operations.
- ❑ In the repeat execution mode, all conflicts caused by the pipelined execution of an instruction are considered.

7.2.9 Examples

Example code is included for each instruction. The effect of the code on memory and/or registers is summarized. Consider this example of the ADD instruction:

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



Here are the facts and events represented in this example:

- ❑ The auxiliary register pointer (ARP) points to the current auxiliary register. Because ARP = 4, the current auxiliary register is AR4.
- ❑ When the addition takes place, the CPU follows AR4 to data-memory address 0302h. The content of that address, 2h, is added to the content of the accumulator, also 2h. The result (4h) is placed in the accumulator. (Because the second operand of the instruction specifies a left shift of 0, the data-memory value is not shifted before being added to the accumulator value.)

- The instruction specifies an increment of 1 for the contents of the current auxiliary register (*+); therefore, after the addition is performed, the content of AR4 is incremented to 0303h.
- The instruction also specifies that AR0 is the next auxiliary register; therefore, after the instruction ARP = 0.
- Because no carry is generated during the addition, the carry bit (C) is cleared to 0.

7.3 Instruction Descriptions

This section contains detailed information on the instruction set for the ’C240. A summary of the instruction set is shown in Section 7.1 on page 7-2. The instructions are presented alphabetically, and the description for each instruction presents the following categories of information:

- Syntax
- Operands
- Opcode
- Execution
- Status Bits
- Description
- Words
- Cycles
- Examples

For a description of how to use each of these categories, see Section 7.2 on page 7-12.

Syntax **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 Increment PC, then ...
$|(ACC)| \rightarrow ACC; 0 \rightarrow C$

Status Bits

Affected by	*Affects*
OVM	C and OV

This instruction is 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 'C20x is always reset to zero by the execution of this instruction.

Note that 8000 0000h is a special case. When the overflow mode is not set (OVM = 0), the ABS of 8000 0000h is 8000 0000h. When the overflow mode is set (OVM = 1), the ABS of 8000 0000h is 7FFF FFFFh. In either case, the OV status bit is set.

Words 1

Cycles

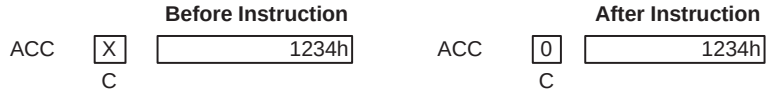
Cycles for a Single ABS Instruction

ROM	DARAM	SARAM	External
1	1	1	1+p

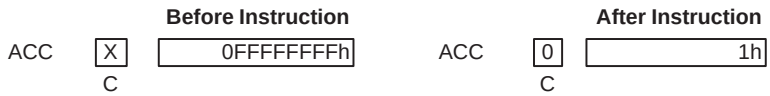
Cycles for a Repeat (RPT) Execution of an ABS Instruction

ROM	DARAM	SARAM	External
n	n	n	n+p

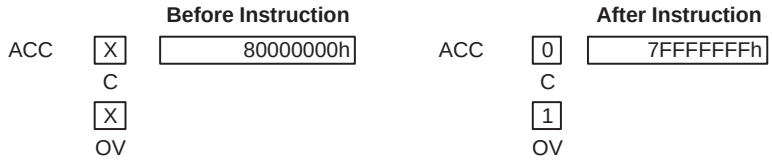
Example 1 ABS



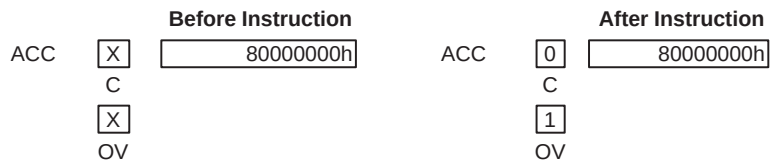
Example 2 ABS



Example 3 ABS ;(OVM = 1)



Example 4 ABS ;(OVM = 0)



Syntax

ADD *dma* [, *shift*]	Direct addressing
ADD *dma*, **16**	Direct with left shift of 16
ADD *ind* [, *shift* [, **AR***n*]]	Indirect addressing
ADD *ind*, **16** [, **AR***n*]	Indirect with left shift of 16
ADD #*k*	Short immediate addressing
ADD #*lk* [, *shift*]	Long immediate addressing

Operands

dma:	7 LSBs of the data-memory address
shift:	Left shift value from 0 to 15 (defaults to 0)
n:	Value from 0 to 7 designating the next auxiliary register
k:	8-bit short immediate value
lk:	16-bit long immediate value
ind:	Select one of the following seven options: * *+ *– *0+ *0– *BR0+ *BR0–

Opcode

ADD *dma* [, *shift*]

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

ADD *dma*, **16**

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

ADD *ind* [, *shift* [, **AR***n*]]

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

Note: ARU, N, and NAR are defined in section 6.3, *Indirect Addressing Mode* (page 6-9).

ADD *ind*, **16** [, **AR***n*]

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

Note: ARU, N, and NAR are defined in section 6.3, *Indirect Addressing Mode* (page 6-9).

ADD #*k*

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

ADD #*lk* [, *shift*]

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	0	0	1	shift			
lk															

Execution

Increment PC, then ...

Event	*Addressing mode*
(ACC) + ((data-memory address) $\times$ 2^{shift}) $\rightarrow$ ACC	Direct or indirect
(ACC) + ((data-memory address) $\times$ 2^{16}) $\rightarrow$ ACC	Direct or indirect (shift of 16)
(ACC) + k $\rightarrow$ ACC	Short immediate
(ACC) + lk $\times$ 2^{shift} $\rightarrow$ ACC	Long immediate

Status Bits

Affected by	*Affects*	*Addressing mode*
SXM and OVM	C and OV	Direct or indirect
OVM	C and OV	Short immediate
SXM and OVM	C and OV	Long immediate

Description

The content of the addressed data memory location or an immediate constant is 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.

If you are using indirect addressing and update the ARP, you must specify a shift operand. However, if you do not want a shift to occur, enter a 0 for this operand. For example:

```
ADD *+,0,AR2
```

Normally, the carry bit is set (C = 1) if the result of the addition generates a carry and is cleared (C = 0) if it does not generate a carry. However, when adding with a shift of 16, the carry bit is set if a carry is generated but otherwise, the carry bit is unaffected. This allows the accumulator to generate the proper single carry when adding a 32-bit number to the accumulator.

Words

Words	*Addressing mode*
1	Direct, indirect, or short immediate
2	Long immediate

Cycles

Cycles for a Single ADD Instruction (Using Direct and Indirect Addressing)

	Program			
Operand	**ROM**	**DARAM**	**SARAM**	**External**
DARAM	1	1	1	1+p
SARAM	1	1	1, 2†	1+p
External	1+d	1+d	1+d	2+d+p

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

Cycles for a Repeat (RPT) Execution of an ADD Instruction (Using Direct and Indirect Addressing)

	Program			
Operand	**ROM**	**DARAM**	**SARAM**	**External**
DARAM	n	n	n	n+p
SARAM	n	n	n, n+1†	n+p
External	n+nd	n+nd	n+nd	n+1+p+nd

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

Cycles for a Single ADD Instruction (Using Short Immediate Addressing)

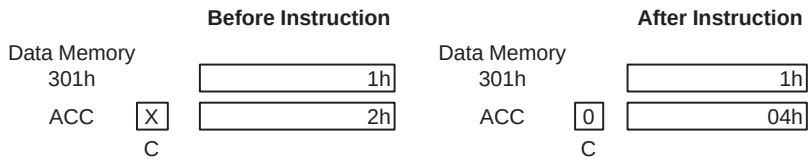
ROM	DARAM	SARAM	External
1	1	1	1+p

Cycles for a Single ADD Instruction (Using Long Immediate Addressing)

ROM	DARAM	SARAM	External
2	2	2	2+2p

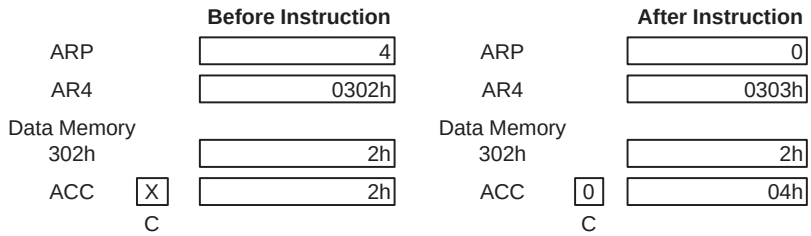
Example 1

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



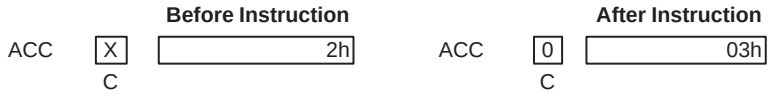
Example 2

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



Example 3

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



Example 4

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

Syntax

ADDC *dma* Direct addressing
ADDC *ind* [, **AR***n*] Indirect addressing

Operands

dma: 7 LSBs of the data-memory address
n: Value from 0 to 7 designating the next auxiliary register
ind: Select one of the following seven options:
* *+ *– *0+ *0– *BR0+ *BR0–

Opcode

ADDC *dma*

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

ADDC *ind* [, **AR***n*]

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

Note: ARU, N, and NAR are defined in section 6.3, *Indirect Addressing Mode* (page 6-9).

Execution

Increment PC, then ...
(ACC) + (data-memory address) + (C) → ACC

Status Bits

Affected by	*Affects*
OVM	C and OV

This instruction is not affected by SXM.

Description

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 carry bit is set (C = 1) if the result of the addition generates a carry and is cleared (C = 0) if it does not generate a carry.

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

Words

1

Cycles

Cycles for a Single ADDC Instruction

	Program			
Operand	**ROM**	**DARAM**	**SARAM**	**External**
DARAM	1	1	1	1+p
SARAM	1	1	1, 2†	1+p
External	1+d	1+d	1+d	2+d+p

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

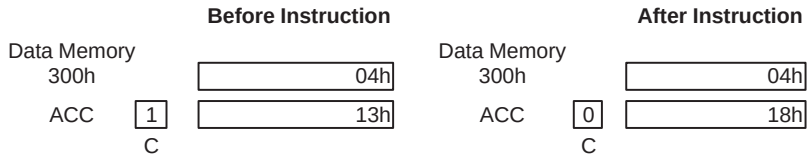
Cycles for a Repeat (RPT) Execution of an ADDC Instruction

	Program			
Operand	**ROM**	**DARAM**	**SARAM**	**External**
DARAM	n	n	n	n+p
SARAM	n	n	n, n+1†	n+p
External	n+nd	n+nd	n+nd	n+1+p+nd

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

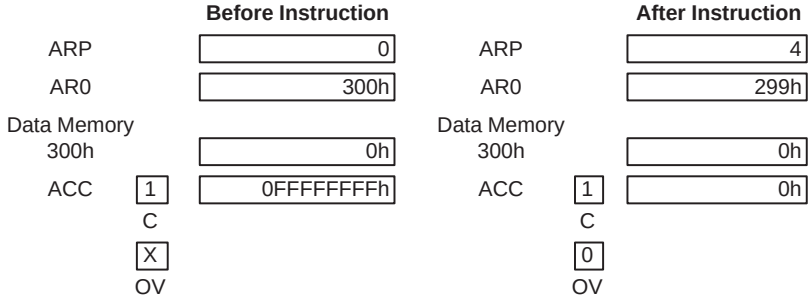
Example 1

```
ADDC    DAT300    ;(DP = 6: addresses 0300h–037Fh;
                  ;DAT300 is a label for 300h)
```



Example 2

```
ADDC    *–,AR4    ;(OVM = 0)
```



Syntax

ADDS *dma* Direct addressing
ADDS *ind* [, **AR***n*] Indirect addressing

Operands

dma: 7 LSBs of the data-memory address
n: Value from 0 to 7 designating the next auxiliary register
ind: Select one of the following seven options:
* *+ *– *0+ *0– *BR0+ *BR0–

Opcode

ADDS *dma*

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

ADDS *ind* [, **AR***n*]

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

Note: ARU, N, and NAR are defined in section 6.3, *Indirect Addressing Mode* (page 6-9).

Execution

Increment PC, then ...
(ACC) + (data-memory address) → ACC

Status Bits

Affected by	*Affects*
OVM	C and OV

This instruction is 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.

The carry bit is set (C = 1) if the result of the addition generates a carry and is cleared (C = 0) if it does not generate a carry.

Words

1

Cycles

Cycles for a Single ADDS Instruction

	Program			
Operand	**ROM**	**DARAM**	**SARAM**	**External**
DARAM	1	1	1	1+p
SARAM	1	1	1, 2†	1+p
External	1+d	1+d	1+d	2+d+p

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

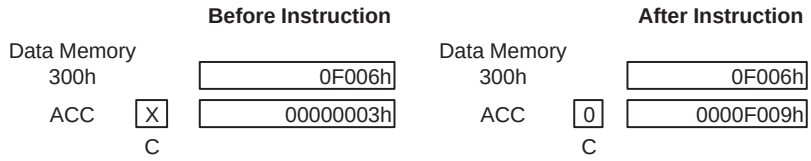
Cycles for a Repeat (RPT) Execution of an ADDS Instruction

	Program			
Operand	**ROM**	**DARAM**	**SARAM**	**External**
DARAM	n	n	n	n+p
SARAM	n	n	n, n+1†	n+p
External	n+nd	n+nd	n+nd	n+1+p+nd

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

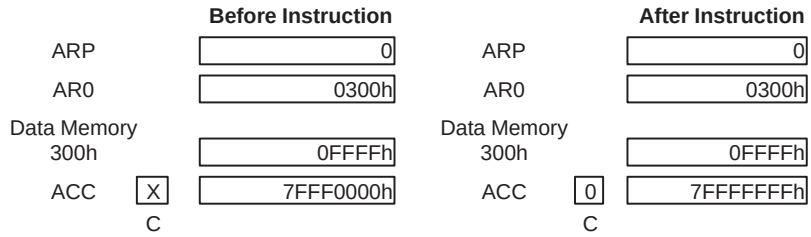
Example 1

```
ADDS    0          ;(DP = 6: addresses 0300h–037Fh)
```



Example 2

```
ADDS    *
```



Syntax

ADDT *dma* — Direct addressing
ADDT *ind* [, **AR***n*] — Indirect addressing

Operands

dma: 7 LSBs of the data-memory address
n: Value from 0 to 7 designating the next auxiliary register
ind: Select one of the following seven options:
* *+ *– *0+ *0– *BR0+ *BR0–

Opcode

ADDT *dma*

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

ADDT *ind* [, **AR***n*]

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

Note: ARU, N, and NAR are defined in section 6.3, *Indirect Addressing Mode* (page 6-9).

Execution

Increment PC, then ...
(ACC) + [(data-memory address) $\times$ $2^{(TREG(3:0))}$] $\rightarrow$ (ACC)

Status Bits

Affected by	*Affects*
SXM and OVM	C and OV

Description

The data-memory value is left shifted and added to the accumulator, and the result replaces the accumulator contents. The left shift is defined by the four LSBs of the TREG, resulting in shift options from 0 to 15 bits. Sign extension on the data-memory value is controlled by SXM. The carry bit (C) is set when a carry is generated out of the MSB of the accumulator; if no carry is generated, the carry bit is cleared.

Words

1

Cycles

Cycles for a Single ADDT Instruction

	Program			
Operand	**ROM**	**DARAM**	**SARAM**	**External**
DARAM	1	1	1	1+p
SARAM	1	1	1, 2†	1+p
External	1+d	1+d	1+d	2+d+p

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

Cycles for a Repeat (RPT) Execution of an ADDT Instruction

	Program			
Operand	**ROM**	**DARAM**	**SARAM**	**External**
DARAM	n	n	n	n+p
SARAM	n	n	n, n+1†	n+p
External	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    127     ;(DP = 4: addresses 0200h–027Fh,
                ;SXM = 0)
```

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

Example 2

```
ADDT    *–,AR4  ;(SXM = 0)
```

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

Syntax **ADRK** #*k* Short immediate addressing

Operands k: 8-bit short immediate value

Opcode

ADRK #*k*

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

Execution

Increment PC, then ...
(current AR) + 8-bit positive constant → current AR

Status Bits None

Description

The 8-bit immediate value is added, right justified, to the current auxiliary register (the one specified by the current ARP value) and the result replaces the auxiliary register contents. The addition takes place in the ARAU, with the immediate value treated as an 8-bit positive integer. All arithmetic operations on the auxiliary registers are unsigned.

Words 1

Cycles

Cycles for a Single ADRK Instruction

ROM	DARAM	SARAM	External
1	1	1	1+p

Example

```
ADRK    #80h
```

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

Syntax

AND *dma*	Direct addressing
AND *ind* [, **AR***n*]	Indirect addressing
AND #*lk* [, *shift*]	Long immediate addressing
AND #*lk*, **16**	Long immediate with left shift of 16

Operands

dma:	7 LSBs of the data-memory address
shift:	Left shift value from 0 to 15 (defaults to 0)
n:	Value from 0 to 7 designating the next auxiliary register
lk:	16-bit long immediate value
ind:	Select one of the following seven options: * *+ *– *0+ *0– *BR0+ *BR0–

Opcode

AND *dma*

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

AND *ind* [, **AR***n*]

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

Note: ARU, N, and NAR are defined in section 6.3, *Indirect Addressing Mode* (page 6-9).

AND #*lk* [, *shift*]

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	0	1	1	shift			
lk															

AND #*lk,* **16**

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	0	1
lk															

Execution

Increment PC, then ...

Event(s)	*Addressing mode*
(ACC(15:0)) AND (data-memory address) → ACC(15:0) 0 → ACC(31:16)	Direct or indirect
(ACC(31:0)) AND lk × 2^{shift} → ACC	Long immediate
(ACC(31:0)) AND lk × 2^{16} → ACC	Long immediate with left shift of 16

Status Bits

None

This instruction is 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 can be shifted. During the shift, low-order and high-order bits not filled by the shifted value are zeroed. The resulting value is ANDed with the accumulator contents.

Words

Words	*Addressing mode*
1	Direct or indirect
2	Long immediate

Cycles

Cycles for a Single AND Instruction (Using Direct and Indirect Addressing)

	Program			
Operand	**ROM**	**DARAM**	**SARAM**	**External**
DARAM	1	1	1	1+p
SARAM	1	1	1, 2†	1+p
External	1+d	1+d	1+d	2+d+p

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

Cycles for a Repeat (RPT) Execution of an AND Instruction (Using Direct and Indirect Addressing)

	Program			
Operand	**ROM**	**DARAM**	**SARAM**	**External**
DARAM	n	n	n	n+p
SARAM	n	n	n, n+1†	n+p
External	n+nd	n+nd	n+nd	n+1+p+nd

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

Cycles for a Single AND Instruction (Using Long Immediate Addressing)

ROM	**DARAM**	**SARAM**	**External**
2	2	2	2+2p

Example 1

```
AND     16          ;(DP = 4: addresses 0200h–027Fh)
```

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

Example 2

```
AND     *
```

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

Example 3

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

	Before Instruction	After Instruction
ACC	12345678h	00000670h

Syntax **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 Increment PC, then ...
(ACC) + shifted (PREG) → ACC

Status Bits

Affected by	*Affects*
PM and OVM	C and OV

This instruction is not affected by SXM.

Description The contents of PREG are shifted as defined by the PM status bits of the ST1 register (see Table 7–7) 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. PREG is always sign extended. The task of the APAC instruction is also performed as a subtask of the LTA, LTD, MAC, MACD, MPYA, and SQRA instructions.

Table 7–7. Product Shift Modes

PM Bits		
Bit 1	**Bit 0**	**Resulting Shift**
0	0	No shift
0	1	Left shift of 1 bit
1	0	Left shift of 4 bits
1	1	Right shift of 6 bits

Words 1

Cycles

Cycles for a Single APAC Instruction

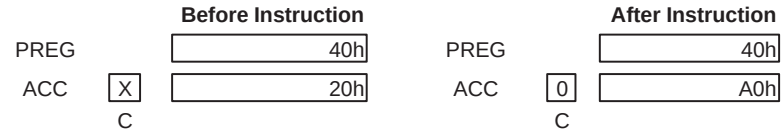
ROM	DARAM	SARAM	External
1	1	1	1+p

Cycles for a Repeat (RPT) Execution of an APAC Instruction

ROM	DARAM	SARAM	External
n	n	n	n+p

Example

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



Syntax **B** *pma* [, *ind* [, **AR***n*]] Indirect addressing

Operands

pma: 16-bit program-memory address
n: Value from 0 to 7 designating the next auxiliary register
ind: Select one of the following seven options:
* *+ *– *0+ *0– *BR0+ *BR0–

Opcode

B *pma* [, *ind* [, **AR***n*]]

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	ARU			N	NAR		
pma															

Note: ARU, N, and NAR are defined in section 6.3, *Indirect Addressing Mode* (page 6-9).

Execution

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

Status Bits None

Description The current auxiliary register and ARP contents are modified as specified, and control is passed to the designated program-memory address (pma). The pma can be either a symbolic or numeric address.

Words 2

Cycles

Cycles for a Single B Instruction

ROM	DARAM	SARAM	External
4	4	4	4+4p

Note: When this instruction reaches the execute phase of the pipeline, two additional instruction words have entered the pipeline. When the PC discontinuity is taken, these two instruction words are discarded.

Example

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

The value 191 is loaded into the program counter, and the program continues to execute from that location. The current auxiliary register is incremented by 1, and ARP is set to point to auxiliary register 1 (AR1).

Syntax **BACC**

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	0	0

Execution ACC(15:0) → PC

Status Bits None

Description Control is passed to the 16-bit address residing in the lower half of the accumulator.

Words 1

Cycles

Cycles for a Single BACC Instruction

ROM	DARAM	SARAM	External
4	4	4	4+3p

Note: When this instruction reaches the execute phase of the pipeline, two additional instruction words have entered the pipeline. When the PC discontinuity is taken, these two instruction words are discarded.

Example

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

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

Syntax

BANZ *pma* [, *ind* [, **AR***n*]] Indirect addressing

Operands

pma: 16-bit program-memory address
n: Value from 0 to 7 designating the next auxiliary register
ind: Select one of the following seven options:
* *+ *– *0+ *0– *BR0+ *BR0–

Opcode

BANZ *pma* [, *ind* [,**AR***n*]]

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	ARU			N	NAR		
pma															

Note: ARU, N, and NAR are defined in section 6.3, *Indirect Addressing Mode* (page 6-9).

Execution

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

Status Bits

None

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 the current AR is a decrement by one. N loop iterations can be executed by initializing an auxiliary register (as a loop counter) to N–1 prior to loop entry. The pma can be either a symbolic or a numeric address.

Words

2

Cycles

Cycles for a Single BANZ Instruction

Condition	ROM	DARAM	SARAM	External
True	4	4	4	4+4p
False	2	2	2	2+2p

Note: The 'C20x performs speculative fetching by reading two additional instruction words. If the PC discontinuity is taken, these two instruction words are discarded.

Example 1

```
BANZ      PGM0      ;(PGM0 labels program address 0)
```

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

Because the content of AR0 is not zero, the program address denoted by PGM0 is loaded into the program counter (PC), and the program continues executing from that location. The default auxiliary register operation is a decrement of the current auxiliary register content; thus, AR0 contains 4h at the end of the execution.

or

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

Because the content of AR0 is zero, the branch is not executed; instead, the PC is incremented by 2, and execution continues with the instruction following the BANZ instruction. Because of the default decrement, AR0 is decremented by 1, becoming –1.

Example 2

```
        MAR *,AR0          ;Set ARP to point to AR0.
        LAR AR1,#3         ;Load AR1 with 3.
        LAR AR0,#60h       ;Load AR0 with 60h.
PGM191  ADD *+,AR1         ;Loop: While AR1 not zero,
        BANZ PGM191,*–AR0  ;add data referenced by AR0
                           ;to accumulator and increment
                           ;AR0 value.
```

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

Syntax

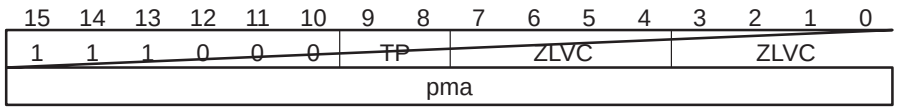
BCND *pma*, *cond* 1 [,*cond* 2] [,...]

Operands

pma: 16-bit program-memory address

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

Opcode



Note: The TP and ZLVC fields are defined on pages 7-3 and 7-4.

Execution

If *cond* 1 AND *cond* 2 AND ...
Then pma → PC
Else increment PC

Status Bits

None

Description

A branch is taken to the specified program-memory address (pma) if the specified conditions are met. Not all combinations of conditions are meaningful. For example, testing for LT and GT is contradictory. In addition, testing $\overline{\text{BIO}}$ is mutually exclusive to testing TC.

Words

2

Cycles

Cycles for a Single BCND Instruction

Condition	ROM	DARAM	SARAM	External
True	4	4	4	4+4p
False	2	2	2	2+2p

Note: The 'C20x performs speculative fetching by reading two additional instruction words. If the PC discontinuity is taken, these two instruction words are discarded.

Example

```
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 to execute from that location. If these conditions do not hold, execution continues from location PC + 2.

Syntax

BIT *dma*, *bit code* — Direct addressing
BIT *ind*, *bit code* [, **AR***n*] — Indirect addressing

Operands

dma: 7 LSBs of the data-memory address
bit code: Value from 0 to 15 indicating which bit to test (see Figure 7–1)
n: Value from 0 to 7 designating the next auxiliary register
ind: Select one of the following seven options:
* *+ *– *0+ *0– *BR0+ *BR0–

Opcode

BIT *dma*, *bit code*

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

BIT *ind*, *bit code* [,**AR***n*]

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	bit code				1	ARU			N	NAR		

Note: ARU, N, and NAR are defined in section 6.3, *Indirect Addressing Mode* (page 6-9).

Execution

Increment PC, then ...
(data bit number (15 – bit code)) → TC

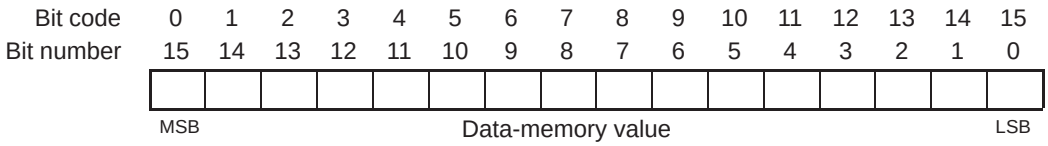
Status Bits

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, LST #1, and NORM instructions also affect the TC bit in ST1. A bit code value is specified that corresponds to a certain bit number of the data-memory value, as shown in Figure 7–1. For example, if you want to copy bit 6, you specify the bit code as 9, which is 15 minus six (15–6).

Figure 7–1. Bit Numbers and Their Corresponding Bit Codes for BIT Instruction



Words

1

Cycles

Cycles for a Single BIT Instruction

	Program			
Operand	**ROM**	**DARAM**	**SARAM**	**External**
DARAM	1	1	1	1+p
SARAM	1	1	1, 2†	1+p
External	1+d	1+d	1+d	2+d+p

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

Cycles for a Repeat (RPT) Execution of a BIT Instruction

	Program			
Operand	**ROM**	**DARAM**	**SARAM**	**External**
DARAM	n	n	n	n+p
SARAM	n	n	n, n+1†	n+p
External	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, then set ARP = 1
```

	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

BITT *dma* Direct addressing
BITT *ind* [, **AR***n*] Indirect addressing

Operands

dma: 7 LSBs of the data-memory address
n: Value from 0 to 7 designating the next auxiliary register
ind: Select one of the following seven options:
* *+ *– *0+ *0– *BR0+ *BR0–

Opcode

BITT *dma*

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

BITT *ind* [, **AR***n*]

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

Note: ARU, N, and NAR are defined in section 6.3, *Indirect Addressing Mode* (page 6-9).

Execution

Increment PC, then ...
(data bit number (15 –TREG(3:0))) → TC

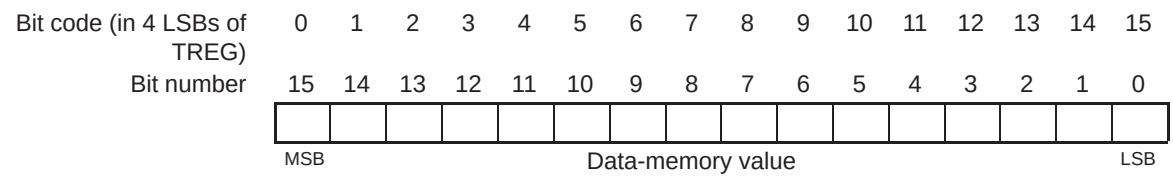
Status Bits

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, LST #1, and NORM instructions also affect the TC bit in status register ST1. The bit number is specified by a bit code value contained in the four LSBs of the TREG, as shown in Figure 7–2.

Figure 7–2. Bit Numbers and Their Corresponding Bit Codes for BITT Instruction



Words

1

Cycles

Cycles for a Single BITT Instruction

	Program			
Operand	**ROM**	**DARAM**	**SARAM**	**External**
DARAM	1	1	1	1+p
SARAM	1	1	1, 2†	1+p
External	1+d	1+d	1+d	2+d+p

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

Cycles for a Repeat (RPT) Execution of an BITT Instruction

	Program			
Operand	**ROM**	**DARAM**	**SARAM**	**External**
DARAM	n	n	n	n+p
SARAM	n	n	n, n+1†	n+p
External	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
TREG	1h	TREG	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
TREG	0Eh	TREG	0Eh
TC	0	TC	0

Syntax

General syntax: **BLDD** *source, destination*

BLDD #*lk*, *dma*	Direct with long immediate source
BLDD #*lk*, *ind* [, **AR***n*]	Indirect with long immediate source
BLDD *dma*, #*lk*	Direct with long immediate destination
BLDD *ind*, #*lk* [, **AR***n*]	Indirect with long immediate destination

Operands

dma:	7 LSBs of the data-memory address
n:	Value from 0 to 7 designating the next auxiliary register
lk:	16-bit long immediate value
ind:	Select one of the following seven options: * *+ *– *0+ *0– *BR0+ *BR0–

Opcode

BLDD #*lk*, *dma*

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

BLDD #*lk*, *ind* [, **AR***n*]

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

Note: ARU, N, and NAR are defined in section 6.3, *Indirect Addressing Mode* (page 6-9).

BLDD *dma*, #*lk*

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

BLDD *ind*, #*lk* [, **AR***n*]

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

Note: ARU, N, and NAR are defined in section 6.3, *Indirect Addressing Mode* (page 6-9).

Execution

Increment PC, then ...
(PC) → MSTACK
lk → PC
(source) → destination
For indirect, modify (current AR) and (ARP) as specified
(PC) + 1 → PC

While (repeat counter) ≠ 0:
    (source) → destination
    For indirect, modify (current AR) and (ARP) as specified
    (PC) + 1 → PC
    (repeat counter) –1 → repeat counter

(MSTACK) → PC

Status Bits

None

Description

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

Note:

BLDD will not work with memory-mapped registers.

RPT can be used with the BLDD instruction 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. When the BLDD instruction is repeated, the source (destination) address specified by the long immediate constant is stored to the PC. Because the PC is incremented by 1 during each repetition, it is possible to access a series of source (destination) addresses. If you use indirect addressing to specify the destination (source) address, a new destination (source) address can be accessed during each repetition. If you use the direct addressing mode, the specified destination (source) address is a constant; it will not be modified during each repetition.

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.

Words

2

Cycles

Cycles for a Single BLDD Instruction

Operand	ROM	DARAM	SARAM	External
Source: DARAM Destination: DARAM	3	3	3	3+2p
Source: SARAM Destination: DARAM	3	3	3	3+2p
Source: External 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: External Destination: SARAM	3+d_{src}	3+d_{src}	3+d_{src} 4+d_{src}†	3+d_{src}+2p
Source: DARAM Destination: External	4+d_{dst}	4+d_{dst}	4+d_{dst}	6+d_{dst}+2p
Source: SARAM Destination: External	4+d_{dst}	4+d_{dst}	4+d_{dst}	6+d_{dst}+2p
Source: External Destination: External	4+d_{src}+d_{dst}	4+d_{src}+d_{dst}	4+d_{src}+d_{dst}	6+d_{src}+d_{dst}+2p

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

Cycles for a Repeat (RPT) Execution of a BLDD Instruction

Operand	ROM	DARAM	SARAM	External
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: External Destination: DARAM	n+2+nd_{src}	n+2+nd_{src}	n+2+nd_{src}	n+2+nd_{src}+2p
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: External Destination: SARAM	n+2+nd_{src}	n+2+nd_{src}	n+2+nd_{src} n+4+nd_{src}†	n+2+nd_{src}+2p
Source: DARAM Destination: External	2n+2+nd_{dst}	2n+2+nd_{dst}	2n+2+nd_{dst}	2n+2+nd_{dst}+2p
Source: SARAM Destination: External	2n+2+nd_{dst}	2n+2+nd_{dst}	2n+2+nd_{dst}	2n+2+nd_{dst}+2p
Source: External Destination: External	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

Syntax

General syntax: **BLPD** *source*, *destination*

BLPD #*pma*, *dma* — Direct with long immediate source

BLPD #*pma*, *ind* [, **AR***n*] — Indirect with long immediate source

Operands

pma: 16-bit program-memory address
dma: 7 LSBs of the data-memory address
n: Value from 0 to 7 designating the next auxiliary register
ind: Select one of the following seven options:
* *+ *– *0+ *0– *BR0+ *BR0–

Opcode

BLPD *#pma, dma*

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

BLPD *#pma, ind* [*,* **AR***n*]

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

Note: ARU, N, and NAR are defined in section 6.3, *Indirect Addressing Mode* (page 6-9).

Execution

Increment PC, then ...
(PC) → MSTACK
pma → PC
(source) → destination
For indirect, modify (current AR) and (ARP) as specified
(PC) + 1 → PC

While (repeat counter) ≠ 0:
    (source) → destination
    For indirect, modify (current AR) and (ARP) as specified
    (PC) + 1 → PC
    (repeat counter) –1 → repeat counter

(MSTACK) → PC

Status Bits

None

Description

A word in program memory pointed to by the *source* is copied to data-memory space pointed to by *destination*. The first word of the source space is pointed to by a long-immediate value. The data-memory destination space is pointed to by a data-memory address or auxiliary register pointer. Not all source/destination combinations of pointer types are valid.

RPT can be used with the BLPD instruction to move consecutive words. 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. When the BLPD instruction is repeated, the source (program-memory) address specified by the long immediate constant is stored to the PC. Because the PC is incremented by 1 during each repetition, it is possible to access a series of program-memory addresses. If you use indirect addressing to specify the destination (data-memory) address, a new data-memory address can be accessed during each repetition. If you use the direct addressing mode, the specified data-memory address is a constant; it will not be modified during each repetition.

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

2

Cycles

Cycles for a Single BLPD Instruction

Operand	ROM	DARAM	SARAM	External
Source: DARAM/ROM Destination: DARAM	3	3	3	$3+2p_{code}$
Source: SARAM Destination: DARAM	3	3	3	$3+2p_{code}$
Source: External 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: External Destination: SARAM	$3+p_{src}$	$3+p_{src}$	$3+p_{src}$ $4+p_{src}$†	$3+p_{src}+2p_{code}$
Source: DARAM/ROM Destination: External	$4+d_{dst}$	$4+d_{dst}$	$4+d_{dst}$	$6+d_{dst}+2p_{code}$
Source: SARAM Destination: External	$4+d_{dst}$	$4+d_{dst}$	$4+d_{dst}$	$6+d_{dst}+2p_{code}$
Source: External Destination: External	$4+p_{src}+d_{dst}$	$4+p_{src}+d_{dst}$	$4+p_{src}+d_{dst}$	$6+p_{src}+d_{dst}+2p_{code}$

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

Cycles for a Repeat (RPT) Execution of a BLPD Instruction

Operand	ROM	DARAM	SARAM	External
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: External 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†	$n+2+2p_{code}$

† 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

Cycles for a Repeat (RPT) Execution of a BLPD Instruction (Continued)

Operand	ROM	DARAM	SARAM	External
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: External 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: External	2n+2+nd*dst*	2n+2+nd*dst*	2n+2+nd*dst*	2n+2+nd*dst*+2p*code*
Source: SARAM Destination: External	2n+2+nd*dst*	2n+2+nd*dst*	2n+2+nd*dst*	2n+2+nd*dst*+2p*code*
Source: External Destination: External	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)
```

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

Example 2

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

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

Syntax **CALA**

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	0	0

Execution

PC + 1 → TOS
ACC(15:0) → PC

Status Bits None

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.

The CALA instruction is used to perform computed subroutine calls.

Words 1

Cycles

Cycles for a Single CALA Instruction

ROM	DARAM	SARAM	External
4	4	4	4+3p

Note: When this instruction reaches the execute phase of the pipeline, two additional instruction words have entered the pipeline. When the PC discontinuity is taken, these two instruction words are discarded.

Example CALA

	Before Instruction		After Instruction
PC	25h	PC	83h
ACC	83h	ACC	83h
TOS	100h	TOS	26h

Syntax **CALL** *pma* [, *ind* [, **AR***n*]] Indirect addressing

Operands

pma: 16-bit program-memory address
n: Value from 0 to 7 designating the next auxiliary register
ind: Select one of the following seven options:
* *+ *– *0+ *0– *BR0+ *BR0–

Opcode

CALL *pma* [, *ind* [, **AR***n*]]

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	ARU			N	NAR		
pma															

Note: ARU, N, and NAR are defined in section 6.3, *Indirect Addressing Mode* (page 6-9).

Execution

PC + 2 → TOS
pma → PC
Modify (current AR) and (ARP) as specified.

Status Bits None

Description The current program counter (PC) is incremented and pushed onto the top of the stack (TOS). Then, the contents of the pma, either a symbolic or numeric address, are loaded into the PC. Execution continues at this address. The current auxiliary register and ARP contents are modified as specified.

Words 2

Cycles

Cycles for a Single CALL Instruction

ROM	DARAM	SARAM	External
4	4	4	4+4p†

Note: When this instruction reaches the execute phase of the pipeline, two additional instruction words have entered the pipeline. When the PC discontinuity is taken, these two instruction words are discarded.

Example

```
CALL    191,*+,AR0
```

	Before Instruction		After Instruction
ARP	1	ARP	0
AR1	05h	AR1	06h
PC	30h	PC	0BFh
TOS	100h	TOS	32h

Program address 0BFh (191) is loaded into the program counter, and the program continues executing from that location.

Syntax

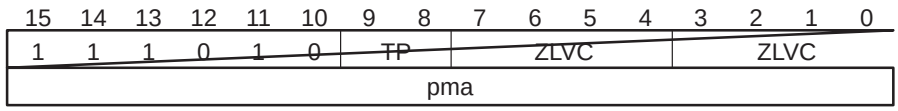
CC *pma*, *cond* 1 [,*cond* 2] [,...]

Operands

pma: 16-bit program-memory address

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

Opcode



Note: The TP and ZLVC fields are defined on pages 7-3 and 7-4.

Execution

If *cond* 1 AND *cond* 2 AND ...
Then
PC + 2 → TOS
pma → PC
Else
Increment PC

Status Bits

None

Description

Control is passed to the specified program-memory address (pma) if the specified conditions are met. Not all combinations of conditions are meaningful. For example, testing for LT and GT is contradictory. In addition, testing $\overline{\text{BIO}}$ is mutually exclusive to testing TC. The CC instruction operates like the CALL instruction if all conditions are true.

Words

2

Cycles

Cycles for a Single CC Instruction

Condition	ROM	DARAM	SARAM	External
True	4	4	4	4+4p†
False	2	2	2	2+2p

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

Example

```
CC          PGM191,LEQ,C
```

If the accumulator contents are less than or equal to zero and the carry bit is set, 0BFh (191) is loaded into the program counter, and the program continues to execute from that location. If the conditions are not met, execution continues at the instruction following the CC instruction.

Syntax

CLRC *control bit*

Operands

control bit: Select one of the following control bits:

C	Carry bit of status register ST1
CNF	RAM configuration control bit of status register ST1
INTM	Interrupt mode bit of status register ST0
OVM	Overflow mode bit of status register ST0
SXM	Sign-extension mode bit of status register ST1
TC	Test/control flag bit of status register ST1
XF	XF pin status bit of status register ST1

Opcode

CLRC 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

CLRC CNF

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

CLRC INTM

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

CLRC 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

CLRC 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

CLRC TC

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

CLRC XF

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

Increment PC, then ...
0 → control bit

Status Bits

None

Description

The specified control bit is cleared to 0. Note that the LST instruction can also be used to load ST0 and ST1. See section 4.5, *Status Registers ST0 and ST1* on page 4-15, for more information on each of these control bits.

Words 1

Cycles

Cycles for a Single CLRC Instruction

ROM	DARAM	SARAM	External
1	1	1	1+p

Cycles for a Repeat (RPT) Execution of a CLRC Instruction

ROM	DARAM	SARAM	External
n	n	n	n+p

Example

```
CLRC TC   ;(TC is bit 11 of ST1)
```

	Before Instruction		After Instruction
ST1	x9xxh	ST1	x1xxh

Syntax **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 Increment PC, then ...
$\overline{(ACC)} \rightarrow ACC$

Status Bits None

Description The contents of the accumulator are replaced with its logical inversion (1s complement). The carry bit is unaffected.

Words 1

Cycles

Cycles for a Single CMPL Instruction

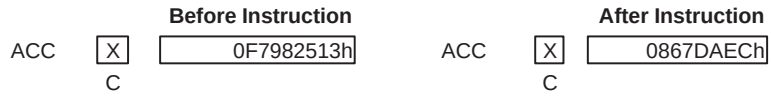
ROM	DARAM	SARAM	External
1	1	1	1+p

Cycles for a Repeat (RPT) Execution of an CMPL Instruction

ROM	DARAM	SARAM	External
n	n	n	n+p

Example

```
CMPL
```



Syntax **CMPR** *CM*

Operands CM: Value from 0 to 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	

Execution

Increment PC, then ...
Compare (current AR) to (AR0) and place the result in the TC bit of status register ST1.

Status Bits

Affects
TC

This instruction is not affected by SXM. It does not affect SXM.

Description

The CMPR instruction performs a comparison specified by the value of CM:

If CM = 00, test whether (current AR) = (AR0)
If CM = 01, test whether (current AR) < (AR0)
If CM = 10, test whether (current AR) > (AR0)
If CM = 11, test whether (current AR) ≠ (AR0)

If the condition is true, the TC bit is set to 1. If the condition is false, the TC bit is cleared to 0.

Note that the auxiliary register values are treated as unsigned integers in the comparisons.

Words 1

Cycles

Cycles for a Single CMPR Instruction

ROM	DARAM	SARAM	External
1	1	1	1+p

Cycles for a Repeat (RPT) Execution of an CMPR Instruction

ROM	DARAM	SARAM	External
n	n	n	n+p

Example

```
CMPR    2       ;(current AR) > (AR0)?
```

	Before Instruction		After Instruction
ARP	4	ARP	4
AR0	0FFFFh	AR0	0FFFFh
AR4	7FFFh	AR4	7FFFh
TC	1	TC	0

Syntax

DMOV *dma* Direct addressing
DMOV *ind* [, **AR***n*] Indirect addressing

Operands

dma: 7 LSBs of the data-memory address
n: Value from 0 to 7 designating the next auxiliary register
ind: Select one of the following seven options:
* *+ *– *0+ *0– *BR0+ *BR0–

Opcode

DMOV *dma*

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

DMOV *ind* [, **AR***n*]

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

Note: ARU, N, and NAR are defined in section 6.3, *Indirect Addressing Mode* (page 6-9).

Execution

Increment PC, then ...
(data-memory address) → data-memory address + 1

Status Bits

Affected by
CNF

Description

The contents of the specified data-memory address are copied into the contents of the next higher address. When data is copied from the addressed location to the next higher location, the contents of the addressed location remain unaltered.

DMOV works only within on-chip data DARAM 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 performed on external data memory. If the instruction specifies an external memory address, DMOV reads the specified memory location but performs *no* operations.

The data move function is useful in implementing the z^{-1} delay encountered in digital signal processing. The DMOV function is a subtask of the LTD and MACD instructions (see the LTD and MACD instructions for more information).

Words

1

Cycles

Cycles for a Single DMOV Instruction

	Program			
Operand	**ROM**	**DARAM**	**SARAM**	**External**
DARAM	1	1	1	1+p
SARAM	1	1	1, 3†	1+p
External‡	2+2d	2+2d	2+2d	5+2d+p

† If the operand and the code are in the same SARAM block
‡ If used on external memory, DMOV reads the specified memory location but performs no operations.

Cycles for a Repeat (RPT) Execution of a DMOV Instruction

	Program			
Operand	**ROM**	**DARAM**	**SARAM**	**External**
DARAM	n	n	n	n+p
SARAM	2n–2	2n–2	2n–2, 2n+1†	2n–2+p
External‡	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
‡ If used on external memory, DMOV reads the specified memory location but performs no operations.

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
AR0	30Ah	AR0	30Ah
Data Memory 30Ah	40h	Data Memory 30Ah	40h
Data Memory 30Bh	41h	Data Memory 30Bh	40h

Syntax **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 Increment PC, then wait for unmasked or nonmaskable hardware interrupt.

Status Bits *Affected by*
INTM

Description The IDLE instruction forces the program being executed to halt until the CPU receives a request from an unmasked hardware interrupt (external or internal), $\overline{\text{NMI}}$, or reset. Execution of the IDLE instruction causes the ’C24x/’C20x to enter a power-down mode. The PC is incremented once before the ’C24x/ ’C20x enters power down; it is not incremented during the idle state. On-chip peripherals remain active; thus, their interrupts are among those that can wake the processor.

The idle state is exited by an unmasked interrupt even if INTM is 1. (INTM, the interrupt mode bit of status register ST0, normally disables maskable interrupts when it is set to 1.) When the idle state is exited by an unmasked interrupt, the CPU’s next action, however, depends on INTM:

- If INTM is 0, the program branches to the corresponding interrupt service routine.
- If INTM is 1, the program continues executing at the instruction following the IDLE.

$\overline{\text{NMI}}$ and reset are not maskable; therefore, if the idle state is exited by $\overline{\text{NMI}}$ or reset, the corresponding interrupt service routine will be executed, regardless of INTM.

Words 1

Cycles

Cycles for a Single IDLE Instruction

ROM	DARAM	SARAM	External
1	1	1	1+p

Example

```
IDLE    ;The processor idles until a hardware reset,
        ;a hardware NMI, or an unmasked interrupt
        ;occurs.
```

Syntax

IN *dma*, *PA* — Direct addressing
IN *ind*, *PA* [, **AR***n*] — Indirect addressing

Operands

dma:	7 LSBs of the data-memory address
n:	Value from 0 to 7 designating the next auxiliary register
PA:	16-bit I/O port or I/O-mapped register address
ind:	Select one of the following seven options:
	* *+ *– *0+ *0– *BR0+ *BR0–

Opcode

IN *dma , PA*

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	dma						
PA															

IN *ind ,PA* [,**AR***n*]

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	ARU			N	NAR		
PA															

Note: ARU, N, and NAR are defined in section 6.3, *Indirect Addressing Mode* (page 6-9).

Execution

Increment PC, then ...
PA → address bus lines A15–A0
Data bus lines D15–D0 → data-memory address
(PA) → data-memory address

Status Bits

None

Description

The IN instruction reads a 16-bit value from an I/O location into the specified data-memory location. The $\overline{IS}$ line goes low to indicate an I/O access. The $\overline{STRB}$, $\overline{RD}$, and READY timings are the same as for an external data-memory read.

The repeat (RPT) instruction can be used with the IN instruction to read in consecutive words from I/O space to data space.

Words

2

Cycles

Cycles for a Single IN Instruction

	Program			
Operand	**ROM**	**DARAM**	**SARAM**	**External**
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: External	$3+d_{dst}+io_{src}$	$3+d_{dst}+io_{src}$	$3+d_{dst}+io_{src}$	$6+d_{dst}+io_{src}+2p_{code}$

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

Cycles for a Repeat (RPT) Execution of an IN Instruction

	Program			
Operand	**ROM**	**DARAM**	**SARAM**	**External**
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: External	$4n–1+nd_{dst}+nio_{src}$	$4n–1+nd_{dst}+nio_{src}$	$4n–1+nd_{dst}+nio_{src}$	$4n+2+nd_{dst}+nio_{src}+2p_{code}$

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

Example 1

```
IN    7,1000h      ;Read in word from peripheral on
                   ;port address 1000h. Store word in
                   ;data memory location 307h (DP=6).
```

Example 2

```
IN    *,5h         ;Read in word from peripheral on
                   ;port address 5h. Store word in
                   ;data memory location specified by
                   ;current auxiliary register.
```

Syntax **INTR** *K*

Operands K: Value from 0 to 31 that indicates the interrupt vector location to branch to

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	K				

Execution (PC) + 1 → stack
corresponding interrupt vector location → PC

Status Bits *Affects*
INTM

This instruction is not affected by INTM.

Description The processor has locations for 32 interrupt vectors; each location is represented by a value K from 0 to 31. The INTR instruction is a software interrupt that transfers program control to the program-memory address specified by K. The vector at that address then leads to the corresponding interrupt service routine. Thus, the instruction allows any one of the interrupt service routines to be executed from your software. For a list of interrupts and their corresponding K values, see Table 5–5 on page 5-15. During execution of the instruction, the value PC + 1 (the return address) is pushed onto the stack. Neither the INTM bit nor the interrupt masks affect the INTR instruction. An INTR for the external interrupts looks exactly like an external interrupt (an interrupt acknowledge is generated, and maskable interrupts are globally disabled by setting INTM = 1).

Words 1

Cycles

Cycles for a Single INTR Instruction

ROM	DARAM	SARAM	External
4	4	4	4+3p†

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

Example

```
INTR    3       ;PC + 1 is pushed onto the stack.
                ;Then control is passed to program
                ;memory location 6h.
```

Syntax

LACC *dma* [, *shift*]	Direct addressing
LACC *dma,* **16**	Direct with left shift of 16
LACC *ind* [, *shift* [, **AR***n*]]	Indirect addressing
LACC *ind*, **16**[, **AR***n*]	Indirect with left shift of 16
LACC #*lk* [, *shift*]	Long immediate addressing

Operands

dma:	7 LSBs of the data-memory address
shift:	Left shift value from 0 to 15 (defaults to 0)
n:	Value from 0 to 7 designating the next auxiliary register
lk:	16-bit long immediate value
ind:	Select one of the following seven options: * *+ *– *0+ *0– *BR0+ *BR0–

Opcode

LACC *dma* [, *shift*]

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

LACC *dma*, **16**

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

LACC *ind* [, *shift*[, **AR***n*]]

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

Note: ARU, N, and NAR are defined in section 6.3, *Indirect Addressing Mode* (page 6-9).

LACC *ind*, **16**[, **AR***n*]

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

Note: ARU, N, and NAR are defined in section 6.3, *Indirect Addressing Mode* (page 6-9).

LACC #*lk* [, *shift*]

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	0	0	0	shift			
lk															

Execution

Increment PC, then ...

Event	*Addressing mode*
(data-memory address) $\times 2^{shift} \rightarrow$ ACC	Direct or indirect
(data-memory address) $\times 2^{16} \rightarrow$ ACC	Direct or indirect (shift of 16)
lk $\times 2^{shift} \rightarrow$ ACC	Long immediate

Status Bits

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

Words	*Addressing mode*
1	Direct or indirect
2	Long immediate

Cycles

Cycles for a Single LACC Instruction (Using Direct and Indirect Addressing)

	Program			
Operand	**ROM**	**DARAM**	**SARAM**	**External**
DARAM	1	1	1	1+p
SARAM	1	1	1, 2†	1+p
External	1+d	1+d	1+d	2+d+p

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

Cycles for a Repeat (RPT) Execution of an LACC Instruction (Using Direct and Indirect Addressing)

	Program			
Operand	**ROM**	**DARAM**	**SARAM**	**External**
DARAM	n	n	n	n+p
SARAM	n	n	n, n+1†	n+p
External	n+nd	n+nd	n+nd	n+1+p+nd

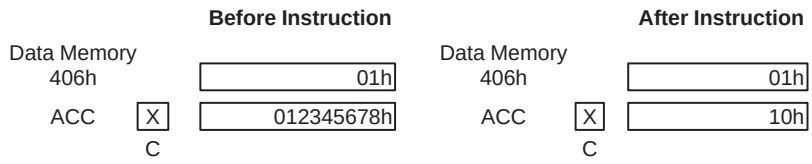
† If the operand and the code are in the same SARAM block

Cycles for a Single LACC Instruction (Using Immediate Addressing)

ROM	**DARAM**	**SARAM**	**External**
2	2	2	2+2p

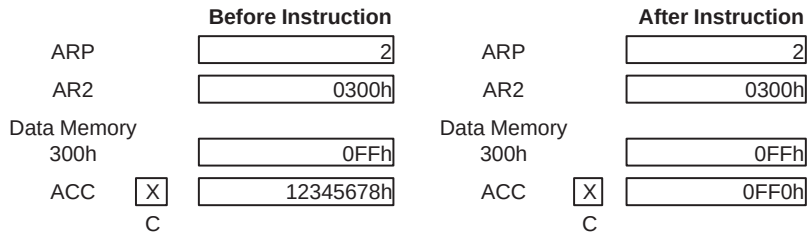
Example 1

```
LACC    6,4        ;(DP = 8: addresses 0400h–047Fh,
                   ;SXM = 0)
```



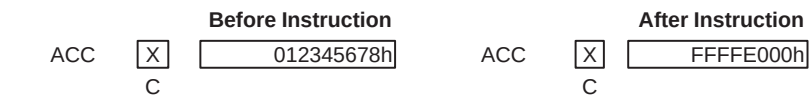
Example 2

```
LACC    *,4        ;(SXM = 0)
```



Example 3

```
LACC    #0F000h,1 ;(SXM = 1)
```



Syntax

LACL *dma*	Direct addressing
LACL *ind* [, **AR***n*]	Indirect addressing
LACL #*k*	Short immediate

Operands

dma:	7 LSBs of the data-memory address
n:	Value from 0 to 7 designating the next auxiliary register
k:	8-bit short immediate value
ind:	Select one of the following seven options: * *+ *– *0+ *0– *BR0+ *BR0–

Opcode

LACL *dma*

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

LACL *ind* [, **AR***n*]

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

Note: ARU, N, and NAR are defined in section 6.3, *Indirect Addressing Mode* (page 6-9).

LACL #*k*

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

Execution

Increment PC, then ...

Events	*Addressing mode*
0 → ACC(31:16) (data-memory address) → ACC(15:0)	Direct or indirect
0 → ACC(31:8) k → ACC(7:0)	Short immediate

Status Bits

This instruction is 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

Cycles for a Single LACL Instruction (Using Direct and Indirect Addressing)

	Program			
Operand	**ROM**	**DARAM**	**SARAM**	**External**
DARAM	1	1	1	1+p
SARAM	1	1	1, 2†	1+p
External	1+d	1+d	1+d	2+d+p

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

Cycles for a Repeat (RPT) Execution of an LACL Instruction (Using Direct and Indirect Addressing)

	Program			
Operand	**ROM**	**DARAM**	**SARAM**	**External**
DARAM	n	n	n	n+p
SARAM	n	n	n, n+1†	n+p
External	n+nd	n+nd	n+nd	n+1+p+nd

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

Cycles for a Single LACL Instruction (Using Immediate Addressing)

ROM	**DARAM**	**SARAM**	**External**
1	1	1	1+p

Example 1

```
LACL    1          ;(DP = 6: addresses 0300h–037Fh)
```

	Before Instruction	After Instruction
Data Memory 301h	0h	0h
ACC	7FFFFFFFh	0h
X		
C		

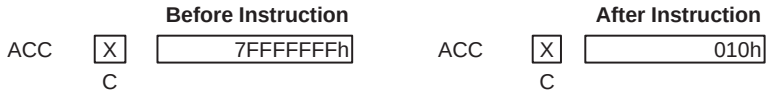
Example 2

```
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
```



Syntax

LACT *dma* Direct addressing
LACT *ind* [, **AR***n*] Indirect addressing

Operands

dma: 7 LSBs of the data-memory address
n: Value from 0 to 7 designating the next auxiliary register
ind: Select one of the following seven options:
* *+ *– *0+ *0– *BR0+ *BR0–

Opcode

LACT *dma*

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

LACT *ind* [, **AR***n*]

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

Note: ARU, N, and NAR are defined in section 6.3, *Indirect Addressing Mode* (page 6-9).

Execution

Increment PC, then ...
(data-memory address) $\times$ $2^{(TREG(3:0))}$ $\rightarrow$ ACC

If SXM = 1:
Then (data-memory address) is sign extended.
If SXM = 0:
Then (data-memory address) is not sign extended.

Status Bits

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 the TREG, resulting in shift options from 0 to 15 bits. Using the four LSBs of the TREG 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 TREG register and the mantissa is referenced by the data-memory address. This method of denormalization can be used only when the magnitude of the exponent has four bits or less.

Words

1

Cycles

Cycles for a Single LACT Instruction

	Program			
Operand	**ROM**	**DARAM**	**SARAM**	**External**
DARAM	1	1	1	1+p
SARAM	1	1	1, 2†	1+p
External	1+d	1+d	1+d	2+d+p

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

Cycles for a Repeat (RPT) Execution of an LACT Instruction

	Program			
Operand	**ROM**	**DARAM**	**SARAM**	**External**
DARAM	n	n	n	n+p
SARAM	n	n	n, n+1†	n+p
External	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    1          ;(DP = 6: addresses 0300h–037Fh,
                   ;SXM = 0)
```

	Before Instruction	After Instruction
Data Memory 301h	1376h	1376h
TREG	14h	14h
ACC	98F7EC83h	13760h
X		
C		

Example 2

```
LACT    *–,AR3     ;(SXM = 1)
```

	Before Instruction	After Instruction
ARP	1	3
AR1	310h	30Fh
Data Memory 310h	0FF00h	0FF00h
TREG	11h	11h
ACC	098F7EC83h	0FFFFFE00h
X		
C		

Syntax

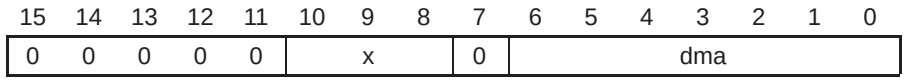
LAR AR*x*, *dma*	Direct addressing
LAR AR*x*, *ind* [, **AR***n*]	Indirect addressing
LAR AR*x*, *#k*	Short immediate addressing
LAR AR*x*, *#lk*	Long immediate addressing

Operands

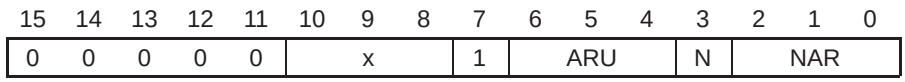
x:	Value from 0 to 7 designating the auxiliary register to be loaded
dma:	7 LSBs of the data-memory address
k:	8-bit short immediate value
lk:	16-bit long immediate value
n:	Value from 0 to 7 designating the next auxiliary register
ind:	Select one of the following seven options: * *+ *– *0+ *0– *BR0+ *BR0–

Opcode

LAR AR*x*, *dma*

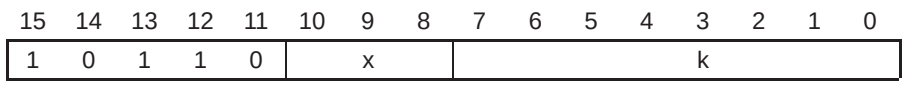


LAR AR*x*, *ind* [, **AR***n*]

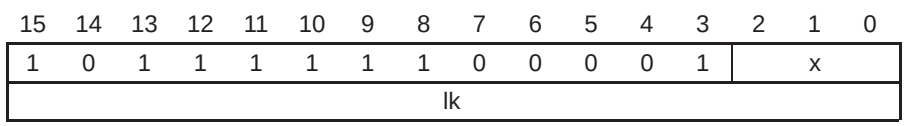


Note: ARU, N, and NAR are defined in section 6.3, *Indirect Addressing Mode* (page 6-9).

LAR AR*x*, *#k*



LAR AR*x*, *#lk*



Execution

Increment PC, then ...

Event	*Addressing mode*
(data-memory address) → ARx	Direct or indirect
k → ARx	Short immediate
lk → ARx	Long immediate

Status Bits

None

Description

The contents of the specified data-memory address or an 8-bit or 16-bit constant are loaded into the specified auxiliary register (ARx). The specified constant is treated as an unsigned integer, regardless of the value of SXM.

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 enable 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

Words	*Addressing mode*
1	Direct, indirect or short immediate
2	Long immediate

Cycles

Cycles for a Single LAR Instruction (Using Direct and Indirect Addressing)

	Program			
Operand	**ROM**	**DARAM**	**SARAM**	**External**
DARAM	2	2	2	$2+p_{code}$
SARAM	2	2	2, 3†	$2+p_{code}$
External	$2+d_{src}$	$2+d_{src}$	$2+d_{src}$	$3+d_{src}+p_{code}$

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

Cycles for a Repeat (RPT) Execution of an LAR Instruction (Using Direct and Indirect Addressing)

	Program			
Operand	**ROM**	**DARAM**	**SARAM**	**External**
DARAM	2n	2n	2n	$2n+p_{code}$
SARAM	2n	2n	2n, 2n+1†	$2n+p_{code}$
External	$2n+nd_{src}$	$2n+nd_{src}$	$2n+nd_{src}$	$2n+1+nd_{src}p_{code}$

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

Cycles for a Single LAR Instruction (Using Short Immediate Addressing)

ROM	**DARAM**	**SARAM**	**External**
2	2	2	$2+p_{code}$

Cycles for a Single LAR Instruction (Using Long Immediate Addressing)

ROM	**DARAM**	**SARAM**	**External**
2	2	2	2+2p

Example 1

```
LAR     AR0,16    ;(DP = 6: addresses 0300h–037Fh)
```

	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     AR6,#3FFFh
```

	Before Instruction		After Instruction
AR6	0h	AR6	3FFFh

Syntax

LDP *dma*	Direct addressing
LDP *ind* [, **AR***n*]	Indirect addressing
LDP #*k*	Short immediate addressing

Operands

dma:	7 LSBs of the data-memory address
n:	Value from 0 to 7 designating the next auxiliary register
k:	9-bit short immediate value
ind:	Select one of the following seven options: * *+ *– *0+ *0– *BR0+ *BR0–

Opcode

LDP *dma*

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

LDP *ind* [, **AR***n*]

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

Note: ARU, N, and NAR are defined in section 6.3, *Indirect Addressing Mode* (page 6-9).

LDP #*k*

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

Execution

Increment PC, then ...

Event	*Addressing mode*
Nine LSBs of (data-memory address) → DP	Direct or indirect
k → DP	Short immediate

Status Bits

Affects
DP

Description

The nine LSBs of the contents of the addressed data-memory location or a 9-bit immediate value is loaded into the data page pointer (DP) of status register ST0. The DP can also be loaded by the LST instruction.

In direct addressing, the 9-bit DP and the 7-bit value specified in the instruction (dma) are concatenated to form the 16-bit data-memory address accessed by the instruction. The DP provides the 9 MSBs, and dma provides the 7 LSBs.

Words

1

Cycles

Cycles for a Single LDP Instruction (Using Direct and Indirect Addressing)

	Program			
Operand	**ROM**	**DARAM**	**SARAM**	**External**
DARAM	2	2	2	$2+p_{code}$
SARAM	2	2	2, 3†	$2+p_{code}$
External	$2+d_{src}$	$2+d_{src}$	$2+d_{src}$	$3+d_{src}+p_{code}$

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

Cycles for a Repeat (RPT) Execution of an LDP Instruction (Using Direct and Indirect Addressing)

	Program			
Operand	**ROM**	**DARAM**	**SARAM**	**External**
DARAM	2n	2n	2n	$2n+p_{code}$
SARAM	2n	2n	2n, 2n+1†	$2n+p_{code}$
External	$2n+nd_{src}$	$2n+nd_{src}$	$2n+nd_{src}$	$2n+1+nd_{src}p_{code}$

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

Cycles for a Single LDP Instruction (Using Short Immediate Addressing)

ROM	**DARAM**	**SARAM**	**External**
2	2	2	$2+p_{code}$

Example 1

```
LDP     127     ;(DP = 511: addresses FF80h–FFFFh)
```

	Before Instruction		After Instruction
Data Memory FFFFh	FEDCh	Data Memory FFFFh	FEDCh
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

LPH *dma* Direct addressing
LPH *ind* [, **AR***n*] Indirect addressing

Operands

dma: 7 LSBs of the data-memory address
n: Value from 0 to 7 designating the next auxiliary register
ind: Select one of the following seven options:
* *+ *– *0+ *0– *BR0+ *BR0–

Opcode

LPH *dma*

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

LPH *ind* [, **AR***n*]

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

Note: ARU, N, and NAR are defined in section 6.3, *Indirect Addressing Mode* (page 6-9).

Execution

Increment PC, then ...
(data-memory address) → PREG (31:16)

Status Bits

None

Description

The 16 high-order bits of the PREG are loaded with the content of the specified data-memory address. The low-order PREG bits are unaffected.

The LPH instruction can be used for restoring the high-order bits of the PREG after interrupts and subroutine calls.

Words

1

Cycles

Cycles for a Single LPH Instruction

	Program			
Operand	**ROM**	**DARAM**	**SARAM**	**External**
DARAM	1	1	1	1+p
SARAM	1	1	1, 2†	1+p
External	1+d	1+d	1+d	2+d+p

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

Cycles for a Repeat (RPT) Execution of an LPH Instruction

	Program			
Operand	**ROM**	**DARAM**	**SARAM**	**External**
DARAM	n	n	n	n+p
SARAM	n	n	n, n+1†	n+p
External	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
PREG	30079844h	PREG	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
PREG	30079844h	PREG	0F79C9844h

Syntax

LST #*m*, *dma* Direct addressing
LST #*m*, *ind* [, **AR***n*] Indirect addressing

Operands

dma: 7 LSBs of the data-memory address
n: Value from 0 to 7 designating the next auxiliary register
m: Select one of the following:
0 Indicates that ST0 will be loaded
1 Indicates that ST1 will be loaded
ind: Select one of the following seven options:
* *+ *– *0+ *0– *BR0+ *BR0–

Opcode

LST #0, *dma*

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

LST #0, *ind* [, **AR***n*]

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

Note: ARU, N, and NAR are defined in section 6.3, *Indirect Addressing Mode* (page 6-9).

LST #1, *dma*

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

LST #1, *ind* [, **AR***n*]

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

Note: ARU, N, and NAR are defined in section 6.3, *Indirect Addressing Mode* (page 6-9).

Execution

Increment PC, then ...
(data-memory address) → status register STm

For details about the differences between an LST #0 operation and an LST #1 operation, see Figure 7–3, Figure 7–4, and the description category below.

Figure 7–3. LST #0 Operation

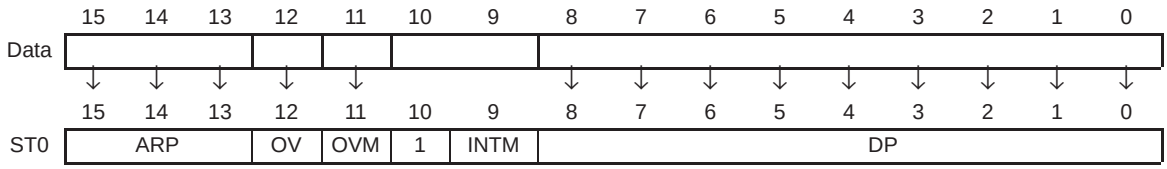
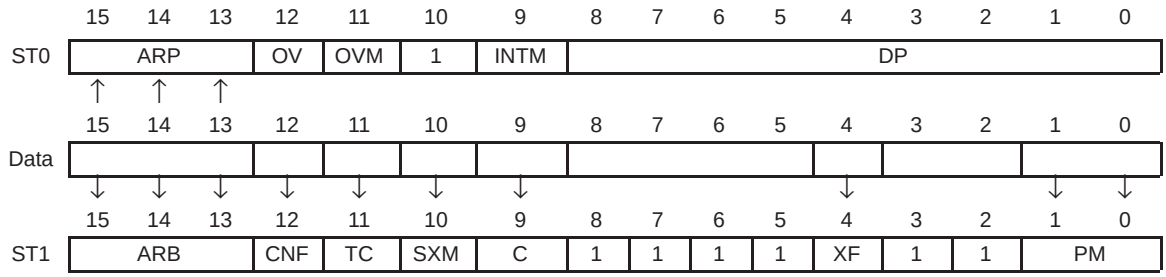


Figure 7–4. LST #1 Operation



Status Bits

Affects
ARB, ARP, OV, OVM, DP, CNF, TC, SXM, C, XF, and PM

This instruction does not affect INTM.

Description

The specified status register (ST0 or ST1) is loaded with the addressed data-memory value. Note the following points:

- The LST #0 operation does not affect the ARB field in the ST1 register, even though a new ARP is loaded.
- During the LST #1 operation, the value loaded into ARB is also loaded into ARP.
- If a next AR value is specified as an operand in the indirect addressing mode, this operand is ignored. ARP is loaded with the three MSBs of the value contained in the addressed data-memory location.
- Reserved bit values in the status registers are always read as 1s. Writes to these bits have no effect.

The LST instruction can be used for restoring the status registers after subroutine calls and interrupts.

Words

1

Cycles

Cycles for a Single LST Instruction

	Program			
Operand	**ROM**	**DARAM**	**SARAM**	**External**
DARAM	2	2	2	$2+p_{code}$
SARAM	2	2	2, 3†	$2+p_{code}$
External	$2+d_{src}$	$2+d_{src}$	$2+d_{src}$	$3+d_{src}+p_{code}$

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

Cycles for a Repeat (RPT) Execution of an LST Instruction

	Program			
Operand	**ROM**	**DARAM**	**SARAM**	**External**
DARAM	2n	2n	2n	2n+p_{code}
SARAM	2n	2n	2n, 2n+1†	2n+p_{code}
External	2n+nd_{src}	2n+nd_{src}	2n+nd_{src}	2n+1+nd_{src}+p_{code}

† If the 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. Also note that
                ;the old ARP is not loaded into the
                ;ARB.
```

Example 2

```
LST     #0,60h    ;(DP = 0)
```

	Before Instruction	After Instruction
Data Memory 60h	2404h	2404h
ST0	6E00h	2604h
ST1	05ECh	05ECh

Example 3

```
LST     #0,*–,AR1
```

	Before Instruction	After Instruction
ARP	4	7
AR4	3FFh	3FEh
Data Memory 3FFh	EE04h	EE04h
ST0	EE00h	EE04h
ST1	F7ECh	F7ECh

Example 4

```
LST     #1,00h    ;(DP = 6)
                  ;Note that the ARB is loaded with
                  ;the new ARP value.
```

	Before Instruction		After Instruction
Data Memory 300h	E1BCh	Data Memory 300h	E1BCh
ST0	0406h	ST0	E406h
ST1	09ECh	ST1	E1FCh

Syntax

LT *dma* Direct addressing
LT *ind* [, **AR***n*] Indirect addressing

Operands

dma: 7 LSBs of the data-memory address
n: Value from 0 to 7 designating the next auxiliary register
ind: Select one of the following seven options:
* *+ *– *0+ *0– *BR0+ *BR0–

Opcode

LT *dma*

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

LT *ind* [, **AR***n*]

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

Note: ARU, N, and NAR are defined in section 6.3, *Indirect Addressing Mode* (page 6-9).

Execution

Increment PC, then ...
(data-memory address) → TREG

Status Bits

None

Description

TREG is loaded with the contents of the specified data-memory address. The LT instruction may be used to load TREG in preparation for multiplication. See also the LTA, LTD, LTP, LTS, MPY, MPYA, MPYS, and MPYU instructions.

Words

1

Cycles

Cycles for a Single LT Instruction

	Program			
Operand	**ROM**	**DARAM**	**SARAM**	**External**
DARAM	1	1	1	1+p
SARAM	1	1	1, 2†	1+p
External	1+d	1+d	1+d	2+d+p

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

Cycles for a Repeat (RPT) Execution of an LT Instruction

	Program			
Operand	**ROM**	**DARAM**	**SARAM**	**External**
DARAM	n	n	n	n+p
SARAM	n	n	n, n+1†	n+p
External	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    24          ;(DP = 8: addresses 0400h–047Fh)
```

	Before Instruction	After Instruction
Data Memory 418h	62h	62h
TREG	3h	62h

Example 2

```
LT    *,AR3
```

	Before Instruction	After Instruction
ARP	2	3
AR2	418h	418h
Data Memory 418h	62h	62h
TREG	3h	62h

Syntax

LTA *dma* Direct addressing
LTA *ind* [, **AR***n*] Indirect addressing

Operands

dma: 7 LSBs of the data-memory address
n: Value from 0 to 7 designating the next auxiliary register
ind: Select one of the following seven options:
* *+ *– *0+ *0– *BR0+ *BR0–

Opcode

LTA *dma*

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

LTA *ind* [, **AR***n*]

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

Note: ARU, N, and NAR are defined in section 6.3, *Indirect Addressing Mode* (page 6-9).

Execution

Increment PC, then ...
(data-memory address) → TREG
(ACC) + shifted (PREG) → ACC

Status Bits

Affected by	*Affects*
PM and OVM	C and OV

Description

TREG is loaded with the contents of the specified data-memory address. The contents of the product register, shifted as defined by the PM status bits, are added to the accumulator, and the result is placed in the accumulator.

The carry bit is set (C = 1) if the result of the addition generates a carry and is cleared (C = 0) if it does not generate a carry.

The function of the LTA instruction is a subtask of the LTD instruction.

Words 1

Cycles

Cycles for a Single LTA Instruction

	Program			
Operand	**ROM**	**DARAM**	**SARAM**	**External**
DARAM	1	1	1	1+p
SARAM	1	1	1, 2†	1+p
External	1+d	1+d	1+d	2+d+p

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

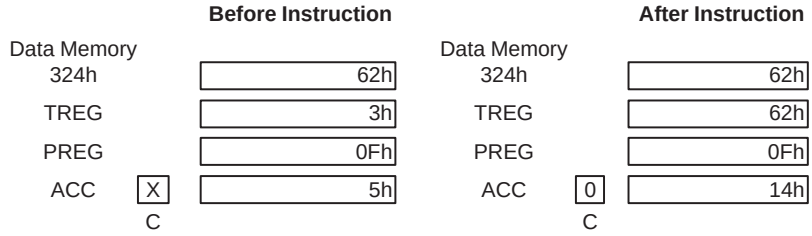
Cycles for a Repeat (RPT) Execution of an LTA Instruction

	Program			
Operand	**ROM**	**DARAM**	**SARAM**	**External**
DARAM	n	n	n	n+p
SARAM	n	n	n, n+1†	n+p
External	n+nd	n+nd	n+nd	n+1+p+nd

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

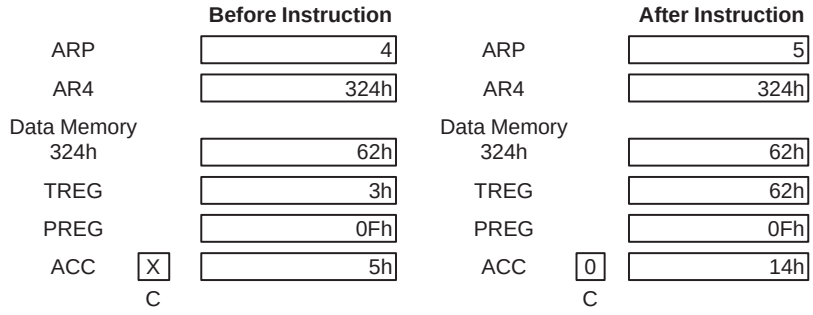
Example 1

```
LTA     36      ;(DP = 6: addresses 0300h–037Fh,
                ;PM =0: no shift of product)
```



Example 2

```
LTA     *,AR5   ;(PM = 0)
```



Syntax

LTD *dma* Direct addressing
LTD *ind* [, **AR***n*] Indirect addressing

Operands

dma: 7 LSBs of the data-memory address
n: Value from 0 to 7 designating the next auxiliary register
ind: Select one of the following seven options:
* *+ *– *0+ *0– *BR0+ *BR0–

Opcode

LTD *dma*

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

LTD *ind* [, **AR***n*]

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

Note: ARU, N, and NAR are defined in section 6.3, *Indirect Addressing Mode* (page 6-9).

Execution

Increment PC, then ...
(data-memory address) → TREG
(data-memory address) → data-memory address + 1
(ACC) + shifted (PREG) → ACC

Status Bits

Affected by	*Affects*
PM and OVM	C and OV

Description

TREG is loaded with the contents of the specified data-memory address. The contents of the PREG, 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.

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. The data move function is described under the instruction DMOV.

Note:

If LTD is used with external data memory, its function is identical to that of LTA; that is, the previous product will be accumulated, and the TREG will be loaded from external data memory, but *the data move will not occur*.

The carry bit is set (C = 1) if the result of the addition generates a carry and is cleared (C = 0) if it does not generate a carry.

Words 1

Cycles

Cycles for a Single LTD Instruction

	Program			
Operand	**ROM**	**DARAM**	**SARAM**	**External‡**
DARAM	1	1	1	1+p
SARAM	1	1	1, 3†	1+p
External	2+2d	2+2d	2+2d	5+2d+p

† If the operand and the code are in the same SARAM block
‡ If the LTD instruction is used with external memory, the data move will not occur. (The previous product will be accumulated, and the TREG will be loaded.)

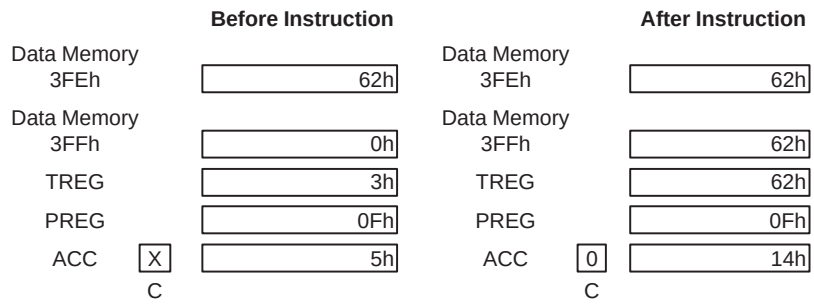
Cycles for a Repeat (RPT) Execution of an LTD Instruction

	Program			
Operand	**ROM**	**DARAM**	**SARAM**	**External‡**
DARAM	n	n	n	n+p
SARAM	2n–2	2n–2	2n–2, 2n+1†	2n–2+p
External	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
‡ If the LTD instruction is used with external memory, the data move will not occur. (The previous product will be accumulated, and the TREG will be loaded.)

Example 1

```
LTD     126     ;(DP = 7: addresses 0380h–03FFh,
                ;PM = 0: no shift of product).
```



Example 2

```
LTD     *,AR3     ;(PM = 0)
```

	Before Instruction		After Instruction
ARP	1	ARP	3
AR1	3FEh	AR1	3FEh
Data Memory 3FEh	62h	Data Memory 3FEh	62h
Data Memory 3FFh	0h	Data Memory 3FFh	62h
TREG	3h	TREG	62h
PREG	0Fh	PREG	0Fh
ACC	5h	ACC	14h
C	X	C	0

Note: The data move function for LTD can occur only within on-chip data memory RAM blocks.

Syntax

LTP *dma* Direct addressing
LTP *ind* [, **AR***n*] Indirect addressing

Operands

dma: 7 LSBs of the data-memory address
n: Value from 0 to 7 designating the next auxiliary register
ind: Select one of the following seven options:
* *+ *– *0+ *0– *BR0+ *BR0–

Opcode

LTP *dma*

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

LTP *ind* [, **AR***n*]

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

Note: ARU, N, and NAR are defined in section 6.3, *Indirect Addressing Mode* (page 6-9).

Execution

Increment PC, then ...
(data-memory address) → TREG
shifted (PREG) → ACC

Status Bits

Affected by
PM

Description

The TREG is loaded with the content of the addressed data-memory location, and the PREG value is stored in the accumulator. The shift at the output of the PREG is controlled by the PM status bits.

Words

1

Cycles

Cycles for a Single LTP Instruction

	Program			
Operand	**ROM**	**DARAM**	**SARAM**	**External**
DARAM	1	1	1	1+p
SARAM	1	1	1, 2†	1+p
External	1+d	1+d	1+d	2+d+p

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

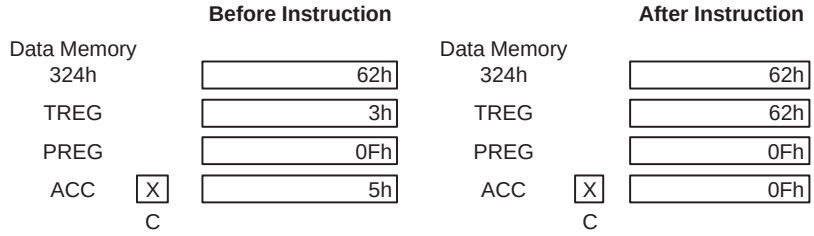
Cycles for a Repeat (RPT) Execution of an LTP Instruction

	Program			
Operand	**ROM**	**DARAM**	**SARAM**	**External**
DARAM	n	n	n	n+p
SARAM	n	n	n, n+1†	n+p
External	n+nd	n+nd	n+nd	n+1+p+nd

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

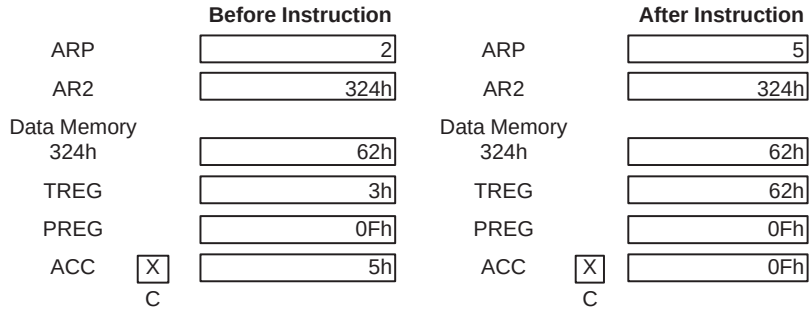
Example 1

```
LTP     36      ;(DP = 6: addresses 0300h–037Fh,
                ;PM = 0: no shift of product)
```



Example 2

```
LTP     *,AR5   ;(PM = 0)
```



Syntax

LTS *dma* Direct addressing
LTS *ind* [, **AR***n*] Indirect addressing

Operands

dma: 7 LSBs of the data-memory address
n: Value from 0 to 7 designating the next auxiliary register
ind: Select one of the following seven options:
* *+ *– *0+ *0– *BR0+ *BR0–

Opcode

LTS *dma*

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

LTS *ind* [, **AR***n*]

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

Note: ARU, N, and NAR are defined in section 6.3, *Indirect Addressing Mode* (page 6-9).

Execution

Increment PC, then ...
(data-memory address) → TREG
ACC – shifted (PREG) → ACC

Status Bits

Affected by	*Affects*
PM and OVM	C and OV

Description

TREG 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.

The carry bit is cleared (C = 0) if the result of the subtraction generates a borrow, and is set (C = 1) if it does not generate a borrow.

Words

1

Cycles for a Single LTS Instruction

	Program			
Operand	**ROM**	**DARAM**	**SARAM**	**External**
DARAM	1	1	1	1+p
SARAM	1	1	1, 2†	1+p
External	1+d	1+d	1+d	2+d+p

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

Cycles for a Repeat (RPT) Execution of an LTS Instruction

	Program			
Operand	**ROM**	**DARAM**	**SARAM**	**External**
DARAM	n	n	n	n+p
SARAM	n	n	n, n+1†	n+p
External	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: addresses 0300h–037Fh,
                  ;PM = 0: no shift of product)
```

	Before Instruction	After Instruction
Data Memory 324h	62h	62h
TREG	3h	62h
PREG	0Fh	0Fh
ACC	05h	0FFFFFFF6h
C	X	0

Example 2

```
LTS     *,AR2     ;(PM = 0)
```

	Before Instruction	After Instruction
ARP	1	2
AR1	324h	324h
324h	62h	62h
TREG	3h	62h
PREG	0Fh	0Fh
ACC	05h	0FFFFFFF6h
C	X	0

Syntax

MAC *pma*, *dma* Direct addressing
MAC *pma*, *ind* [, **AR***n*] Indirect addressing

Operands

dma: 7 LSBs of the data-memory address
pma: 16-bit program-memory address
n: Value from 0 to 7 designating the next auxiliary register
ind: Select one of the following seven options:
* *+ *– *0+ *0– *BR0+ *BR0–

Opcode

MAC *pma*, *dma*

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	dma						
pma															

MAC *pma*, *ind* [, **AR***n*]

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	ARU			N	NAR		
pma															

Note: ARU, N, and NAR are defined in section 6.3, *Indirect Addressing Mode* (page 6-9).

Execution

Increment PC, then . . .
(PC) → MSTACK
pma → PC
(ACC) + shifted (PREG) → ACC
(data-memory address) → TREG
(data-memory address) × (pma) → PREG
For indirect, modify (current AR) and (ARP) as specified
(PC) + 1 → PC

While (repeat counter) ≠ 0:
(ACC) + shifted (PREG) → ACC
(data-memory address) → TREG
(data-memory address) × (pma) → PREG
For indirect, modify (current AR) and (ARP) as specified
(PC) + 1 → PC
(repeat counter) – 1 → repeat counter

(MSTACK) → PC

Status Bits

Affected by	*Affects*
PM and OVM	C and OV

Description

The MAC instruction:

- ❏ Adds the previous product, shifted as defined by the PM status bits, to the accumulator. The carry bit is set (C = 1) if the result of the addition generates a carry and is cleared (C = 0) if it does not generate a carry.
- ❏ Loads the TREG with the content of the specified data-memory address.
- ❏ Multiplies the data-memory value in the TREG by the contents of the specified program-memory address.

The data and program memory locations on the ’C20x may be any nonreserved on-chip or off-chip memory locations. If the program memory is block B0 of on-chip RAM, the CNF bit must be set to 1.

When the MAC instruction is repeated, the program-memory address contained in the PC is incremented by 1 during each repetition. This makes it possible to access a series of operands in program memory. If you use indirect addressing to specify the data-memory address, a new data-memory address can be accessed during each repetition. If you use the direct addressing mode, the specified data-memory address is a constant; it will not be modified during each repetition.

MAC is useful for long sum-of-products operations because, when repeated, it becomes a single-cycle instruction once the RPT pipeline is started.

Words

2

Cycles

Cycles for a Single MAC Instruction

Operand	ROM	DARAM	SARAM	External
Operand 1: DARAM/ ROM Operand 2: DARAM	3	3	3	$3+2p_{code}$
Operand 1: SARAM Operand 2: DARAM	3	3	3	$3+2p_{code}$
Operand 1: External Operand 2: DARAM	$3+p_{op1}$	$3+p_{op1}$	$3+p_{op1}$	$3+p_{op1}+2p_{code}$
Operand 1: DARAM/ ROM Operand 2: SARAM	3	3	3	$3+2p_{code}$
Operand 1: SARAM Operand 2: SARAM	3 4†	3 4†	3 4†	$3+2p_{code}$ $4+2p_{code}$†
Operand 1: External Operand 2: SARAM	$3+p_{op1}$	$3+p_{op1}$	$3+p_{op1}$	$3+p_{op1}+2p_{code}$
Operand 1: DARAM/ ROM Operand 2: External	$3+d_{op2}$	$3+d_{op2}$	$3+d_{op2}$	$3+d_{op2}+2p_{code}$
Operand 1: SARAM Operand 2: External	$3+d_{op2}$	$3+d_{op2}$	$3+d_{op2}$	$3+d_{op2}+2p_{code}$
Operand 1: External Operand 2: External	$4+p_{op1}+d_{op2}$	$4+p_{op1}+d_{op2}$	$4+p_{op1}+d_{op2}$	$4+p_{op1}+d_{op2}+2p_{code}$

† If both operands are in the same SARAM block

Cycles for a Repeat (RPT) Execution of an MAC Instruction

Operand	ROM	DARAM	SARAM	External
Operand 1: DARAM/ ROM Operand 2: DARAM	n+2	n+2	n+2	$n+2+2p_{code}$
Operand 1: SARAM Operand 2: DARAM	n+2	n+2	n+2	$n+2+2p_{code}$
Operand 1: External Operand 2: DARAM	$n+2+np_{op1}$	$n+2+np_{op1}$	$n+2+np_{op1}$	$n+2+np_{op1}+2p_{code}$

† If both operands are in the same SARAM block

Cycles for a Repeat (RPT) Execution of an MAC Instruction (Continued)

Operand	ROM	DARAM	SARAM	External
Operand 1: DARAM/ ROM Operand 2: SARAM	n+2	n+2	n+2	n+2+2p_{code}
Operand 1: SARAM Operand 2: SARAM	n+2 2n+2†	n+2 2n+2†	n+2 2n+2†	n+2+2p_{code} 2n+2†
Operand 1: External Operand 2: SARAM	n+2+np_{op1}	n+2+np_{op1}	n+2+np_{op1}	n+2+np_{op1}+2p_{code}
Operand 1: DARAM/ ROM Operand 2: External	n+2+nd_{op2}	n+2+nd_{op2}	n+2+nd_{op2}	n+2+nd_{op2}+2p_{code}
Operand 1: SARAM Operand 2: External	n+2+nd_{op2}	n+2+nd_{op2}	n+2+nd_{op2}	n+2+nd_{op2}+2p_{code}
Operand 1: External Operand 2: External	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
TREG	45h	23h
PREG	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
TREG	45h	23h
PREG	458972h	8Ch
ACC	723EC41h	76975B3h
C	X	0

Syntax

MACD *pma*, *dma* Direct addressing
MACD *pma*, *ind* [, **AR***n*] Indirect addressing

Operands

dma: 7 LSBs of the data-memory address
pma: 16-bit program-memory address
n: Value from 0 to 7 designating the next auxiliary register
ind: Select one of the following seven options:
* *+ *– *0+ *0– *BR0+ *BR0–

Opcode

MACD *pma, dma*

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	dma						
pma															

MACD *pma, ind* [, **AR***n*]

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	ARU			N	NAR		
pma															

Note: ARU, N, and NAR are defined in section 6.3, *Indirect Addressing Mode* (page 6-9).

Execution

Increment PC, then . . .
(PC) → MSTACK
pma → PC
(ACC) + shifted (PREG) → ACC
(data-memory address) → TREG
(data-memory address) × (pma) → PREG
For indirect, modify (current AR) and (ARP) as specified
(PC) + 1 → PC
(data-memory address) → data-memory address + 1

While (repeat counter) ≠ 0:
(ACC) + shifted (PREG) → ACC
(data-memory address) → TREG
(data-memory address) × (pma) → PREG
For indirect, modify (current AR) and (ARP) as specified
(PC) + 1 → PC
(data-memory address) → data-memory address + 1
(repeat counter) – 1 → repeat counter

(MSTACK) → PC

Status Bits

Affected by	*Affects*
PM and OVM	C and OV

Description

The MACD instruction:

- ❑ Adds the previous product, shifted as defined by the PM status bits, to the accumulator. The carry bit is set (C = 1) if the result of the addition generates a carry and is cleared (C = 0) if it does not generate a carry.
- ❑ Loads the TREG with the content of the specified data-memory address.
- ❑ Multiplies the data-memory value in the TREG by the contents of the specified program-memory address.
- ❑ Copies the contents of the specified data-memory address to the next higher data-memory address.

The data- and program-memory locations on the 'C20x may be any nonreserved, on-chip or off-chip memory locations. If the program memory is block B0 of on-chip RAM, the CNF bit must be set to 1. If MACD addresses one of the memory-mapped registers or external memory as a data-memory location, the effect of the instruction is that of a MAC instruction; the data move will not occur (see the DMOV instruction description).

When the MACD instruction is repeated, the program-memory address contained in the PC is incremented by 1 during each repetition. This makes it possible to access a series of operands in program memory. If you use indirect addressing to specify the data-memory address, a new data-memory address can be accessed during each repetition. If you use the direct addressing mode, the specified data-memory address is a constant; it will not be modified during each repetition.

MACD functions in the same manner as MAC, with the addition of a data move for on-chip RAM blocks. This feature makes MACD useful for applications such as convolution and transversal filtering. When used with RPT, MACD becomes a single-cycle instruction once the RPT pipeline is started.

Words 2

Cycles

Cycles for a Single MACD Instruction

Operand	ROM	DARAM	SARAM	External
Operand 1: DARAM/ ROM Operand 2: DARAM	3	3	3	$3+2p_{code}$
Operand 1: SARAM Operand 2: DARAM	3	3	3	$3+2p_{code}$
Operand 1: External Operand 2: DARAM	$3+p_{op1}$	$3+p_{op1}$	$3+p_{op1}$	$3+p_{op1}+2p_{code}$
Operand 1: DARAM/ ROM Operand 2: SARAM	3	3	3	$3+2p_{code}$
Operand 1: SARAM Operand 2: SARAM	3	3	3 4† 5‡	$3+2p_{code}$ $4+2p_{code}$†
Operand 1: External Operand 2: SARAM	$3+p_{op1}$	$3+p_{op1}$	$3+p_{op1}$	$3+p_{op1}+2p_{code}$
Operand 1: DARAM/ ROM Operand 2: External§	$3+d_{op2}$	$3+d_{op2}$	$3+d_{op2}$	$3+d_{op2}+2p_{code}$
Operand 1: SARAM Operand 2: External§	$3+d_{op2}$	$3+d_{op2}$	$3+d_{op2}$	$3+d_{op2}+2p_{code}$
Operand 1: External Operand 2: External§	$4+p_{op1}+d_{op2}$	$4+p_{op1}+d_{op2}$	$4+p_{op1}+d_{op2}$	$4+p_{op1}+d_{op2}+2p_{code}$

† 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.

Cycles for a Repeat (RPT) Execution of an MACD Instruction

Operand	ROM	DARAM	SARAM	External
Operand 1: DARAM/ ROM Operand 2: DARAM	n+2	n+2	n+2	$n+2+2p_{code}$
Operand 1: SARAM Operand 2: DARAM	n+2	n+2	n+2	$n+2+2p_{code}$

† If operand 2 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.

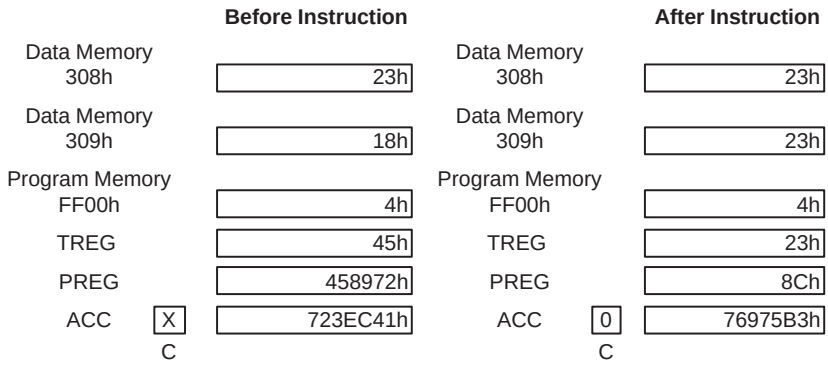
Cycles for a Repeat (RPT) Execution of an MACD Instruction (Continued)

Operand	ROM	DARAM	SARAM	External
Operand 1: External Operand 2: DARAM	n+2+np$_{op1}$	n+2+np$_{op1}$	n+2+np$_{op1}$	n+2+np$_{op1}$+2p$_{code}$
Operand 1: DARAM/ ROM Operand 2: SARAM	2n	2n	2n 2n+2†	2n+2p$_{code}$
Operand 1: SARAM Operand 2: SARAM	2n 3n‡	2n 3n‡	2n 2n+2† 3n‡ 3n+2§	2n+2p$_{code}$ 3n‡
Operand 1: External Operand 2: SARAM	2n+np$_{op1}$	2n+np$_{op1}$	2n+np$_{op1}$ 2n+2+np$_{op1}$†	2n+np$_{op1}$+2p$_{code}$
Operand 1: DARAM/ ROM Operand 2: External¶	n+2+nd$_{op2}$	n+2+nd$_{op2}$	n+2+nd$_{op2}$	n+2+nd$_{op2}$+2p$_{code}$
Operand 1: SARAM Operand 2: External¶	n+2+nd$_{op2}$	n+2+nd$_{op2}$	n+2+nd$_{op2}$	n+2+nd$_{op2}$+2p$_{code}$
Operand 1: External Operand 2: External¶	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 operand 2 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: addresses 0300h–037Fh,
                    ;PM = 0: no shift of product,
                    ;CNF = 1: RAM B0 configured to
                    ;program memory).
```



Example 2

```
MACD    0FF00h,*,AR6        ;(PM = 0, CNF = 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
TREG	45h	23h
PREG	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

MAR *dma* — Direct addressing
MAR *ind* [, **AR***n*] — Indirect addressing

Operands

n: Value from 0 to 7 designating the next auxiliary register
ind: Select one of the following seven options:
* *+ *– *0+ *0– *BR0+ *BR0–

Opcode

MAR *dma*

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	dma						

MAR *ind* [, **AR***n*]

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

Note: ARU, N, and NAR are defined in section 6.3, *Indirect Addressing Mode* (page 6-9).

Execution

Event(s)	*Addressing mode*
Increment PC	Direct
Increment PC Modify (current AR) and (ARP) as specified	Indirect

Status Bits

Affects	*Addressing mode*
None	Direct
ARP and ARB	Indirect

Description

In the direct addressing mode, the MAR instruction acts as a NOP instruction.

In the indirect addressing mode, an auxiliary register value and the ARP value can be modified; however, the memory being referenced is not used. When MAR modifies the ARP value, the old ARP value is copied to the ARB field of ST1. Any operation that MAR performs with indirect addressing can also be performed with any instruction that supports indirect addressing. The ARP can also be loaded by an LST instruction.

The LARP instruction from the ’C25 instruction set is a subset of MAR. For example, MAR *, AR4 performs the same function as LARP 4, which loads the ARP with 4.

For loading an auxiliary register, see the description for the LAR instruction. For storing an auxiliary register value to data memory, see the SAR instruction.

Words 1

Cycles

Cycles for a Single MAR Instruction

ROM	DARAM	SARAM	External
1	1	1	1+p

Cycles for a Repeat (RPT) Execution of an MAR Instruction

ROM	DARAM	SARAM	External
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	ARB	1

Syntax

MPY *dma*	Direct addressing
MPY *ind* [, **AR***n*]	Indirect addressing
MPY #*k*	Short immediate addressing

Operands

dma:	7 LSBs of the data-memory address
n:	Value from 0 to 7 designating the next auxiliary register
k:	13-bit short immediate value
ind:	Select one of the following seven options: * *+ *– *0+ *0– *BR0+ *BR0–

Opcode

MPY *dma*

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

MPY *ind* [, **AR***n*]

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

Note: ARU, N, and NAR are defined in section 6.3, *Indirect Addressing Mode* (page 6-9).

MPY #*k*

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

Execution

Increment PC, then ...

Event	*Addressing mode*
(TREG) × (data-memory address) → PREG	Direct or indirect
(TREG) × k → PREG	Short immediate

Status Bits

None

Description

The contents of TREG are multiplied by the contents of the addressed data memory location. The result is placed in the product register (PREG). With short immediate addressing, TREG is multiplied by a signed 13-bit constant. The short-immediate value is right justified and sign extended before the multiplication, regardless of SXM.

Words

1

Cycles

Cycles for a Single MPY Instruction (Using Direct and Indirect Addressing)

	Program			
Operand	**ROM**	**DARAM**	**SARAM**	**External**
DARAM	1	1	1	1+p
SARAM	1	1	1, 2†	1+p
External	1+d	1+d	1+d	2+d+p

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

Cycles for a Repeat (RPT) Execution of an MPY Instruction (Using Direct and Indirect Addressing)

	Program			
Operand	**ROM**	**DARAM**	**SARAM**	**External**
DARAM	n	n	n	n+p
SARAM	n	n	n, n+1†	n+p
External	n+nd	n+nd	n+nd	n+1+p+nd

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

Cycles for a Single MPY Instruction (Using Short Immediate Addressing)

ROM	**DARAM**	**SARAM**	**External**
1	1	1	1+p

Example 1

```
MPY     DAT13     ;(DP = 8)
```

	Before Instruction		**After Instruction**
Data Memory 40Dh	7h	Data Memory 40Dh	7h
TREG	6h	TREG	6h
PREG	36h	PREG	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
TREG	6h	TREG	6h
PREG	36h	PREG	2Ah

Example 3 `MPY #031h`

	Before Instruction		After Instruction
TREG	2h	TREG	2h
PREG	36h	PREG	62h

Syntax

MPYA *dma* Direct addressing
MPYA *ind* [, **AR***n*] Indirect addressing

Operands

dma: 7 LSBs of the data-memory address
n: Value from 0 to 7 designating the next auxiliary register
ind: Select one of the following seven options:
* *+ *– *0+ *0– *BR0+ *BR0–

Opcode

MPYA *dma*

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

MPYA *ind* [, **AR***n*]

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

Note: ARU, N, and NAR are defined in section 6.3, *Indirect Addressing Mode* (page 6-9).

Execution

Increment PC, then ...
(ACC) + shifted (PREG) → ACC
(TREG) × (data-memory address) → PREG

Status Bits

Affected by	*Affects*
PM and OVM	C and OV

Description

The contents of TREG are multiplied by the contents of the addressed data memory location. The result is placed in the product register (PREG). The previous product, shifted as defined by the PM status bits, is also added to the accumulator.

Words

1

Cycles

Cycles for a Single MPYA Instruction

	Program			
Operand	**ROM**	**DARAM**	**SARAM**	**External**
DARAM	1	1	1	1+p
SARAM	1	1	1, 2†	1+p
External	1+d	1+d	1+d	2+d+p

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

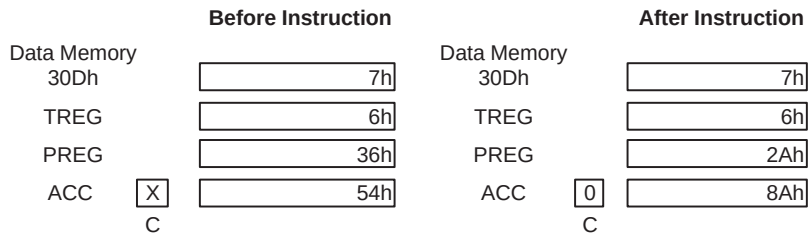
Cycles for a Repeat (RPT) Execution of an MPYA Instruction

	Program			
Operand	**ROM**	**DARAM**	**SARAM**	**External**
DARAM	n	n	n	n+p
SARAM	n	n	n, n+1†	n+p
External	n+nd	n+nd	n+nd	n+1+p+nd

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

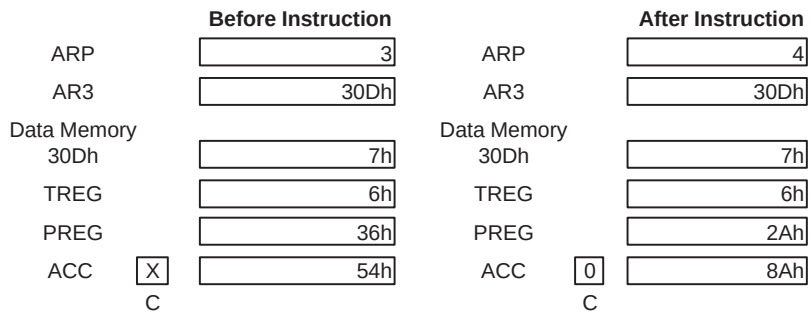
Example 1

```
MPYA    DAT13    ;(DP = 6, PM = 0)
```



Example 2

```
MPYA    *,AR4    ;(PM = 0)
```



Syntax

MPYS *dma* Direct addressing
MPYS *ind* [, **AR***n*] Indirect addressing

Operands

dma: 7 LSBs of the data-memory address
n: Value from 0 to 7 designating the next auxiliary register
ind: Select one of the following seven options:
* *+ *– *0+ *0– *BR0+ *BR0–

Opcode

MPYS *dma*

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

MPYS *ind* [, **AR***n*]

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

Note: ARU, N, and NAR are defined in section 6.3, *Indirect Addressing Mode* (page 6-9).

Execution

Increment PC, then ...
(ACC) – shifted (PREG) → ACC
(TREG) × (data-memory address) → PREG

Status Bits

Affected by	*Affects*
PM and OVM	C and OV

Description

The contents of TREG are multiplied by the contents of the addressed data memory location. The result is placed in the product register (PREG). 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

Cycles for a Single MPYS Instruction

	Program			
Operand	**ROM**	**DARAM**	**SARAM**	**External**
DARAM	1	1	1	1+p
SARAM	1	1	1, 2†	1+p
External	1+d	1+d	1+d	2+d+p

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

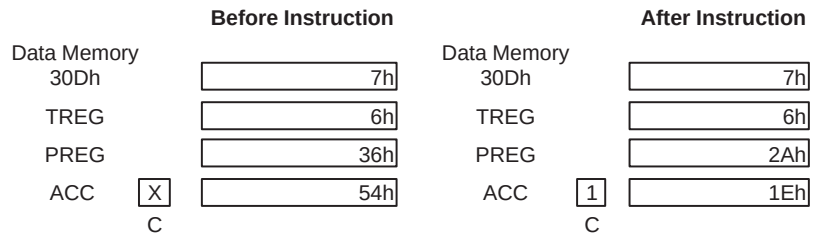
Cycles for a Repeat (RPT) Execution of an MPYS Instruction

	Program			
Operand	**ROM**	**DARAM**	**SARAM**	**External**
DARAM	n	n	n	n+p
SARAM	n	n	n, n+1†	n+p
External	n+nd	n+nd	n+nd	n+1+p+nd

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

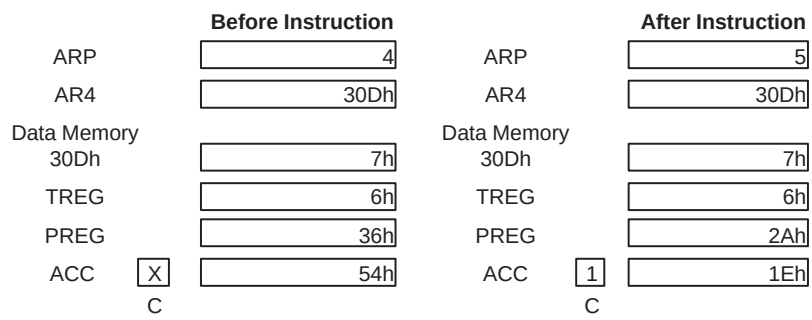
Example 1

```
MPYS    DAT13    ;(DP = 6, PM = 0)
```



Example 2

```
MPYS    *,AR5    ;(PM = 0)
```



Syntax

MPYU *dma* Direct addressing
MPYU *ind* [, **AR***n*] Indirect addressing

Operands

dma: 7 LSBs of the data-memory address
n: Value from 0 to 7 designating the next auxiliary register
ind: Select one of the following seven options:
* *+ *– *0+ *0– *BR0+ *BR0–

Opcode

MPYU *dma*

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

MPYU *ind* [,**AR***n*]

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

Note: ARU, N, and NAR are defined in section 6.3, *Indirect Addressing Mode* (page 6-9).

Execution

Increment PC, then ...
Unsigned (TREG) × unsigned (data-memory address) → PREG

Status Bits

None

This instruction is not affected by SXM.

Description

The unsigned contents of TREG are multiplied by the unsigned contents of the addressed data-memory location. The result is placed in the product register (PREG). The multiplier acts as a signed 17 × 17-bit multiplier for this instruction, with the MSB of both operands forced to 0.

When another instruction passes the resulting PREG value to data memory or to the CALU, the value passes first through the product shifter at the output of the PREG. This shifter always invokes sign extension on the PREG value 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

Cycles for a Single MPYU Instruction

	Program			
Operand	**ROM**	**DARAM**	**SARAM**	**External**
DARAM	1	1	1	1+p
SARAM	1	1	1, 2†	1+p
External	1+d	1+d	1+d	2+d+p

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

Cycles for a Repeat (RPT) Execution of an MPYU Instruction

	Program			
Operand	**ROM**	**DARAM**	**SARAM**	**External**
DARAM	n	n	n	n+p
SARAM	n	n	n, n+1†	n+p
External	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    16      ;(DP = 4: addresses 0200h–027Fh)
```

	Before Instruction	After Instruction
Data Memory 210h	0FFFFh	0FFFFh
TREG	0FFFFh	0FFFFh
PREG	1h	0FFFE0001h

Example 2

```
MPYU    *,AR6
```

	Before Instruction	After Instruction
ARP	5	6
AR5	210h	210h
Data Memory 210h	0FFFFh	0FFFFh
TREG	0FFFFh	0FFFFh
PREG	1h	0FFFE0001h

Syntax **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 Increment PC, then ...
(ACC) × –1 → ACC

Status Bits

Affected by	*Affects*
OVM	C and OV

Description The content of the accumulator is replaced with its arithmetic complement (2s complement). The OV bit is set when taking the NEG of 8000 0000h. If OVM = 1, the accumulator content is replaced with 7FFF FFFFh. If OVM = 0, the result is 8000 0000h. The carry bit (C) is cleared to 0 by this instruction for all nonzero values of the accumulator, and is set to 1 if the accumulator equals zero.

Words 1

Cycles

Cycles for a Single NEG Instruction

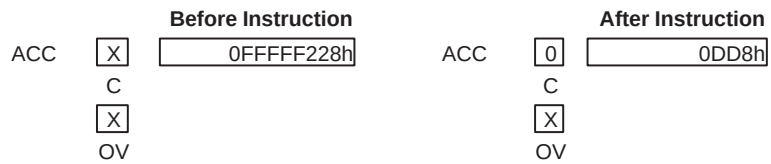
ROM	DARAM	SARAM	External
1	1	1	1+p

Cycles for a Repeat (RPT) Execution of an NEG Instruction

ROM	DARAM	SARAM	External
n	n	n	n+p

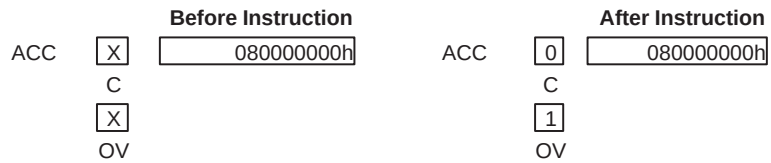
Example 1

```
NEG       ;(OVM = X) Convert –3544 to +3544
```



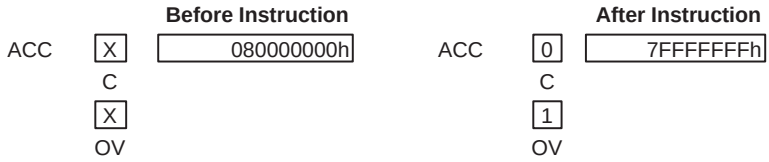
Example 2

```
NEG       ;(OVM = 0)
```



Example 3

```
NEG        ;(OVM = 1)
```



Syntax **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

Status Bits
Affects
INTM

This instruction is not affected by INTM.

Description The NMI instruction forces the program counter to the nonmaskable interrupt vector located at 24h. This instruction has the same effect as the hardware nonmaskable interrupt $\overline{\text{NMI}}$.

Words 1

Cycles

Cycles for a Single NMI Instruction

ROM	DARAM	SARAM	External
4	4	4	4+3p†

† The 'C20x performs speculative fetching by reading two additional instruction words. If the PC discontinuity is taken, these two instruction words are discarded.

Example

```
NMI   ;PC + 1 is pushed onto the stack, and then
      ;control is passed to program memory location
      ;24h.
```

Syntax **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 Increment PC

Status Bits None

Description No operation is performed. The NOP instruction affects only the PC. The NOP instruction is useful for creating pipeline and execution delays.

Words 1

Cycles

Cycles for a Single NOP Instruction

ROM	DARAM	SARAM	External
1	1	1	1+p

Cycles for a Repeat (RPT) Execution of an NOP Instruction

ROM	DARAM	SARAM	External
n	n	n	n+p

Example

```
NOP       ;No operation is performed.
```

Syntax **NORM** *ind* Indirect addressing

Operands ind: Select one of the following seven options:
* *+ *– *0+ *0– *BR0+ *BR0–

Opcode

NORM *ind*

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	ARU			N	NAR		

Note: ARU, N, and NAR are defined in section 6.3, *Indirect Addressing Mode* (page 6-9).

Execution

Increment PC, then ...

If (ACC) = 0:
  Then TC → 1;
Else, if (ACC(31)) XOR (ACC(30)) = 0:
  Then TC → 0,
    (ACC) × 2 → ACC
    Modify (current AR) as specified;
  Else TC → 1.

Status Bits

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 by finding the magnitude of the sign-extended number. An exclusive-OR operation is performed on accumulator bits 31 and 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 current AR is modified as specified to generate the magnitude of the exponent. It is assumed that the current AR is initialized before normalization begins. The default modification of the current AR 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. NORM functions on both positive and negative 2s-complement numbers.

Notes:

For the NORM instruction, the auxiliary register operations are executed during the fourth phase of the pipeline, the execution phase. For other instructions, the auxiliary register operations take place in the second phase of the pipeline, in the decode phase. Therefore:

1) **The auxiliary register values should not be modified by the two instruction words following NORM.** If the auxiliary register used in the NORM instruction is to be affected by either of the next two instruction words, the auxiliary register value will be modified by the other instructions *before* it is modified by the NORM instruction.

2) **The value in the auxiliary register pointer (ARP) should not be modified by the two instruction words following NORM.** If either of the next two instruction words specify a change in the ARP value, the ARP value will be changed *before* NORM is executed; the ARP will not be pointing at the correct auxiliary register when NORM is executed.

Words 1

Cycles

Cycles for a Single NORM Instruction

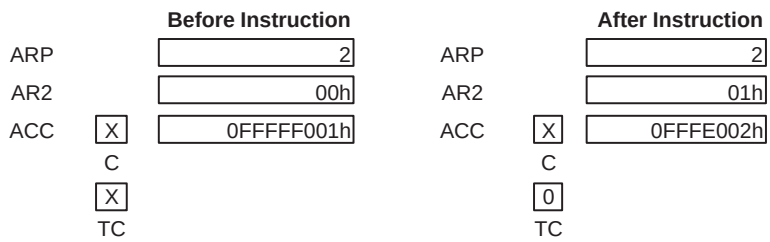
ROM	DARAM	SARAM	External
1	1	1	1+p

Cycles for a Repeat (RPT) Execution of a NORM Instruction

ROM	DARAM	SARAM	External
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 used in Example 2 normalizes a 32-bit number and yields a 5-bit exponent magnitude. The method used in Example 3 normalizes 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 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 fewer shifts. If the number requires six or more shifts, Example 3 is more efficient.

Syntax

OR *dma*	Direct addressing
OR *ind* [, **AR***n*]	Indirect addressing
OR #*lk* [, *shift*]	Long immediate addressing
OR #*lk*, **16**	Long immediate with left shift of 16

Operands

dma:	7 LSBs of the data-memory address
shift:	Left shift value from 0 to 15 (defaults to 0)
n:	Value from 0 to 7 designating the next auxiliary register
lk:	16-bit long immediate value
ind:	Select one of the following seven options: * *+ *– *0+ *0– *BR0+ *BR0–

Opcode

OR *dma*

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

OR *ind* [, **AR***n*]

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

Note: ARU, N, and NAR are defined in section 6.3, *Indirect Addressing Mode* (page 6-9).

OR #*lk* [, *shift*]

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	0	shift			
lk															

OR #*lk* [, **16**]

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	0
lk															

Execution

Increment PC, then ...

Event(s)	*Addressing mode*
(ACC(15:0)) OR (data-memory address) → ACC(15:0) (ACC(31:16)) → ACC(31:16)	Direct or indirect
(ACC) OR lk × 2^{shift} → ACC	Long immediate
(ACC) OR lk × 2^{16} → ACC	Long immediate with left shift of 16

Status Bits

None
This instruction is not affected by SXM.

Description

An OR operation is performed on the contents of the accumulator and the contents of the addressed data-memory location or a long-immediate value. The long-immediate value may be shifted before the OR operation. The result remains in the accumulator. All bit positions unoccupied by the data operand are zero filled, regardless of the value of the SXM status bit. 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 0. Zeros are shifted into the least significant bits of the operand if immediate addressing is used with a nonzero shift count.

Words

Words	*Addressing mode*
1	Direct or indirect
2	Long immediate

Cycles

Cycles for a Single OR Instruction (Using Direct and Indirect Addressing)

	Program			
Operand	**ROM**	**DARAM**	**SARAM**	**External**
DARAM	1	1	1	1+p
SARAM	1	1	1, 2†	1+p
External	1+d	1+d	1+d	2+d+p

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

Cycles for a Repeat (RPT) Execution of an OR Instruction (Using Direct and Indirect Addressing)

	Program			
Operand	**ROM**	**DARAM**	**SARAM**	**External**
DARAM	n	n	n	n+p
SARAM	n	n	n, n+1†	n+p
External	n+nd	n+nd	n+nd	n+1+p+nd

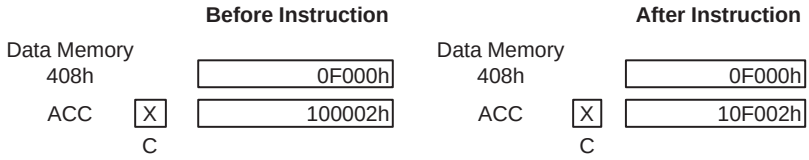
† If the operand and the code are in the same SARAM block

Cycles for a Single OR Instruction (Using Long Immediate Addressing)

ROM	**DARAM**	**SARAM**	**External**
2	2	2	2+2p

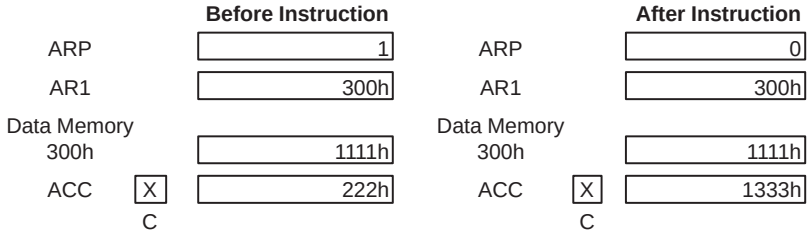
Example 1

```
OR DAT8     ;(DP = 8)
```



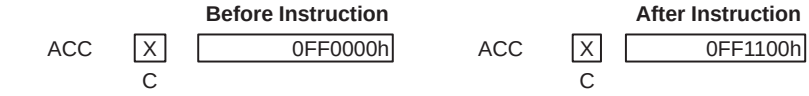
Example 2

```
OR *,AR0
```



Example 3

```
OR #08111h,8
```



Syntax

OUT *dma*, *PA* — Direct addressing
OUT *ind*, *PA* [, **AR***n*] — Indirect addressing

Operands

dma: 7 LSBs of the data-memory address
PA: 16-bit I/O address
n: Value from 0 to 7 designating the next auxiliary register
ind: Select one of the following seven options:
* *+ *– *0+ *0– *BR0+ *BR0–

Opcode

OUT *dma*, *PA*

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	dma						
PA															

OUT *ind*, *PA* [, **AR***n*]

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	ARU			N	NAR		
PA															

Note: ARU, N, and NAR are defined in section 6.3, *Indirect Addressing Mode* (page 6-9).

Execution

Increment PC, then ...
PA → address bus A15–A0
(data-memory address) → data bus D15–D0
(data-memory address) → PA

Status Bits

None

Description

The OUT instruction writes a 16-bit value from a data-memory location to the specified I/O location. The $\overline{IS}$ line goes low to indicate an I/O access. The $\overline{STRB}$, R/$\overline{W}$, and READY timings are the same as for an external data-memory write.

RPT can be used with the OUT instruction to write consecutive words from data memory to I/O space.

Words

2

Cycles

Cycles for a Single OUT Instruction

	Program			
Operand	**ROM**	**DARAM**	**SARAM**	**External**
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: External	$3+d_{src}+io_{dst}$	$3+d_{src}+io_{dst}$	$3+d_{src}+io_{dst}$	$6+d_{src}+io_{dst}+2p_{code}$

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

Cycles for a Repeat (RPT) Execution of an OUT Instruction

	Program			
Operand	**ROM**	**DARAM**	**SARAM**	**External**
Destination: DARAM	$3n+nio_{dst}$	$3n+nio_{dst}$	$3n+nio_{dst}$	$3n+3+nio_{dst}+2p_{code}$
Destination: SARAM	$3n+nio_{dst}$	$3n+nio_{dst}$	$3n+nio_{dst}$ $3n+1+nio_{dst}$ †	$3n+3+nio_{dst}+2p_{code}$
Destination: External	$5n–2+nd_{src}+nio_{dst}$	$5n–2+nd_{src}+nio_{dst}$	$5n–2+nd_{src}+nio_{dst}$	$5n+1+nd_{src}+nio_{dst}+2p_{code}$

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

Example 1

```
OUT   DAT0,100h     ;(DP = 4) Write data word stored in
                    ;data memory location 200h to
                    ;peripheral at I/O port address
                    ;100h.
```

Example 2

```
OUT   *,100h        ;Write data word referenced by
                    ;current auxiliary register to
                    ;peripheral at I/O port address
                    ;100h.
```

Syntax **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 Increment PC, then ...
shifted (PREG) → ACC

Status Bits *Affected by*
PM

Description The content of PREG, shifted as specified by the PM status bits, is loaded into the accumulator.

Words 1

Cycles

Cycles for a Single PAC Instruction

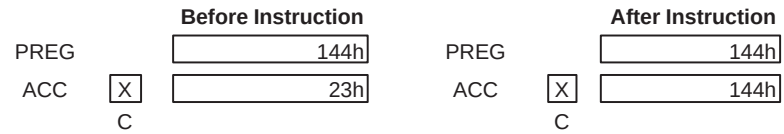
ROM	DARAM	SARAM	External
1	1	1	1+p

Cycles for a Repeat (RPT) Execution of a PAC Instruction

ROM	DARAM	SARAM	External
n	n	n	n+p

Example

```
PAC       ;(PM = 0: no shift of product)
```



Syntax **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

Increment PC, then ...
(TOS) → ACC(15:0)
0 → ACC(31:16)
Pop stack one level

Status Bits None

Description

The content of the top of the stack (TOS) is copied to the low accumulator, and then the stack values move up one level. The upper half of the accumulator is set to all zeros.

The hardware stack functions as a last-in, first-out stack 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 (using the POP, POPD, RETC, or RET instructions) occur before any pushes occur, all levels of the stack will contain the same value. No provision exists to check stack underflow.

Words 1

Cycles

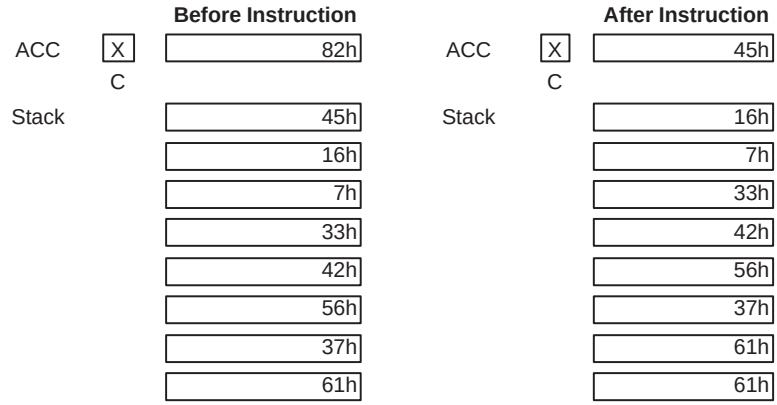
Cycles for a Single POP Instruction

ROM	DARAM	SARAM	External
1	1	1	1+p

Cycles for a Repeat (RPT) Execution of a POP Instruction

ROM	DARAM	SARAM	External
n	n	n	n+p

Example POP


Before Instruction
After Instruction
ACC
X
C
82h
45h
Stack
45h
16h
7h
33h
42h
56h
37h
61h
16h
7h
33h
42h
56h
37h
61h

Syntax

POPD *dma* Direct addressing
POPD *ind* [, **AR***n*] Indirect addressing

Operands

dma: 7 LSBs of the data-memory address
n: Value from 0 to 7 designating the next auxiliary register
ind: Select one of the following seven options:
* *+ *– *0+ *0– *BR0+ *BR0–

Opcode

POPD *dma*

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

POPD *ind* [,**AR***n*]

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

Note: ARU, N, and NAR are defined in section 6.3, *Indirect Addressing Mode* (page 6-9).

Execution

Increment PC, then ...
(TOS) → data-memory address
Pop stack one level

Status Bits

None

Description

The value from the top of the stack is transferred into the data-memory location specified by the instruction. In the lower seven locations of the stack, the values are copied up one level. The stack operation is explained in the description for the POP instruction. No provision exists to check stack underflow.

Words

1

Cycles

Cycles for a Single POPD Instruction

	Program			
Operand	**ROM**	**DARAM**	**SARAM**	**External**
DARAM	1	1	1	1+p
SARAM	1	1	1, 2†	1+p
External	2+d	2+d	2+d	4+d+p

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

Cycles for a Repeat (RPT) Execution of a POPD Instruction

	Program			
Operand	**ROM**	**DARAM**	**SARAM**	**External**
DARAM	n	n	n	n+p
SARAM	n	n	n, n+2†	n+p
External	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	92h
Stack	92h	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	1
AR0	300h	301h
Data Memory 300h	55h	92h
Stack	92h	72h
	72h	8h
	8h	44h
	44h	81h
	81h	75h
	75h	32h
	32h	0AAh
	0AAh	0AAh

Syntax

PSHD *dma* — Direct addressing
PSHD *ind* [, **AR***n*] — Indirect addressing

Operands

dma: 7 LSBs of the data-memory address
n: Value from 0 to 7 designating the next auxiliary register
ind: Select one of the following seven options:
* *+ *– *0+ *0– *BR0+ *BR0–

Opcode

PSHD *dma*

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

PSHD *ind* [, **AR***n*]

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

Note: ARU, N, and NAR are defined in section 6.3, *Indirect Addressing Mode* (page 6-9).

Execution

Increment PC, then ...
(data-memory address) → TOS
Push all stack locations down one level

Status Bits

None

Description

The value from the data-memory location specified by the instruction is transferred to the top of the stack. In the lower seven locations of the stack, the values are also copied one level down, as explained in the description for the PUSH instruction. The value in the lowest stack location is lost.

Words

1

Cycles

Cycles for a Single PSHD Instruction

	Program			
Operand	**ROM**	**DARAM**	**SARAM**	**External**
DARAM	1	1	1	1+p
SARAM	1	1	1, 2†	1+p
External	1+d	1+d	1+d	2+d+p

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

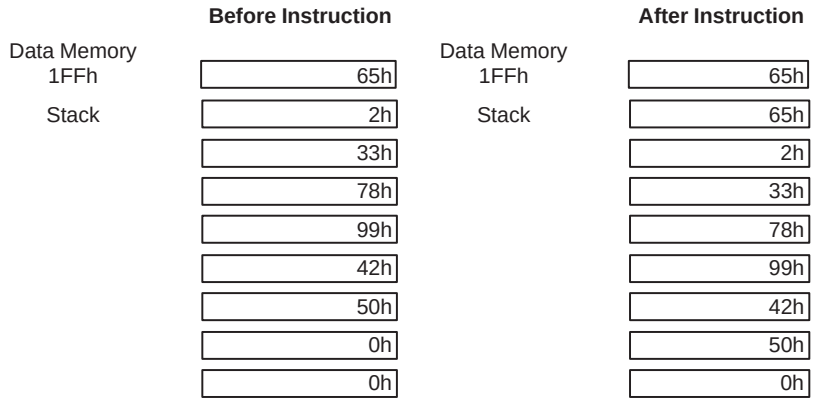
Cycles for a Repeat (RPT) Execution of a PSHD Instruction

	Program			
Operand	**ROM**	**DARAM**	**SARAM**	**External**
DARAM	n	n	n	n+p
SARAM	n	n	n, n+1†	n+p
External	n+nd	n+nd	n+nd	n+1+nd+p

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

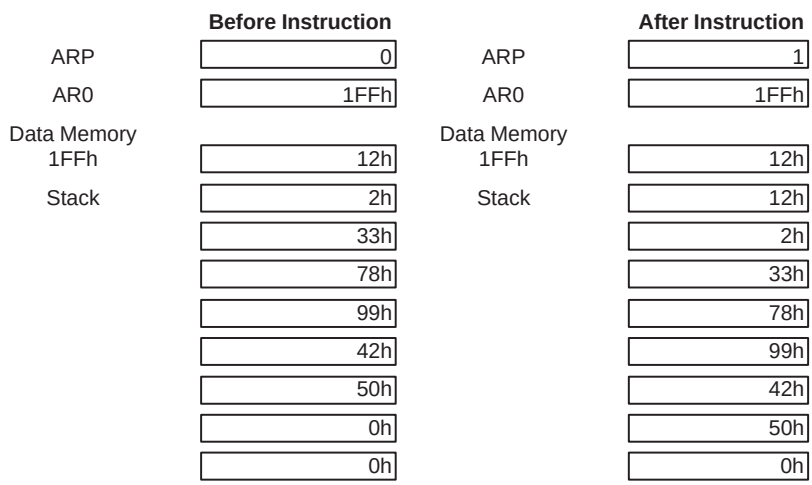
Example 1

```
PSHD    127     ;(DP = 3: addresses 0180–01FFh)
```



Example 2

```
PSHD    *,AR1
```



Syntax **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

Increment PC, then...
Push all stack locations down one level
ACC(15:0) → TOS

Status Bits None

Description

The stack values move down one level. Then, the content of the lower half of the accumulator is copied onto the top of the hardware stack.

The hardware stack operates as a last-in, first-out stack with eight locations. If more than eight pushes (due to a CALA, CALL, CC, PSHD, PUSH, TRAP, INTR, or NMI instruction) occur before a pop, the first data values written are lost with each succeeding push.

Words 1

Cycles

Cycles for a Single PUSH Instruction

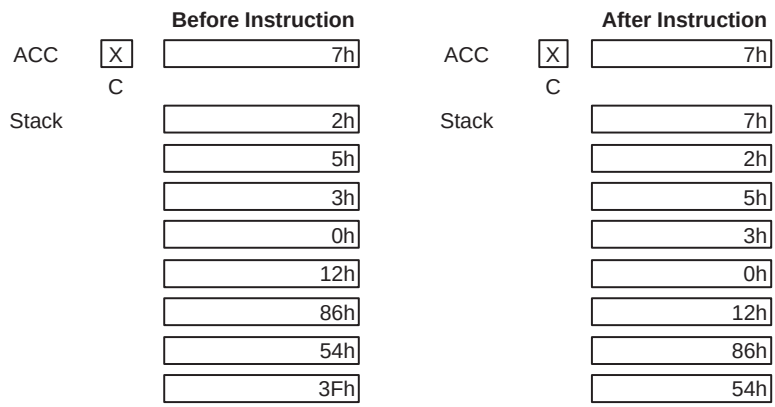
ROM	DARAM	SARAM	External
1	1	1	1+p

Cycles for a Repeat (RPT) Execution of a PUSH Instruction

ROM	DARAM	SARAM	External
n	n	n	n+p

Example

PUSH



Syntax **RET**

Operands None

Opcode

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

Execution (TOS) → PC
Pop stack one level.

Status Bits None

Description The contents of the top stack register are copied into the program counter. The remaining stack values are then copied up one level. RET concludes subroutines and interrupt service routines to return program control to the calling or interrupted program sequence.

Words 1

Cycles

Cycles for a Single RET Instruction

ROM	DARAM	SARAM	External
4	4	4	4+3p

Note: When this instruction reaches the execute phase of the pipeline, two additional instruction words have entered the pipeline. When the PC discontinuity is taken, these two instruction words are discarded.

Example 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

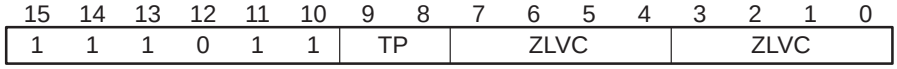
Syntax **RETC** *cond* 1 [, *cond* 2] [,...]

Operands

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

‡

Opcode



Note: The TP and ZLVC fields are defined on pages 7-3 and 7-4.

Execution

If *cond* 1 AND *cond* 2 AND ...
(TOS) → PC
Pop stack one level
Else, continue

Status Bits None

Description If the specified condition or conditions are met, a standard return is executed (see the description for the RET instruction). Note that not all combinations of conditions are meaningful. For example, testing for LT and GT is contradictory. In addition, testing $\overline{\text{BIO}}$ is mutually exclusive to testing TC.

Words 1

Cycles

Cycles for a Single RETC Instruction

Condition	ROM	DARAM	SARAM	External
True	4	4	4	4+4p
False	2	2	2	2+2p

Note: The processor performs speculative fetching by reading two additional instruction words. If the PC discontinuity is taken, these two instruction words are discarded.

Example

```
RETC      GEQ,NOV   ;A return is executed if the
                    ;accumulator content is positive
                    ;or zero and if the OV (overflow)
                    ;-bit is zero.
```

Syntax **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

Increment PC, then ...
C → ACC(0)
(ACC(31)) → C
(ACC(30:0)) → ACC(31:1)

Status Bits

Affects
C

This instruction is not affected by SXM.

Description The ROL instruction rotates the accumulator left one bit. The value of the carry bit is shifted into the LSB, then the MSB is shifted into the carry bit.

Words 1

Cycles

Cycles for a Single ROL Instruction

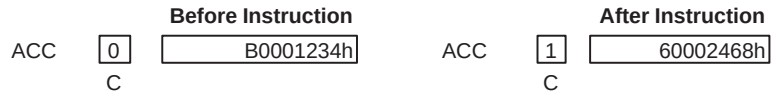
ROM	DARAM	SARAM	External
1	1	1	1+p

Cycles for a Repeat (RPT) Execution of an ROL Instruction

ROM	DARAM	SARAM	External
n	n	n	n+p

Example

```
ROL
```



Syntax **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

Increment PC, then ...
C → ACC(31)
(ACC(0)) → C
(ACC(31:1)) → ACC(30:0)

Status Bits

Affects
C

This instruction is not affected by SXM.

Description The ROR instruction rotates the accumulator right one bit. The value of the carry bit is shifted into the MSB of the accumulator, then the LSB of the accumulator is shifted into the carry bit.

Words 1

Cycles

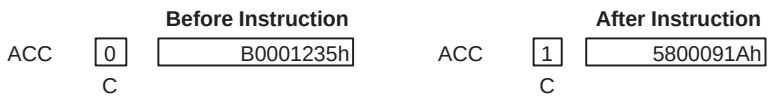
Cycles for a Single ROR Instruction

ROM	DARAM	SARAM	External
1	1	1	1+p

Cycles for a Repeat (RPT) Execution of an ROR Instruction

ROM	DARAM	SARAM	External
n	n	n	n+p

Example R0R



Syntax

RPT *dma*	Direct addressing
RPT *ind* [, **AR***n*]	Indirect addressing
RPT #*k*	Short immediate

Operands

dma:	7 LSBs of the data-memory address
n:	Value from 0 to 7 designating the next auxiliary register
k:	8-bit short immediate value
ind:	Select one of the following seven options: * *+ *– *0+ *0– *BR0+ *BR0–

Opcode

RPT *dma*

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

RPT *ind* [, **AR***n*]

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

Note: ARU, N, and NAR are defined in section 6.3, *Indirect Addressing Mode* (page 6-9).

RPT #*k*

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

Execution

Increment PC, then ...

Event	*Addressing mode*
(data-memory address) → RPTC	Direct or indirect
k → RPTC	Short immediate

Status Bits

None

Description

The repeat counter (RPTC) is loaded with the content of the addressed data-memory location if direct or indirect addressing is used; it is loaded with an 8-bit immediate value if short immediate addressing is used. The instruction following the RPT is repeated *n* times, where *n* is the initial value of the RPTC plus 1. Since the RPTC cannot be saved during a context switch, repeat loops are regarded as multicycle instructions and are not interruptible. The RPTC is cleared to 0 on a device reset.

RPT is especially useful for block moves, multiply/accumulates, and normalization. The repeat instruction itself is not repeatable.

Words

1

Cycles

Cycles for a Single RPT Instruction (Using Direct and Indirect Addressing)

	Program			
Operand	**ROM**	**DARAM**	**SARAM**	**External**
DARAM	1	1	1	1+p
SARAM	1	1	1, 2†	1+p
External	1+d	1+d	1+d	2+d+p

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

Cycles for a Single RPT Instruction (Using Short Immediate Addressing)

ROM	DARAM	SARAM	External
1	1	1	1+p

Example 1

```
RPT   DAT127    ;(DP = 31: addresses 0F80h–0FFFh)
                ;Repeat next instruction 13 times.
```

	Before Instruction		After Instruction
Data Memory 0FFFh	0Ch	Data Memory 0FFFh	0Ch
RPTC	0h	RPTC	0Ch

Example 2

```
RPT   *,AR1     ;Repeat next instruction 4096 times.
```

	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 two times.
```

	Before Instruction		After Instruction
RPTC	0h	RPTC	1h

Syntax

SACH *dma* [, *shift2*] Direct addressing
SACH *ind* [, *shift2* [, **AR***n*]] Indirect addressing

Operands

dma: 7 LSBs of the data-memory address
shift2: Left shift value from 0 to 7 (defaults to 0)
n: Value from 0 to 7 designating the next auxiliary register
ind: Select one of the following seven options:
* *+ *– *0+ *0– *BR0+ *BR0–

Opcode

SACH *dma* [, *shift2*]

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	1	shift2			0	dma						

SACH *ind* [, *shift2*[, **AR***n*]]

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	1	shift2			1	ARU			N	NAR		

Note: ARU, N, and NAR are defined in section 6.3, *Indirect Addressing Mode* (page 6-9).

Execution

Increment PC, then ...
16 MSBs of ((ACC) $\times$ 2^{shift2}) $\rightarrow$ data-memory address

Status Bits

This instruction is not affected by SXM

Description

The SACH instruction copies the entire accumulator into the output 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. During the shift, the low-order bits are filled with zeros, and the high-order bits are lost. The accumulator itself remains unaffected.

Words

1

Cycles

Cycles for a Single SACH Instruction

	Program			
Operand	**ROM**	**DARAM**	**SARAM**	**External**
DARAM	1	1	1	1+p
SARAM	1	1	1, 2†	1+p
External	2+d	2+d	2+d	4+d+p

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

Cycles for a Repeat (RPT) Execution of an SACH Instruction

	Program			
Operand	**ROM**	**DARAM**	**SARAM**	**External**
DARAM	n	n	n	n+p
SARAM	n	n	n, n+2†	n+p
External	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: addresses 0200h–027Fh,
                  ;left shift of 1)
```

	Before Instruction	After Instruction
ACC	4208001h	4208001h
C	X	X
Data Memory 20Ah	0h	0841h

Example 2

```
SACH    *+,0,AR2  ;(No shift)
```

	Before Instruction	After Instruction
ARP	1	2
AR1	300h	301h
ACC	4208001h	4208001h
C	X	X
Data Memory 300h	0h	0420h

Syntax

SACL *dma* [, *shift2*] Direct addressing
SACL ind [, *shift2* [, **AR***n*]] Indirect addressing

Operands

dma: 7 LSBs of the data-memory address
shift2: Left shift value from 0 to 7 (defaults to 0)
n: Value from 0 to 7 designating the next auxiliary register
ind: Select one of the following seven options:
* *+ *– *0+ *0– *BR0+ *BR0–

Opcode

SACL *dma* [, *shift2*]

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	0	shift2			0	dma						

SACL *ind* [, *shift2* [, **AR***n*]]

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	0	shift2			1	ARU			N	NAR		

Note: ARU, N, and NAR are defined in section 6.3, *Indirect Addressing Mode* (page 6-9).

Execution

Increment PC, then ...
16 LSBs of ((ACC) $\times$ 2^{shift2}) $\rightarrow$ data-memory address

Status Bits

This instruction is not affected by SXM.

Description

The SACL instruction copies the entire accumulator into the output shifter, where it left shifts the entire 32-bit number from 0 to 7 bits. It then copies the lower 16 bits of the shifted value into data memory. During the shift, the low-order bits are filled with zeros, and the high-order bits are lost. The accumulator itself remains unaffected.

Words

1

Cycles

Cycles for a Single SACL Instruction

	Program			
Operand	**ROM**	**DARAM**	**SARAM**	**External**
DARAM	1	1	1	1+p
SARAM	1	1	1, 2†	1+p
External	2+d	2+d	2+d	4+d+p

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

Cycles for a Repeat (RPT) Execution of an SACL Instruction

	Program			
Operand	**ROM**	**DARAM**	**SARAM**	**External**
DARAM	n	n	n	n+p
SARAM	n	n	n, n+2†	n+p
External	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: addresses 0200h–027Fh,
                  ;left shift of 1)
```

	Before Instruction	After Instruction
ACC	7C63 8421	7C63 8421h
C (X)		
Data Memory 20Bh	05h	0842h

Example 2

```
SACL    *,0,AR7   ;(No shift)
```

	Before Instruction	After Instruction
ARP	6	7
AR6	300h	300h
ACC	00FF 8421h	00FF 8421h
C (X)		
Data Memory 300h	05h	8421h

Syntax

SAR AR*x*, *dma* Direct addressing
SAR AR*x,* ind [, **AR***n*] Indirect addressing

Operands

dma: 7 LSBs of the data-memory address
x: Value from 0 to 7 designating the auxiliary register value to be stored
n: Value from 0 to 7 designating the next auxiliary register
ind: Select one of the following seven options:
* *+ *– *0+ *0– *BR0+ *BR0–

Opcode

SAR AR*x*, *dma*

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

SAR AR*x*, *ind* [*,* **AR***n*]

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

Note: ARU, N, and NAR are defined in section 6.3, *Indirect Addressing Mode* (page 6-9).

Execution

Increment PC, then ...
(ARx) → data-memory address

Status Bits

None

Description

The content of the designated auxiliary register (ARx) is stored in the specified data-memory location. When the content of the designated auxiliary register is also modified by the instruction (in indirect addressing mode), SAR copies the auxiliary register value to data memory before it increments or decrements the contents of the auxiliary register.

Words

1

Cycles

Cycles for a Single SAR Instruction

	Program			
Operand	**ROM**	**DARAM**	**SARAM**	**External**
DARAM	1	1	1	1+p
SARAM	1	1	1, 2†	1+p
External	2+d	2+d	2+d	4+d+p

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

Cycles for a Repeat (RPT) Execution of an SAR Instruction

	Program			
Operand	**ROM**	**DARAM**	**SARAM**	**External**
DARAM	n	n	n	n+p
SARAM	n	n	n, n+2†	n+p
External	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: addresses 0300h–037Fh)
```

	Before Instruction		After Instruction
AR0	37h	AR0	37h
Data Memory 31Eh	18h	Data Memory 31Eh	37h

Example 2

```
SAR     AR0,*+
```

	Before Instruction		After Instruction
ARP	0	ARP	0
AR0	401h	AR0	402h
Data Memory 401h	0h	Data Memory 401h	401h

Syntax **SBRK** #*k* Short immediate addressing

Operands k: 8-bit positive short immediate value

Opcode

SBRK #*k*

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

Execution

Increment PC, then ...
(current AR) – k → current AR

Note that k is an 8-bit positive constant.

Status Bits None

Description The 8-bit immediate value is subtracted, right justified, from the content of the current auxiliary register (the one pointed to by the ARP) and the result replaces the contents of the auxiliary register. The subtraction takes place in the auxiliary register arithmetic unit (ARAU), with the immediate value treated as an 8-bit positive integer. All arithmetic operations on the auxiliary registers are unsigned.

Words 1

Cycles

Cycles for a Single SBRK Instruction

ROM	DARAM	SARAM	External
1	1	1	1+p

Example

```
SBRK    #0FFh
```

	Before Instruction		After Instruction
ARP	7	ARP	7
AR7	0h	AR7	FF01h

Syntax

SETC *control bit*

Operands

control bit: Select one of the following control bits:

C	Carry bit of status register ST1
CNF	RAM configuration control bit of status register ST1
INTM	Interrupt mode bit of status register ST0
OVM	Overflow mode bit of status register ST0
SXM	Sign-extension mode bit of status register ST1
TC	Test/control flag bit of status register ST1
XF	XF pin status bit of status register ST1

Opcode

SETC 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

SETC CNF

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

SETC INTM

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

SETC 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

SETC 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

SETC TC

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

SETC XF

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

Execution

Increment PC, then ...
1 → control bit

Status Bits

None

Description

The specified control bit is set to 1. Note that LST may also be used to load ST0 and ST1. See section 4.5, *Status and Control Registers*, on page 4-15 for more information on each control bit.

Words 1

Cycles

Cycles for a Single SETC Instruction

ROM	DARAM	SARAM	External
1	1	1	1+p

Cycles for a Repeat (RPT) Execution of an SETC Instruction

ROM	DARAM	SARAM	External
n	n	n	n+p

Example

```
SETC    TC      ;TC is bit 11 of ST1
```

	Before Instruction		After Instruction
ST1	x1xxh	ST1	x9xxh

Syntax **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

Increment PC, then ...
(ACC(31)) → C
(ACC(30:0)) → ACC(31:1)
0 → ACC(0)

Status Bits

Affects
C

This instruction is not affected by SXM.

Description The SFL instruction shifts the entire accumulator left one bit. The least significant bit is filled with a 0, and the most significant bit is shifted into the carry bit (C). SFL, unlike SFR, is unaffected by SXM.

Words 1

Cycles

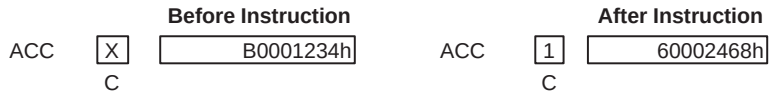
Cycles for a Single SFL Instruction

ROM	DARAM	SARAM	External
1	1	1	1+p

Cycles for a Repeat (RPT) Execution of an SFL Instruction

ROM	DARAM	SARAM	External
n	n	n	n+p

Example SFL



Syntax **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

Increment PC, then ...
If SXM = 0
  Then 0 → ACC(31).
If SXM = 1
  Then (ACC(31)) → ACC(31)

(ACC(31:1)) → ACC(30:0)
(ACC(0)) → C

Status Bits

Affected by	*Affects*
SXM	C

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 0.

Words 1

Cycles

Cycles for a Single SFR Instruction

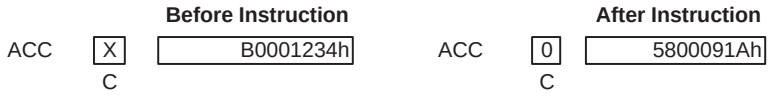
ROM	DARAM	SARAM	External
1	1	1	1+p

Cycles for a Repeat (RPT) Execution of an SFR Instruction

ROM	DARAM	SARAM	External
n	n	n	n+p

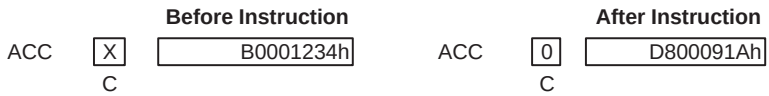
Example 1

```
SFR       ;(SXM = 0: no sign extension)
```



Example 2

```
SFR       ;(SXM = 1: sign extend)
```



Syntax **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 Increment PC, then ...
(ACC) – shifted (PREG) → ACC

Status Bits

Affected by	*Affects*
PM and OVM	C and OV

This instruction is not affected by SXM.

Description The content of PREG, shifted as defined by the PM status bits, is subtracted from the content of the accumulator. The result is stored in the accumulator. SPAC is not affected by SXM, and the PREG value is always sign extended.

The function of the SPAC instruction is a subtask of the LTS, MPYS, and SQRS instructions.

Words 1

Cycles

Cycles for a Single SPAC Instruction

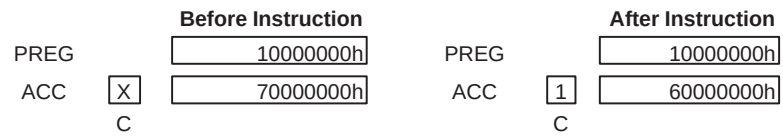
ROM	DARAM	SARAM	External
1	1	1	1+p

Cycles for a Repeat (RPT) Execution of an SPAC Instruction

ROM	DARAM	SARAM	External
n	n	n	n+p

Example

```
SPAC      ;(PM = 0)
```



Syntax

SPH *dma* Direct addressing
SPH *ind* [, **AR***n*] Indirect addressing

Operands

dma: 7 LSBs of the data-memory address
n: Value from 0 to 7 designating the next auxiliary register
ind: Select one of the following seven options:
* *+ *– *0+ *0– *BR0+ *BR0–

Opcode

SPH *dma*

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

SPH *ind* [, **AR***n*]

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

Note: ARU, N, and NAR are defined in section 6.3, *Indirect Addressing Mode* (page 6-9).

Execution

Increment PC, then ...
16 MSBs of shifted (PREG) → data-memory address

Status Bits

Affected by
PM

Description

The 16 high-order bits of the PREG, shifted as specified by the PM bits, are stored in data memory. First, the 32-bit PREG value is copied into the product shifter, where it is shifted as specified by the PM bits. If the right-shift-by-6 mode is selected, the high-order bits are sign extended and the low-order bits are lost. If a left shift is selected, the high-order bits are lost and the low-order bits are zero filled. If PM = 00, no shift occurs. Then the 16 MSBs of the shifted value are stored in data memory. Neither the PREG value nor the accumulator value is modified by this instruction.

Words

1

Cycles

Cycles for a Single SPH Instruction

	Program			
Operand	**ROM**	**DARAM**	**SARAM**	**External**
DARAM	1	1	1	1+p
SARAM	1	1	1, 2†	1+p
External	2+d	2+d	2+d	4+d+p

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

Cycles for a Repeat (RPT) Execution of an SPH Instruction

	Program			
Operand	**ROM**	**DARAM**	**SARAM**	**External**
DARAM	n	n	n	n+p
SARAM	n	n	n, n+2†	n+p
External	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: addresses 0200h–027Fh,
                ;PM = 0: no shift)
```

	Before Instruction		After Instruction
PREG	FE079844h	PREG	FE079844h
Data Memory 203h	4567h	Data Memory 203h	FE07h

Example 2

```
SPH     *,AR7   ;(PM = 2: left shift of four)
```

	Before Instruction		After Instruction
ARP	6	ARP	7
AR6	203h	AR6	203h
PREG	FE079844h	PREG	FE079844h
Data Memory 203h	4567h	Data Memory 203h	E079h

Syntax

SPL *dma* Direct addressing
SPL *ind* [, **AR***n*] Indirect addressing

Operands

dma: 7 LSBs of the data-memory address
n: Value from 0 to 7 designating the next auxiliary register
ind: Select one of the following seven options:
* *+ *– *0+ *0– *BR0+ *BR0–

Opcode

SPL *dma*

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

SPL *ind* [, **AR***n*]

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

Note: ARU, N, and NAR are defined in section 6.3, *Indirect Addressing Mode* (page 6-9).

Execution

Increment PC, then ...
16 LSBs of shifted (PREG) → data-memory address

Status Bits

Affected by
PM

Description

The 16 low-order bits of the PREG, shifted as specified by the PM bits, are stored in data memory. First, the 32-bit PREG value is copied into the product shifter, where it is shifted as specified by the PM bits. If the right-shift-by-6 mode is selected, the high-order bits are sign extended and the low-order bits are lost. If a left shift is selected, the high-order bits are lost and the low-order bits are zero filled. If PM = 00, no shift occurs. Then the 16 LSBs of the shifted value are stored in data memory. Neither the PREG value nor the accumulator value is modified by this instruction.

Words

1

Cycles

Cycles for a Single SPL Instruction

	Program			
Operand	**ROM**	**DARAM**	**SARAM**	**External**
DARAM	1	1	1	1+p
SARAM	1	1	1, 2†	1+p
External	2+d	2+d	2+d	4+d+p

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

Cycles for a Repeat (RPT) Execution of an SPL Instruction

	Program			
Operand	**ROM**	**DARAM**	**SARAM**	**External**
DARAM	n	n	n	n+p
SARAM	n	n	n, n+2†	n+p
External	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 = 4: addresses 0200h–027Fh,
                ;PM = 2: left shift of four)
```

	Before Instruction		After Instruction
PREG	0FE079844h	PREG	0FE079844h
Data Memory 205h	4567h	Data Memory 205h	08440h

Example 2

```
SPL     *,AR3   ;(PM = 0: no shift)
```

	Before Instruction		After Instruction
ARP	2	ARP	3
AR2	205h	AR2	205h
PREG	0FE079844h	PREG	0FE079844h
Data Memory 205h	4567h	Data Memory 205h	09844h

Syntax

SPLK #*lk*, *dma* Direct addressing
SPLK #*lk*, *ind* [, **AR***n*] Indirect addressing

Operands

dma: 7 LSBs of the data-memory address
n: Value from 0 to 7 designating the next auxiliary register
lk: 16-bit long immediate value
ind: Select one of the following seven options:
* *+ *– *0+ *0– *BR0+ *BR0–

Opcode

SPLK *#lk, dma*

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	dma						
lk															

SPLK *#lk, ind* [, **AR***n*]

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	ARU			N	NAR		
lk															

Note: ARU, N, and NAR are defined in section 6.3, *Indirect Addressing Mode* (page 6-9).

Execution

Increment PC, then ...
lk → data-memory address

Status Bits

None

Description

The SPLK instruction allows a full 16-bit pattern to be written into any data memory location.

Words

2

Cycles

Cycles for a Single SPLK Instruction

	Program			
Operand	**ROM**	**DARAM**	**SARAM**	**External**
DARAM	2	2	2	2+2p
SARAM	2	2	2, 3†	2+2p
External	3+d	3+d	3+d	5+d+2p

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

Example 1

```
SPLK    #7FFFh,DAT3          ;(DP = 6)
```

	Before Instruction		**After Instruction**
Data Memory 303h	FE07h	Data Memory 303h	7FFFh

Example 2

```
SPLK    #1111h,*+,AR4
```

	Before Instruction		After Instruction
ARP	0	ARP	4
AR0	300h	AR0	301h
Data Memory 300h	07h	Data Memory 300h	1111h

Syntax **SPM** *constant*

Operands constant: Value from 0 to 3 that determines the product shift mode

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	constant	

Execution Increment PC, then ...
constant → product shift mode (PM) bits

Status Bits *Affects*
PM

This instruction is not affected by SXM.

Description The two LSBs of the instruction word are copied into the product shift mode (PM) bits of status register ST1 (bits 1 and 0 of ST1). The PM bits control the mode of the shifter at the output of the PREG. This shifter can shift the PREG output either one or four bits to the left or six bits to the right. The possible PM bit combinations and their meanings are shown in Table 7–8. When an instruction accesses the PREG value, the value first passes through the shifter, where it is shifted by the specified amount.

Table 7–8. Product Shift Modes

PM Field	Specified Product Shift
00	No shift of PREG output
01	PREG output to be left shifted 1 place
10	PREG output to be left shifted 4 places
11	PREG output to be right shifted 6 places and sign extended

The left shifts allow the product to be justified for fractional arithmetic. The right-shift-by-six mode allows 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

Cycles for a Single SPM Instruction

ROM	DARAM	SARAM	External
1	1	1	1+p

Example

```
SPM   3     ;Product register shift mode 3 (PM = 11)
            ;is selected causing all subsequent
            ;transfers from the product register (PREG)
            ;to be shifted to the right six places.
```

Syntax

SQRA *dma* Direct addressing
SQRA *ind* [, **AR***n*] Indirect addressing

Operands

dma: 7 LSBs of the data-memory address
n: Value from 0 to 7 designating the next auxiliary register
ind: Select one of the following seven options:
* *+ *– *0+ *0– *BR0+ *BR0–

Opcode

SQRA *dma*

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

SQRA *ind* [, **AR***n*]

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

Note: ARU, N, and NAR are defined in section 6.3, *Indirect Addressing Mode* (page 6-9).

Execution

Increment PC, then ...
(ACC) + shifted (PREG) → ACC
(data-memory address) → TREG
(TREG) × (data-memory address) → PREG

Status Bits

Affected by	*Affects*
OVM and PM	OV and C

Description

The content of the PREG, shifted as defined by the PM status bits, is added to the accumulator. Then the addressed data-memory value is loaded into the TREG, squared, and stored in the PREG.

Words

1

Cycles

Cycles for a Single SQRA Instruction

	Program			
Operand	**ROM**	**DARAM**	**SARAM**	**External**
DARAM	1	1	1	1+p
SARAM	1	1	1, 2†	1+p
External	1+d	1+d	1+d	2+d+p

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

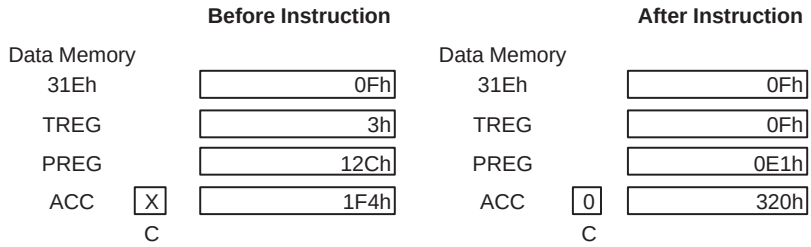
Cycles for a Repeat (RPT) Execution of an SQRA Instruction

	Program			
Operand	**ROM**	**DARAM**	**SARAM**	**External**
DARAM	n	n	n	n+p
SARAM	n	n	n, n+1†	n+p
External	n+nd	n+nd	n+nd	n+1+p+nd

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

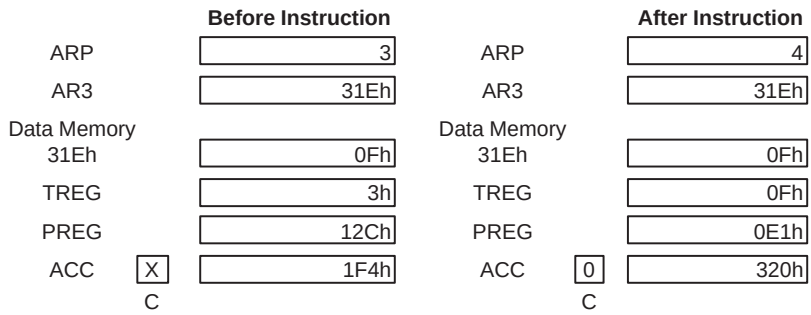
Example 1

```
SQRA    DAT30    ;(DP = 6: addresses 0300h–037Fh,
                 ;PM = 0: no shift of product)
```



Example 2

```
SQRA    *,AR4    ;(PM = 0)
```



Syntax

SQRS *dma* Direct addressing
SQRS *ind* [, **AR***n*] Indirect addressing

Operands

dma: 7 LSBs of the data-memory address
n: Value from 0 to 7 designating the next auxiliary register
ind: Select one of the following seven options:
* *+ *– *0+ *0– *BR0+ *BR0–

Opcode

SQRS *dma*

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

SQRS *ind* [, **AR***n*]

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

Note: ARU, N, and NAR are defined in section 6.3, *Indirect Addressing Mode* (page 6-9).

Execution

Increment PC, then ...
(ACC) – shifted (PREG) → ACC
(data-memory address) → TREG
(TREG) × (data-memory address) → PREG

Status Bits

Affected by	*Affects*
OVM and PM	OV and C

Description

The content of the PREG, shifted as defined by the PM status bits, is subtracted from the accumulator. Then the addressed data-memory value is loaded into the TREG, squared, and stored in the PREG.

Words

1

Cycles

Cycles for a Single SQRS Instruction

	Program			
Operand	**ROM**	**DARAM**	**SARAM**	**External**
DARAM	1	1	1	1+p
SARAM	1	1	1, 2†	1+p
External	1+d	1+d	1+d	2+d+p

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

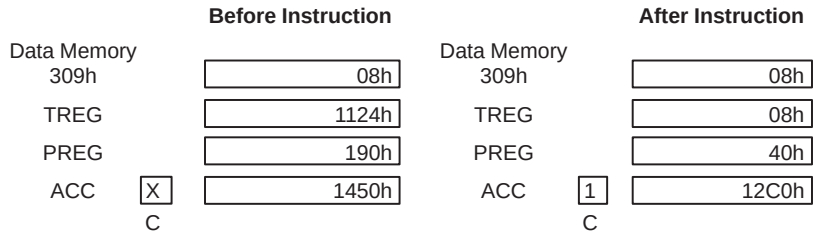
Cycles for a Repeat (RPT) Execution of an SQRS Instruction

	Program			
Operand	**ROM**	**DARAM**	**SARAM**	**External**
DARAM	n	n	n	n+p
SARAM	n	n	n, n+1†	n+p
External	n+nd	n+nd	n+nd	n+1+p+nd

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

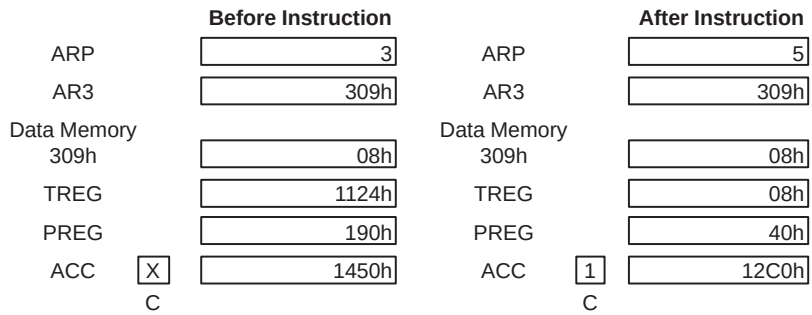
Example 1

```
SQRS    DAT9    ;(DP = 6: addresses 0300h–037Fh,
                ;PM = 0: no shift of product)
```



Example 2

```
SQRS    *,AR5   ;(PM = 0)
```



Syntax

SST #*m*, *dma* Direct addressing
SST #*m*, *ind* [, **AR***n*] Indirect addressing

Operands

dma: 7 LSBs of the data-memory address
n: Value from 0 to 7 designating the next auxiliary register
m: Select one of the following:
0 Indicates that ST0 will be stored
1 Indicates that ST1 will be stored
ind: Select one of the following seven options:
* *+ *– *0+ *0– *BR0+ *BR0–

Opcode

SST #0, *dma*

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

SST #0, *ind* [, **AR***n*]

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

Note: ARU, N, and NAR are defined in section 6.3, *Indirect Addressing Mode* (page 6-9).

SST #1, *dma*

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

SST #1, *ind* [, **AR***n*]

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

Note: ARU, N, and NAR are defined in section 6.3, *Indirect Addressing Mode* (page 6-9).

Execution

Increment PC, then ...
(status register STm) → data-memory address

Status Bits

None

Description

Status register ST0 or ST1 (whichever is specified) is stored in data memory.

In direct addressing mode, the specified status register is always stored in data page 0, regardless of the value of the data page pointer (DP) in ST0. Although the processor automatically accesses page 0, the DP is not physically modified; this allows the DP value to be stored unchanged when ST0 is stored.

In indirect addressing mode, the storage address is obtained from the auxiliary register selected; thus, the specified status register contents can be stored to an address on any page in data memory.

Status registers ST0 and ST1 are defined in section 4.5, *Status Registers ST0 and ST1*, on page 4-15.

Words 1

Cycles

Cycles for a Single SST Instruction

	Program			
Operand	**ROM**	**DARAM**	**SARAM**	**External**
DARAM	1	1	1	1+p
SARAM	1	1	1, 2†	1+p
External	2+d	2+d	2+d	4+d+p

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

Cycles for a Repeat (RPT) Execution of an SST Instruction

	Program			
Operand	**ROM**	**DARAM**	**SARAM**	**External**
DARAM	n	n	n	n+p
SARAM	n	n	n, n+2†	n+p
External	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     #0,96     ;Direct addressing: data page 0
                  ;accessed automatically
```

	Before Instruction		After Instruction
ST0	0A408h	ST0	0A408h
Data Memory 60h	0Ah	Data Memory 60h	0A408h

Example 2

```
SST     #1,*,AR7  ;Indirect addressing
```

	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

SUB *dma* [, *shift*]	Direct addressing
SUB *dma*,**16**	Direct with left shift of 16
SUB *ind* [,*shift* [, **AR***n*]]	Indirect addressing
SUB *ind*,**16**[, **AR***n*]	Indirect with left shift of 16
SUB #*k*	Short immediate
SUB #*lk* [,*shift*]	Long immediate

Operands

dma:	7 LSBs of the data-memory address
shift:	Left shift value from 0 to 15 (defaults to 0)
n:	Value from 0 to 7 designating the next auxiliary register
k:	8-bit short immediate value
lk:	16-bit long immediate value
ind:	Select one of the following seven options: * *+ *– *0+ *0– *BR0+ *BR0–

Opcode

SUB *dma* [,*shift*]

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

SUB *dma,* **16**

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

SUB *ind* [, *shift* [, **AR***n*]]

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

Note: ARU, N, and NAR are defined in section 6.3, *Indirect Addressing Mode* (page 6-9).

SUB *ind*,**16** [, **AR***n*]

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

Note: ARU, N, and NAR are defined in section 6.3, *Indirect Addressing Mode* (page 6-9).

SUB #*k*

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

SUB #*lk* [, *shift*]

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	0	1	0	shift			
lk															

Execution

Increment PC, then ...

Event	*Addressing mode*
(ACC) – ((data-memory address) $\times$ 2^{shift}) $\rightarrow$ ACC	Direct or indirect
(ACC) – ((data-memory address) $\times$ 2^{16}) $\rightarrow$ ACC	Direct or indirect (shift of 16)
(ACC) – k $\rightarrow$ ACC	Short immediate
(ACC) – lk $\times$ 2^{shift} $\rightarrow$ ACC	Long immediate

Status Bits

Affected by	*Affects*	*Addressing mode*
OVM and SXM	OV and C	Direct or indirect
OVM	OV and C	Short immediate
OVM and SXM	OV and C	Long immediate

Description

In direct, indirect, and long immediate addressing, the content of the addressed data-memory location or a 16-bit constant are left shifted and subtracted from 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 then stored in the accumulator.

If 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.

Normally, the carry bit is cleared (C = 0) if the result of the subtraction generates a borrow; it is set (C = 1) if it does not generate a borrow. However, if a 16-bit shift is specified with the subtraction, the instruction will clear the carry bit if a borrow is generated but will not affect the carry bit otherwise.

Words

Words	*Addressing mode*
1	Direct, indirect or short immediate
2	Long immediate

Cycles

Cycles for a Single SUB Instruction (Using Direct and Indirect Addressing)

	Program			
Operand	**ROM**	**DARAM**	**SARAM**	**External**
DARAM	1	1	1	1+p
SARAM	1	1	1, 2†	1+p
External	1+d	1+d	1+d	2+d+p

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

Cycles for a Repeat (RPT) Execution of an SUB Instruction (Using Direct and Indirect Addressing)

	Program			
Operand	**ROM**	**DARAM**	**SARAM**	**External**
DARAM	n	n	n	n+p
SARAM	n	n	n, n+1†	n+p
External	n+nd	n+nd	n+nd	n+1+p+nd

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

Cycles for a Single SUB Instruction (Using Short Immediate Addressing)

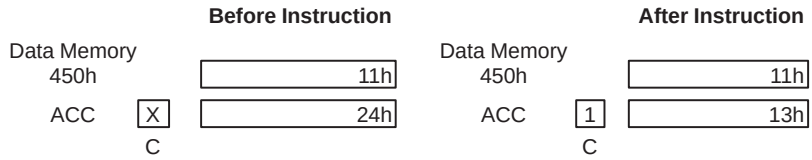
ROM	**DARAM**	**SARAM**	**External**
1	1	1	1+p

Cycles for a Single SUB Instruction (Using Long Immediate Addressing)

ROM	**DARAM**	**SARAM**	**External**
2	2	2	2+2p

Example 1

```
SUB     DAT80     ;(DP = 8: addresses 0400h–047Fh
```



Example 2

```
SUB     *–,1,AR0  ;(Left shift by 1, SXM = 0)
```

	Before Instruction		After Instruction	
ARP		7	ARP	0
AR7		301h	AR7	300h
Data Memory 301h		04h	Data Memory 301h	04h
ACC	X	09h	ACC	1 01h
	C			C

Example 3

```
SUB     #8h
```

	Before Instruction		After Instruction	
ACC	X	07h	ACC	0 FFFFFFFFh
	C			C

Example 4

```
SUB     #0FFFh,4  ;(Left shift by four, SXM = 0)
```

	Before Instruction		After Instruction	
ACC	X	0FFFFh	ACC	1 0Fh
	C			C

Syntax

SUBB *dma* Direct addressing
SUBB *ind* [, **AR***n*] Indirect addressing

Operands

dma: 7 LSBs of the data-memory address
n: Value from 0 to 7 designating the next auxiliary register
ind: Select one of the following seven options:
* *+ *– *0+ *0– *BR0+ *BR0–

Opcode

SUBB *dma*

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

SUBB *ind* [, **AR***n*]

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

Note: ARU, N, and NAR are defined in section 6.3, *Indirect Addressing Mode* (page 6-9).

Execution

Increment PC, then ...
(ACC) – (data-memory address) – (logical inversion of C) → ACC

Status Bits

Affected by	*Affects*
OVM	OV and C

This instruction is not affected by SXM.

Description

The content of the addressed data-memory location and the logical inversion of the carry bit is subtracted from the accumulator with sign extension suppressed. The carry bit is then affected in the normal manner: the carry bit is cleared (C = 0) if the result of the subtraction generates a borrow; it is set (C = 1) if it does not generate a borrow.

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

Words

1

Cycles

Cycles for a Single SUBB Instruction

	Program			
Operand	**ROM**	**DARAM**	**SARAM**	**External**
DARAM	1	1	1	1+p
SARAM	1	1	1, 2†	1+p
External	1+d	1+d	1+d	2+d+p

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

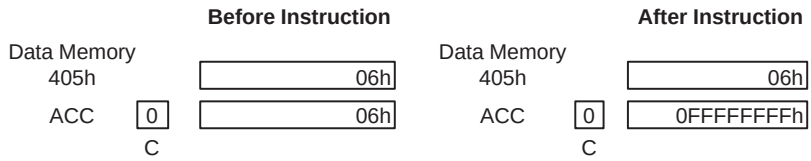
Cycles for a Repeat (RPT) Execution of an SUBB Instruction

	Program			
Operand	**ROM**	**DARAM**	**SARAM**	**External**
DARAM	n	n	n	n+p
SARAM	n	n	n, n+1†	n+p
External	n+nd	n+nd	n+nd	n+1+p+nd

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

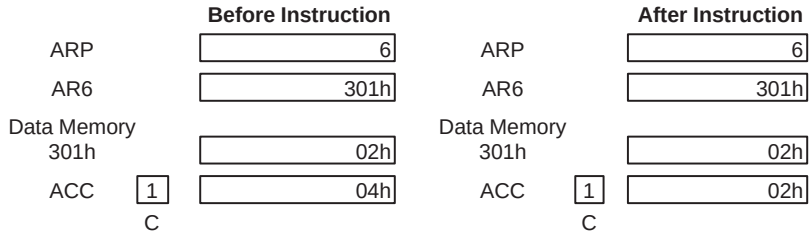
Example 1

```
SUBB    DAT5    ;(DP = 8: addresses 0400h–047Fh)
```



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

SUBC *dma* Direct addressing
SUBC *ind* [, **AR***n*] Indirect addressing

Operands

dma: 7 LSBs of the data-memory address
n: Value from 0 to 7 designating the next auxiliary register
ind: Select one of the following seven options:
* *+ *– *0+ *0– *BR0+ *BR0–

Opcode

SUBC *dma*

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

SUBC *ind* [, **AR***n*]

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

Note: ARU, N, and NAR are defined in section 6.3, *Indirect Addressing Mode* (page 6-9).

Execution

For (ACC) ≥ 0 and (data-memory address) ≥ 0:

Increment PC, then ...
(ACC) – [(data-memory address) × 2^{15}] → ALU output

If ALU output ≥ 0
Then (ALU output) × 2 + 1 → ACC
Else (ACC) × 2 → ACC

Status Bits

Affects
OV and C

Description

The SUBC instruction performs conditional subtraction, which can be used for division as follows: Place a positive 16-bit dividend in the low accumulator and clear the high accumulator. Place a 16-bit positive divisor in data memory. Execute SUBC 16 times. After completion of the last SUBC, the quotient of the division is in the lower-order 16 bits of the accumulator, and the remainder is in the higher-order 16 bits of the accumulator. For negative accumulator and/or data-memory values, SUBC cannot be used for division.

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 0s. The number of executions of SUBC is reduced from 16 by that number. One leading 0 is always significant.

SUBC operations performed as stated above are not affected by the sign-extension mode bit (SXM).

SUBC affects OV but is not affected by OVM; 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: the carry bit is cleared (C = 0) if the result of the subtraction generates a borrow and is set (C = 1) if it does not generate a borrow.

Words 1

Cycles

Cycles for a Single SUBC Instruction

	Program			
Operand	**ROM**	**DARAM**	**SARAM**	**External**
DARAM	1	1	1	1+p
SARAM	1	1	1, 2†	1+p
External	1+d	1+d	1+d	2+d+p

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

Cycles for a Repeat (RPT) Execution of an SUBC Instruction

	Program			
Operand	**ROM**	**DARAM**	**SARAM**	**External**
DARAM	n	n	n	n+p
SARAM	n	n	n, n+1†	n+p
External	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)
```

	Before Instruction		After Instruction	
Data Memory 302h		01h	Data Memory 302h	01h
ACC	X	04h	ACC	0	08h
C			C	

Example 2

```
RPT     #15
SUBC    *
```

	Before Instruction		After Instruction	
ARP		3	ARP	3
AR3		1000h	AR3	1000h
Data Memory 1000h		07h	Data Memory 1000h	07h
ACC	X	41h	ACC	1	20009h
C			C	

Syntax

SUBS *dma* Direct addressing
SUBS *ind* [, **AR***n*] Indirect addressing

Operands

dma: 7 LSBs of the data-memory address
n: Value from 0 to 7 designating the next auxiliary register
ind: Select one of the following seven options:
* *+ *– *0+ *0– *BR0+ *BR0–

Opcode

SUBS *dma*

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

SUBS *ind* [, **AR***n*]

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

Note: ARU, N, and NAR are defined in section 6.3, *Indirect Addressing Mode* (page 6-9).

Execution

Increment PC, then ...
(ACC) – (data-memory address) → ACC

Status Bits

Affected by	*Affects*
OVM	OV and C

This instruction is not affected by SXM.

Description

The content of the specified data-memory location is 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.

The carry bit is cleared (C = 0) if the result of the subtraction generates a borrow and is set (C = 1) if it does not generate a borrow.

Words

1

Cycles

Cycles for a Single SUBS Instruction

	Program			
Operand	**ROM**	**DARAM**	**SARAM**	**External**
DARAM	1	1	1	1+p
SARAM	1	1	1, 2†	1+p
External	1+d	1+d	1+d	2+d+p

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

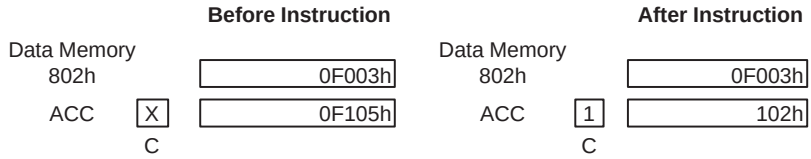
Cycles for a Repeat (RPT) Execution of an SUBS Instruction

	Program			
Operand	**ROM**	**DARAM**	**SARAM**	**External**
DARAM	n	n	n	n+p
SARAM	n	n	n, n+1†	n+p
External	n+nd	n+nd	n+nd	n+1+p+nd

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

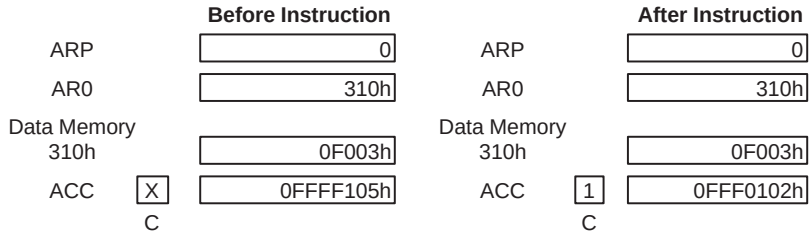
Example 1

```
SUBS    DAT2        ;(DP = 16, SXM = 1)
```



Example 2

```
SUBS    *           ;(SXM = 1)
```



Syntax

SUBT *dma* Direct addressing
SUBT *ind* [, **AR***n*] Indirect addressing

Operands

dma: 7 LSBs of the data-memory address
n: Value from 0 to 7 designating the next auxiliary register
ind: Select one of the following seven options:
* *+ *– *0+ *0– *BR0+ *BR0–

Opcode

SUBT *dma*

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

SUBT *ind* [, **AR***n*]

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

Note: ARU, N, and NAR are defined in section 6.3, *Indirect Addressing Mode* (page 6-9).

Execution

Increment PC, then ...
(ACC) – [(data-memory address) $\times$ $2^{(TREG(3:0))}$] $\rightarrow$ (ACC)

If SXM = 1
Then (data-memory address) is sign-extended.
If SXM = 0
Then (data-memory address) is not sign-extended.

Status Bits

Affected by	*Affects*
OVM and SXM	OV and C

Description

The data-memory value is left shifted and subtracted from the accumulator. The left shift is defined by the four LSBs of TREG, 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.

The carry bit is cleared (C = 0) if the result of the subtraction generates a borrow and is set (C = 1) if it does not generate a borrow.

Words

1

Cycles

Cycles for a Single SUBT Instruction

	Program			
Operand	**ROM**	**DARAM**	**SARAM**	**External**
DARAM	1	1	1	1+p
SARAM	1	1	1, 2†	1+p
External	1+d	1+d	1+d	2+d+p

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

Cycles for a Repeat (RPT) Execution of an SUBT Instruction

	Program			
Operand	**ROM**	**DARAM**	**SARAM**	**External**
DARAM	n	n	n	n+p
SARAM	n	n	n, n+1†	n+p
External	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 = 5: addresses 0280h–02FFh)
```

	Before Instruction		After Instruction	
Data Memory 2FFh		06h		06h
TREG		08h		08h
ACC	X	0FDA5h	1	0F7A5h
	C		C	

Example 2

```
SUBT    *
```

	Before Instruction		After Instruction	
ARP		1		1
AR1		800h		800h
Data Memory 800h		01h		01h
TREG		08h		08h
ACC	X	0h	0	FFFFFF00h
	C		C	

Syntax

TBLR *dma* — Direct addressing
TBLR *ind* [, **AR***n*] — Indirect addressing

Operands

dma: 7 LSBs of the data-memory address
n: Value from 0 to 7 designating the next auxiliary register
ind: Select one of the following seven options:
* *+ *– *0+ *0– *BR0+ *BR0–

Opcode

TBLR *dma*

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

TBLR *ind* [, **AR***n*]

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

Note: ARU, N, and NAR are defined in section 6.3, *Indirect Addressing Mode* (page 6-9).

Execution

Increment PC, then ...

(PC) → MSTACK
(ACC(15:0)) → PC
(pma) → data-memory address
For indirect, modify (current AR) and (ARP) as specified
(PC) + 1 → PC

While (repeat counter) ≠ 0
  (pma) → data-memory address
  For indirect, modify (current AR) and (ARP) as specified
  (PC) + 1 → PC
  (repeat counter) –1 → repeat counter.

(MSTACK) → PC

Status Bits

None

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 repeated with the repeat (RPT) instruction, TBLR effectively becomes a single-cycle instruction, and the program counter that was loaded with (ACC(15:0)) is incremented once each cycle.

Words

1

Cycles

Cycles for a Single TBLR Instruction

	Program			
Operand	**ROM**	**DARAM**	**SARAM**	**External**
Source: DARAM/ROM Destination: DARAM	3	3	3	$3+p_{code}$
Source: SARAM Destination: DARAM	3	3	3	$3+p_{code}$
Source: External 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: External Destination: SARAM	$3+p_{src}$	$3+p_{src}$	$3+p_{src}$ $4+p_{src}$†	$3+p_{src}+p_{code}$
Source: DARAM/ROM Destination: External	$4+d_{dst}$	$4+d_{dst}$	$4+d_{dst}$	$6+d_{dst}+p_{code}$
Source: SARAM Destination: External	$4+d_{dst}$	$4+d_{dst}$	$4+d_{dst}$	$6+d_{dst}+p_{code}$
Source: External Destination: External	$4+p_{src}+d_{dst}$	$4+p_{src}+d_{dst}$	$4+p_{src}+d_{dst}$	$6+p_{src}+d_{dst}+p_{code}$

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

Cycles for a Repeat (RPT) Execution of a TBLR Instruction

	Program			
Operand	**ROM**	**DARAM**	**SARAM**	**External**
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: External Destination: DARAM	$n+2+np_{src}$	$n+2+np_{src}$	$n+2+np_{src}$	$n+2+np_{src}+p_{code}$

† 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

Cycles for a Repeat (RPT) Execution of a TBLR Instruction (Continued)

	Program			
Operand	**ROM**	**DARAM**	**SARAM**	**External**
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: External 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: External	2n+2+nd_{dst}	2n+2+nd_{dst}	2n+2+nd_{dst}	2n+4+nd_{dst}+p_{code}
Source: SARAM Destination: External	2n+2+nd_{dst}	2n+2+nd_{dst}	2n+2+nd_{dst}	2n+4+nd_{dst}+p_{code}
Source: External Destination: External	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: addresses 0200h–027Fh)
```

	Before Instruction	After Instruction
ACC	23h	23h
Program Memory 23h	306h	306h
Data Memory 206h	75h	306h

Example 2

```
TBLR    *,AR7
```

	Before Instruction	After Instruction
ARP	0	7
AR0	300h	300h
ACC	24h	24h
Program Memory 24h	307h	307h
Data Memory 300h	75h	307h

Syntax

TBLW *dma* — Direct addressing
TBLW *ind* [, **AR***n*] — Indirect addressing

Operands

dma: 7 LSBs of the data-memory address
n: Value from 0 to 7 designating the next auxiliary register
ind: Select one of the following seven options:
* *+ *– *0+ *0– *BR0+ *BR0–

Opcode

TBLW *dma*

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

TBLW *ind* [, **AR***n*]

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

Note: ARU, N, and NAR are defined in section 6.3, *Indirect Addressing Mode* (page 6-9).

Execution

Increment PC, then ...
(PC+1) → MSTACK
(ACC(15:0)) → PC+1
(data-memory address) → pma,
For indirect, modify (current AR) and (ARP) as specified
(PC) + 1 → PC

While (repeat counter) ≠ 0
  (data-memory address) → pma,
  For indirect, modify (current AR) and (ARP) as specified
  (PC) + 1 → PC
  (repeat counter) –1 → repeat counter.

(MSTACK) → PC+1

Status Bits

None

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 repeated with the repeat (RPT) instruction, TBLW effectively becomes a single-cycle instruction, and the program counter that was loaded with (ACC(15:0)) is incremented once each cycle.

Words

1

Cycles

Cycles for a Single TBLW Instruction

	Program			
Operand	**ROM**	**DARAM**	**SARAM**	**External**
Source: DARAM/ROM Destination: DARAM	3	3	3	$3+p_{code}$
Source: SARAM Destination: DARAM	3	3	3	$3+p_{code}$
Source: External Destination: DARAM	$3+d_{src}$	$3+d_{src}$	$3+d_{src}$	$3+d_{src}+p_{code}$
Source: DARAM/ROM Destination: SARAM	3	3	3 4†	$3+p_{code}$
Source: SARAM Destination: SARAM	3	3	3 4†	$3+p_{code}$
Source: External Destination: SARAM	$3+d_{src}$	$3+d_{src}$	$3+d_{src}$ $4+d_{src}$†	$3+d_{src}+p_{code}$
Source: DARAM/ROM Destination: External	$4+p_{dst}$	$4+p_{dst}$	$4+p_{dst}$	$5+p_{dst}+p_{code}$
Source: SARAM Destination: External	$4+p_{dst}$	$4+p_{dst}$	$4+p_{dst}$	$5+p_{dst}+p_{code}$
Source: External Destination: External	$4+d_{src}+p_{dst}$	$4+d_{src}+p_{dst}$	$4+d_{src}+p_{dst}$	$5+d_{src}+p_{dst}+p_{code}$

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

Cycles for a Repeat (RPT) Execution of a TBLW Instruction

	Program			
Operand	**ROM**	**DARAM**	**SARAM**	**External**
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: External Destination: DARAM	$n+2+nd_{src}$	$n+2+nd_{src}$	$n+2+nd_{src}$	$n+2+nd_{src}+p_{code}$

† 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

Cycles for a Repeat (RPT) Execution of a TBLW Instruction (Continued)

	Program			
Operand	**ROM**	**DARAM**	**SARAM**	**External**
Source: DARAM/ROM 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: External 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/ROM Destination: External	2n+2+np_{dst}	2n+2+np_{dst}	2n+2+np_{dst}	2n+3+np_{dst}+p_{code}
Source: SARAM Destination: External	2n+2+np_{dst}	2n+2+np_{dst}	2n+2+np_{dst}	2n+3+np_{dst}+p_{code}
Source: External Destination: External	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: addresses 1000h–107Fh)
```

	Before Instruction		After Instruction
ACC	257h	ACC	257h
Data Memory 1005h	4339h	Data Memory 1005h	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 **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

Status Bits 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 (PC) plus 1 onto the hardware stack. The instruction at location 22h may contain a branch instruction to transfer control to the TRAP routine. Putting (PC + 1) onto the stack enables a return instruction to pop the return address (which points to the instruction after TRAP) from the stack. The TRAP instruction is not maskable.

Words 1

Cycles

Cycles for a Single TRAP Instruction

ROM	DARAM	SARAM	External
4	4	4	4+3p†

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

Example

```
TRAP    ;PC + 1 is pushed onto the stack, and then
        ;control is passed to program memory location
        ;22h.
```

Syntax

XOR *dma*	Direct addressing
XOR *ind* [, **AR***n*]	Indirect addressing
XOR #*lk* [, *shift*]	Long immediate addressing
XOR #*lk*,**16**	Long immediate with left shift of 16

Operands

dma:	7 LSBs of the data-memory address
shift:	Left shift value from 0 to 15 (defaults to 0)
n:	Value from 0 to 7 designating the next auxiliary register
lk:	16-bit long immediate value
ind:	Select one of the following seven options: * *+ *– *0+ *0– *BR0+ *BR0–

Opcode

XOR *dma*

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	dma						

XOR *ind* [, **AR***n*]

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	ARU			N	NAR		

Note: ARU, N, and NAR are defined in section 6.3, *Indirect Addressing Mode* (page 6-9).

XOR #*lk* [, *shift*]

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	shift			
lk															

XOR #*lk,* **16**

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
lk															

Execution

Increment PC, then ...

Event(s)	*Addressing mode*
(ACC(15:0)) XOR (data-memory address) → ACC(15:0) (ACC(31:16)) → ACC(31:16)	Direct or indirect
(ACC(31:0)) XOR lk × 2^{shift} → ACC(31:0)	Long immediate
(ACC(31:0)) XOR lk × 2^{16} → ACC(31:0)	Long immediate with left shift of 16

Status Bits None

Description With direct or indirect addressing, the low half of the accumulator value is exclusive ORed with the content of the addressed data memory location, and the result replaces the low half of the accumulator value; the upper half of the accumulator value is unaffected. With immediate addressing, the long immediate constant is shifted and zero filled on both ends and exclusive ORed with the entire content of the accumulator. The carry bit (C) is unaffected by XOR.

Words

Words	*Addressing mode*
1	Direct or indirect
2	Long immediate

Cycles

Cycles for a Single XOR Instruction (Using Direct and Indirect Addressing)

	Program			
Operand	**ROM**	**DARAM**	**SARAM**	**External**
DARAM	1	1	1	1+p
SARAM	1	1	1, 2†	1+p
External	1+d	1+d	1+d	2+d+p

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

Cycles for a Repeat (RPT) Execution of an XOR Instruction (Using Direct and Indirect Addressing)

	Program			
Operand	**ROM**	**DARAM**	**SARAM**	**External**
DARAM	n	n	n	n+p
SARAM	n	n	n, n+1†	n+p
External	n+nd	n+nd	n+nd	n+1+p+nd

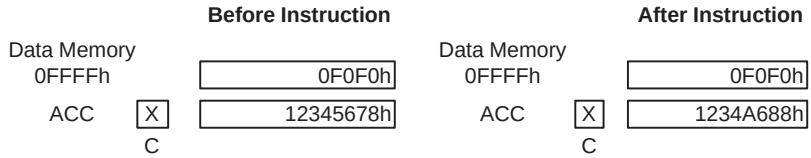
† If the operand and the code are in the same SARAM block

Cycles for a Single XOR Instruction (Using Long Immediate Addressing)

ROM	**DARAM**	**SARAM**	**External**
2	2	2	2+2p

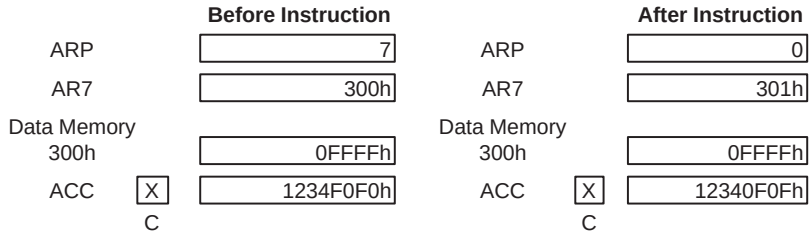
Example 1

```
XOR   DAT127        ;(DP = 511: addresses FF80h–FFFFh)
```



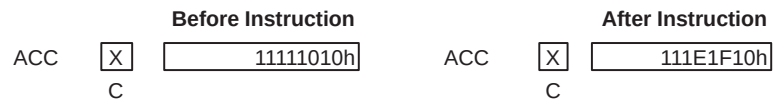
Example 2

```
XOR   *+,AR0
```



Example 3

```
XOR   #0F0F0h,4     ;(First shift data value left by
                    ;four)
```



Syntax

ZALR *dma* Direct addressing
ZALR *ind* [, **AR***n*] Indirect addressing

Operands

dma: 7 LSBs of the data-memory address
n: Value from 0 to 7 designating the next auxiliary register
ind: Select one of the following seven options:
* *+ *– *0+ *0– *BR0+ *BR0–

Opcode

ZALR *dma*

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

ZALR *ind* [, **AR***n*]

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

Note: ARU, N, and NAR are defined in section 6.3, *Indirect Addressing Mode* (page 6-9).

Execution

Increment PC, then ...
(data-memory address) → ACC(31:16)
8000h → ACC(15:0)

Status Bits

None

Description

The ZALR instruction loads a 16-bit data-memory value into the high word of the accumulator. The instruction rounds the value by adding half of the value of the LSB: bit 15 of the accumulator is set, and bits 14 are cleared.

Words

1

Cycles

Cycles for a Single ZALR Instruction

	Program			
Operand	**ROM**	**DARAM**	**SARAM**	**External**
DARAM	1	1	1	1+p
SARAM	1	1	1, 2†	1+p
External	1+d	1+d	1+d	2+d+p

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

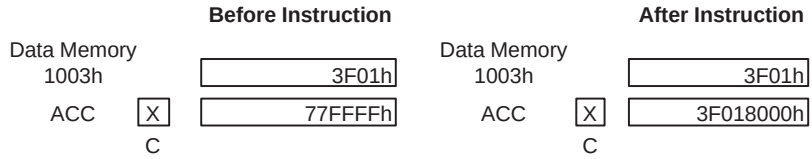
Cycles for a Repeat (RPT) Execution of a ZALR Instruction

	Program			
Operand	**ROM**	**DARAM**	**SARAM**	**External**
DARAM	n	n	n	n+p
SARAM	n	n	n, n+1†	n+p
External	n+nd	n+nd	n+nd	n+1+p+nd

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

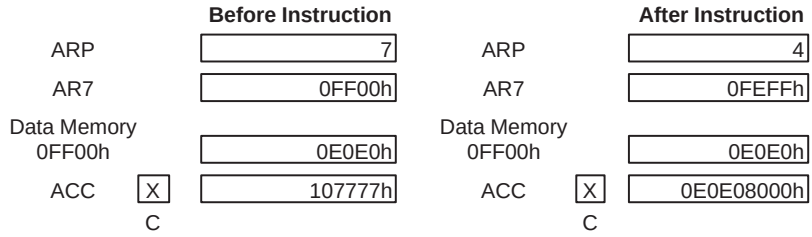
Example 1

```
ZALR    DAT3    ;(DP = 32: addresses 1000h–107Fh)
```



Example 2

```
ZALR    *–,AR4
```



Appendix A

TMS320C1x/C2x/C20x/C5x Instruction Set Comparison

This appendix contains a table that compares the TMS320C1x, TMS320C2x, TMS320C20x, and TMS320C5x instructions alphabetically. Each table entry shows the syntax for the instruction, indicates which devices support the instruction, and describes the operation of the instruction. Section A.1 shows a sample table entry and describes the symbols and abbreviations used in the table.

The TMS320C2x, TMS320C20x, and TMS320C5x devices have *enhanced instructions*; enhanced instructions are single mnemonics that perform the functions of several similar instructions. Section A.2 summarizes the enhanced instructions.

This appendix does not cover topics such as opcodes, instruction timing, or addressing modes; in addition to this book, the following documents cover such topics in detail:

TMS320C1x User's Guide (literature number SPRU013)

TMS320C2x User's Guide (literature number SPRU014)

TMS320C5x User's Guide (literature number SPRU056)

Topic **Page**

A.1 Using the Instruction Set Comparison Table

To help you read the comparison table, this section provides an example of a table entry and a list of acronyms.

A.1.1 An Example of a Table Entry

In cases where more than one syntax is used, the first syntax is usually for direct addressing and the second is usually for indirect addressing. Where three or more syntaxes are used, the syntaxes are normally specific to a device.

This is how the AND instruction appears in the table:

Syntax	1x	2x	2xx	5x	Description
AND *dma*	√	√	√	√	**AND With Accumulator**
AND {*ind*} [, *next ARP*]	√	√	√	√	TMS320C1x and TMS320C2x devices: AND the contents of the addressed data-memory location with the 16 LSBs of the accumulator. The 16 MSBs of the accumulator are ANDed with 0s.
AND #*lk* [, *shift*]			√	√	TMS320C20x and TMS320C5x devices: AND the contents of the addressed data-memory location or a 16-bit immediate value with the contents of the accumulator. The 16 MSBs of the accumulator are ANDed with 0s. If a shift is specified, left shift the constant before the AND. Low-order bits below and high-order bits above the shifted value are treated as 0s.

The first column, *Syntax*, states the mnemonic and the syntaxes for the AND instruction.

The checks in the second through the fifth columns, *1x*, *2x*, *2xx,* and *5x*, indicate the devices that can be used with each of the syntaxes.

1x refers to the TMS320C1x devices
2x refers to the TMS320C2x devices, including TMS320C25
2xx refers to the TMS320C20x devices
5x refers to the TMS320C5x devices

In this example, you can use the first two syntaxes with TMS320C1x, TMS320C2x, TMS320C20x, and TMS320C5x devices, but you can use the last syntax only with TMS320C20x and TMS320C5x devices.

The sixth column, *Description*, briefly describes how the instruction functions. Often, an instruction functions slightly differently for the different devices: read the entire description before using the instruction.

A.1.2 Symbols and Acronyms Used in the Table

The following table lists the instruction set symbols and acronyms used throughout this chapter:

Table A–1. Symbols and Acronyms Used in the Instruction Set Comparison Table

Symbol	Description	Symbol	Description
lk	16-bit immediate value	INTM	interrupt mask bit
k	8-bit immediate value	INTR	interrupt mode bit
{ind}	indirect address	OV	overflow bit
ACC	accumulator	P	program bus
ACCB	accumulator buffer	PA	port address
AR	auxiliary register	PC	program counter
ARCR	auxiliary register compare	PM	product shifter mode
ARP	auxiliary register pointer	pma	program-memory address
BMAR	block move address register	RPTC	repeat counter
BRCR	block repeat count register	shift, $shift_n$	shift value
C	carry bit	src	source address
DBMR	dynamic bit manipulation register	ST	status register
dma	data-memory address	SXM	sign-extension mode bit
DP	data-memory page pointer	TC	test/control bit
dst	destination address	T	temporary register
FO	format status list	TREGn	TMS320C5x temporary register (0–2)
FSX	external framing pulse	TXM	transmit mode status register
IMR	interrupt mask register	XF	XF pin status bit

Based on the device, this is how the indirect addressing operand {ind} is interpreted:

{ind}	**'C1x**:	{ * | *+ | *– }
	'C2x:	{ * | *+ | *– | *0+| *0– | *BR0+ | *BR0– }
	'C20x:	{ * | *+ | *– | *0+| *0– | *BR0+ | *BR0– }
	'C5x:	{ * | *+ | *– | *0+| *0– | *BR0+ | *BR0– }

where the possible options are separated by vertical bars (|). For example:

ADD {*ind*}

is interpreted as:

'C1x devices	ADD { * | *+ | *– }
'C2x devices	ADD { * | *+ | *– | *0+ | *0– | *BR0+ | *BR0– }
'C20x devices	ADD { * | *+ | *– | *0+ | *0– | *BR0+ | *BR0– }
'C5x devices	ADD { * | *+ | *– | *0+ | *0– | *BR0+ | *BR0– }

Based on the device, these are the sets of values for shift, $shift_1$, and $shift_2$:

shift	**'C1x**:	0–15 (shift of 0–15 bits)
	'C2x:	0–15 (shift of 0–15 bits)
	'C20x:	0–16 (shift of 0–16 bits)
	'C5x:	0–16 (shift of 0–16 bits)
$shift_1$	**'C1x**:	n/a
	'C2x:	0–15 (shift of 0–15 bits)
	'C20x:	0–16 (shift of 0–16 bits)
	'C5x:	0–16 (shift of 0–16 bits)
$shift_2$	**'C1x**:	n/a
	'C2x:	n/a
	'C20x:	0–15 (shift of 0–15 bits)
	'C5x:	0–15 (shift of 0–15 bits)

In some cases, the sets are smaller; in these cases, the valid sets are given in the *Description* column of the table.

A.2 Enhanced Instructions

An enhanced instruction is a single mnemonic that performs the functions of several similar instructions. For example, the enhanced instruction ADD performs the ADD, ADDH, ADDK, and ADLK functions and replaces any of these other instructions at assembly time. For example, when a program using ADDH is assembled for the 'C20x or 'C5x, ADDH is replaced by an ADD instruction that performs the same function. These enhanced instructions are valid for TMS320C2x, TMS320C20x, and TMS320C5x devices (not TMS320C1x).

Table A–2 below summarizes the enhanced instructions and the functions that the enhanced instructions perform (based on TMS320C1x/2x mnemonics).

Table A–2. Summary of Enhanced Instructions

Enhanced Instruction	Includes These Operations
ADD	ADD, ADDH, ADDK, ADLK
AND	AND, ANDK
BCND	BBNZ, BBZ, BC, BCND, BGEZ, BGZ, BIOZ, BLEZ, BLZ, BNC, BNV, BNZ, BV, BZ
BLDD	BLDD, BLKD
BLDP	BLDP, BLKP
CLRC	CLRC, CNFD, EINT, RC, RHM, ROVM, RSXM, RTC, RXF
LACC	LAC, LACC, LALK, ZALH
LACL	LACK, LACL, ZAC, ZALS
LAR	LAR, LARK, LRLK
LDP	LDP, LDPK
LST	LST, LST1
MAR	LARP, MAR
MPY	MPY, MPYK
OR	OR, ORK
RPT	RPT, RPTK
SETC	CNFP, DINT, SC, SETC, SHM, SOVM, SSXM, STC, SXF
SUB	SUB, SUBH, SUBK

A.3 Instruction Set Comparison Table

Syntax	1x	2x	2xx	5x	Description
ABS	√	√	√	√	**Absolute Value of Accumulator** If the contents of the accumulator are less than zero, replace the contents with the 2s complement of the contents. If the contents are ≥ 0, the accumulator is not affected.
ADCB				√	**Add ACCB to Accumulator With Carry** Add the contents of the ACCB and the value of the carry bit to the accumulator. If the result of the addition generates a carry from the accumulator's MSB, the carry bit is set to 1.
ADD *dma* [*, shift*]	√	√	√	√	**Add to Accumulator With Shift**
ADD {*ind*} [*, shift* [*, next ARP*]]	√	√	√	√	TMS320C1x and TMS320C2x devices: Add the contents of the addressed data-memory location to the accumulator; if a shift is specified, left shift the contents of the location before the add. During shifting, low-order bits are zero filled, and high-order bits are sign extended. TMS320C20x and TMS320C5x devices: Add the contents of the addressed data-memory location or an immediate value to the accumulator; if a shift is specified, left shift the data before the add. During shifting, low-order bits are zero filled, and high-order bits are sign extended if SXM = 1.
ADD #*k*			√	√	
ADD #*lk* [*, shift2*]			√	√	
ADDB				√	**Add ACCB to Accumulator** Add the contents of the ACCB to the accumulator.
ADDC *dma*		√	√	√	**Add to Accumulator With Carry**
ADDC {*ind*} [*, next ARP*]		√	√	√	Add the contents of the addressed data-memory location and the carry bit to the accumulator.
ADDH *dma*	√	√	√	√	**Add High to Accumulator**
ADDH {*ind*} [*, next ARP*]	√	√	√	√	Add the contents of the addressed data-memory location to the 16 MSBs of the accumulator. The LSBs are not affected. If the result of the addition generates a carry, the carry bit is set to 1. TMS320C2x, TMS320C20x, and TMS320C5x devices: If the result of the addition generates a carry from the accumulator's MSB, the carry bit is set to 1.

Syntax	1x	2x	2xx	5x	Description
ADDK *#k*		√	√	√	**Add to Accumulator Short Immediate** TMS320C1x devices: Add an 8-bit immediate value to the accumulator. TMS320C2x, TMS320C20x, and TMS320C5x devices: Add an 8-bit immediate value, right justified, to the accumulator with the result replacing the accumulator contents. The immediate value is treated as an 8-bit positive number; sign extension is suppressed.
ADDS *dma*	√	√	√	√	**Add to Accumulator With Sign Extension Suppressed** Add the contents of the addressed data-memory location to the accumulator. The value is treated as a 16-bit unsigned number; sign extension is suppressed.
ADDS {*ind*} [*, next ARP*]	√	√	√	√	
ADDT *dma*		√	√	√	**Add to Accumulator With Shift Specified by T Register** Left shift the contents of the addressed data-memory location by the value in the 4 LSBs of the T register; add the result to the accumulator. If a shift is specified, left shift the data before the add. During shifting, low-order bits are zero filled, and high-order bits are sign extended if SXM = 1. TMS320C20x and TMS320C5x devices: If the result of the addition generates a carry from the accumulator's MSB, the carry bit is set to 1.
ADDT {*ind*} [*, next ARP*]		√	√	√	
ADLK *#lk* [*, shift*]		√	√	√	**Add to Accumulator Long Immediate With Shift** Add a 16-bit immediate value to the accumulator; if a shift is specified, left shift the value before the add. During shifting, low-order bits are zero filled, and high-order bits are sign extended if SXM = 1.
ADRK *#k*		√	√	√	**Add to Auxiliary Register Short Immediate** Add an 8-bit immediate value to the current auxiliary register.

Syntax	1x	2x	2xx	5x	Description
AND *dma* **AND** {*ind*} [, *next ARP*] **AND** #*lk* [, *shift*]	√	√	√	√	**AND With Accumulator** TMS320C1x and TMS320C2x devices: AND the contents of the addressed data-memory location with the 16 LSBs of the accumulator. The 16 MSBs of the accumulator are ANDed with 0s. TMS320C20x and TMS320C5x devices: AND the contents of the addressed data-memory location or a 16-bit immediate value with the contents of the accumulator. The 16 MSBs of the accumulator are ANDed with 0s. If a shift is specified, left shift the constant before the AND. Low-order bits below and high-order bits above the shifted value are treated as 0s.
ANDB				√	**AND ACCB to Accumulator** AND the contents of the ACCB to the accumulator.
ANDK #*lk* [, *shift*]		√	√	√	**AND Immediate With Accumulator With Shift** AND a 16-bit immediate value with the contents of the accumulator; if a shift is specified, left shift the constant before the AND.
APAC	√	√	√	√	**Add P Register to Accumulator** Add the contents of the P register to the accumulator. TMS320C2x, TMS320C20x, and TMS320C5x devices: Before the add, left shift the contents of the P register as defined by the PM status bits.
APL [#*lk*] ,*dma* **APL** [#*lk,*] {*ind*} [, *next ARP*]				√ √	**AND Data-Memory Value With DBMR or Long Constant** AND the data-memory value with the contents of the DBMR or a long constant. If a long constant is specified, it is ANDed with the contents of the data-memory location. The result is written back into the data-memory location previously holding the first operand. If the result is 0, the TC bit is set to 1; otherwise, the TC bit is cleared.
B *pma* **B** *pma* [, {*ind*} [, *next ARP*]]	√	√	√		**Branch Unconditionally** Branch to the specified program-memory address. TMS320C2x and TMS320C20x devices: Modify the current AR and ARP as specified.

Syntax	1x	2x	2xx	5x	Description
B[*D*] *pma* [, {*ind*} [, *next ARP*]]				√	**Branch Unconditionally With Optional Delay** Modify the current auxiliary register and ARP as specified and pass control to the designated program-memory address. If you specify a delayed branch (BD), the next two instruction words (two 1-word instructions or one 2-word instruction) are fetched and executed before branching.
BACC		√	√		**Branch to Address Specified by Accumulator** Branch to the location specified by the 16 LSBs of the accumulator.
BACC[*D*]				√	**Branch to Address Specified by Accumulator With Optional Delay** Branch to the location specified by the 16 LSBs of the accumulator. If you specify a delayed branch (BACCD), the next two instruction words (two 1-word instructions or one 2-word instruction) are fetched and executed before branching.
BANZ *pma* **BANZ** *pma* [, {*ind*} [, *next ARP*]]	√	√	√		**Branch on Auxiliary Register Not Zero** If the contents of the 9 LSBs of the current auxiliary register (TMS320C1x) or the contents of the entire current auxiliary register (TMS320C2x) are ≠ 0, branch to the specified program-memory address. TMS320C2x and TMS320C20x devices: Modify the current AR and ARP (if specified) or decrement the current AR (default). TMS320C1x devices: Decrement the current AR.
BANZ[*D*] *pma* [, {*ind*} [, *next ARP*]]				√	**Branch on Auxiliary Register Not Zero With Optional Delay** If the contents of the current auxiliary register are ≠ 0, branch to the specified program-memory address. Modify the current AR and ARP as specified, or decrement the current AR. If you specify a delayed branch (BANZD), the next two instruction words (two 1-word instructions or one 2-word instruction) are fetched and executed before branching.

Syntax	1x	2x	2xx	5x	Description
BBNZ *pma* [, {*ind*} [, *next ARP*]]		√	√	√	**Branch on Bit ≠ Zero** If the TC bit = 1, branch to the specified program-memory address. TMS320C2x devices: Modify the current AR and ARP as specified. TMS320C20x and TMS320C5x devices: If the –p porting switch is used, modify the current AR and ARP as specified.
BBZ *pma* [, {*ind*} [, *next ARP*]]		√	√	√	**Branch on Bit = Zero**
BBZ *pma*				√	If the TC bit = 0, branch to the specified program-memory address. TMS320C2x devices: Modify the current AR and ARP as specified. TMS320C20x and TMS320C5x devices: Modify the current AR and ARP as specified when the –p porting switch is used.
BC *pma* [, {*ind*} [, *next ARP*]]		√		√	**Branch on Carry**
BC *pma*			√	√	If the C bit = 1, branch to the specified program-memory address. TMS320C2x devices: Modify the current AR and ARP as specified. TMS320C20x and TMS320C5x devices: Modify the current AR and ARP as specified when the –p porting switch is used.
BCND *pma*, $cond_1$ [, $cond_2$] [, ...]			√		**Branch Conditionally** Branch to the program-memory address if the specified conditions are met. Not all combinations of conditions are meaningful.
BCND[*D*] *pma*, $cond_1$ [, $cond_2$] [, ...]				√	**Branch Conditionally With Optional Delay** Branch to the program-memory address if the specified conditions are met. Not all combinations of conditions are meaningful. If you specify a delayed branch (BCNDD), the next two instruction words (two 1-word instructions or one 2-word instruction) are fetched and executed before branching.

Syntax	1x	2x	2xx	5x	Description
BGEZ *pma*	√		√	√	**Branch if Accumulator ≥ Zero**
BGEZ *pma* [, {*ind*} [, *next ARP*]]		√		√	If the contents of the accumulator ≥ 0, branch to the specified program-memory address. TMS320C2x devices: Modify the current AR and ARP as specified. TMS320C20x and TMS320C5x devices: Modify the current AR and ARP as specified when the –p porting switch is used.
BGZ *pma*	√		√	√	**Branch if Accumulator > Zero**
BGZ *pma* [, {*ind*} [, *next ARP*]]		√		√	If the contents of the accumulator are > 0, branch to the specified program-memory address. TMS320C2x devices: Modify the current AR and ARP as specified. TMS320C20x and TMS320C5x devices: Modify the current AR and ARP as specified when the –p porting switch is used.
BIOZ *pma*	√		√	√	**Branch on I/O Status = Zero**
BIOZ *pma* [, {*ind*} [, *next ARP*]]		√		√	If the $\overline{\text{BIO}}$ pin is low, branch to the specified program-memory address. TMS320C2x devices: Modify the current AR and ARP as specified. TMS320C20x and TMS320C5x devices: Modify the current AR and ARP as specified when the –p porting switch is used.
BIT *dma, bit code*		√	√	√	**Test Bit**
BIT {*ind*}, *bit code* [, *next ARP*]		√	√	√	Copy the specified bit of the data-memory value to the TC bit in ST1.
BITT *dma*		√	√	√	**Test Bit Specified by T Register**
BITT {*ind*} [, *next ARP*]		√	√	√	TMS320C2x and TMS320C20x devices: Copy the specified bit of the data-memory value to the TC bit in ST1. The 4 LSBs of the T register specify which bit is copied. TMS320C5x devices: Copy the specified bit of the data-memory value to the TC bit in ST1. The 4 LSBs of the TREG2 specify which bit is copied.

Syntax	1x	2x	2xx	5x	Description
BLDD *#lk, dma*			√	√	**Block Move From Data Memory to Data Memory**
BLDD *#lk,* {*ind*} [, *next ARP*]			√	√	Copy a block of data memory into data memory. The block of data memory is pointed to by *src*, and the destination block of data memory is pointed to by *dst*.
BLDD *dma, #lk*			√	√	
BLDD {*ind*}, *#lk* [, *next ARP*]			√	√	TMS320C20x devices: The word of the source and/or the destination space can be pointed to with a long immediate value or a data-memory address. You can use the RPT instruction with BLDD to move consecutive words, pointed to indirectly in data memory, to a contiguous program-memory space. The number of words to be moved is 1 greater than the number contained in the RPTC at the beginning of the instruction.
BLDD BMAR, *dma*				√	
BLDD BMAR, {*ind*} [, *next ARP*]				√	
BLDD *dma* BMAR				√	
BLDD {*ind*}, BMAR [, *next ARP*]				√	
					TMS320C5x devices: The word of the source and/or the destination space can be pointed to with a long immediate value, the contents of the BMAR, or a data-memory address. You can use the RPT instruction with BLDD to move consecutive words, pointed to indirectly in data memory, to a contiguous program-memory space. The number of words to be moved is 1 greater than the number contained in the RPTC at the beginning of the instruction.
BLDP *dma*				√	**Block Move From Data Memory to Program Memory**
BLDP {*ind*} [, *next ARP*]				√	Copy a block of data memory into program memory pointed to by the BMAR. You can use the RPT instruction with BLDP to move consecutive words, indirectly pointed to in data memory, to a contiguous program-memory space pointed to by the BMAR.
BLEZ *pma*	√		√	√	**Branch if Accumulator ≤ Zero**
BLEZ *pma* [, {*ind*} [, *next ARP*]]		√	√	√	If the contents of the accumulator are ≤ 0, branch to the specified program-memory address.
					TMS320C2x devices: Modify the current AR and ARP as specified.
					TMS320C20x and TMS320C5x devices: Modify the current AR and ARP as specified when the –p porting switch is used.

Syntax	1x	2x	2xx	5x	Description
BLKD *dma1, dma2*		√	√	√	**Block Move From Data Memory to Data Memory**
BLKD *dma1,* {*ind*} [*, next ARP*]		√	√	√	Move a block of words from one location in data memory to another location in data memory. Modify the current AR and ARP as specified. RPT or RPTK must be used with BLKD, in the indirect addressing mode, if more than one word is to be moved. The number of words to be moved is 1 greater than the number contained in RPTC at the beginning of the instruction.
BLKP *pma, dma*		√	√	√	**Block Move From Program Memory to Data Memory**
BLKP *pma,* {*ind*} [*, next ARP*]		√	√	√	Move a block of words from a location in program memory to a location in data memory. Modify the current AR and ARP as specified. RPT or RPTK must be used with BLKD, in the indirect addressing mode, if more than one word is to be moved. The number of words to be moved is 1 greater than the number contained in RPTC at the beginning of the instruction.
BLPD† #*pma, dma*			√	√	**Block Move From Program Memory to Data Memory**
BLPD† #*pma,* {*ind*} [*, next ARP*]			√	√	
BLPD† BMAR, *dma*				√	Copy a block of program memory into data memory. The block of program memory is pointed to by *src*, and the destination block of data memory is pointed to by *dst*.
BLPD† BMAR, {*ind*} [*, next ARP*]				√	TMS320C20x devices: The word of the source space can be pointed to with a long immediate value. You can use the RPT instruction with BLPD to move consecutive words that are pointed at indirectly in data memory to a contiguous program-memory space.
					TMS320C5x devices: The word of the source space can be pointed to with a long immediate value or the contents of the BMAR. You can use the RPT instruction with BLPD to move consecutive words that are pointed at indirectly in data memory to a contiguous program-memory space.
BLZ *pma*	√		√	√	**Branch if Accumulator < Zero**
BLZ *pma* [*,* {*ind*} [*, next ARP*]]		√	√		If the contents of the accumulator are < 0, branch to the specified program-memory address.
					TMS320C2x devices: Modify the current AR and ARP as specified.
					TMS320C20x and TMS320C5x devices: Modify the current AR and ARP as specified when the –p porting switch is used.

† BLDD and BLPD are TMS320C5x and TMS320C20x instructions for the BLKD and BLKP instructions in the TMS320C2x and TMS320C1 devices. The assembler converts TMS320C2x code to BLKB and BLKP.

Syntax	1x	2x	2xx	5x	Description
BNC *pma* [, {*ind*} [, *next ARP*]]		√	√	√	**Branch on No Carry** If the C bit = 0, branch to the specified program-memory address. TMS320C2x devices: Modify the current AR and ARP as specified. TMS320C20x and TMS320C5x devices: Modify the current AR and ARP as specified when the –p porting switch is used.
BNV *pma* [, {*ind*} [, *next ARP*]]		√	√	√	**Branch if No Overflow** If the OV flag is clear, branch to the specified program-memory address. TMS320C2x devices: Modify the current AR and ARP as specified. TMS320C20x and TMS320C5x devices: Modify the current AR and ARP as specified when the –p porting switch is used.
BNZ *pma*	√				**Branch if Accumulator ≠ Zero**
BNZ *pma* [, {*ind*} [, *next ARP*]]		√	√	√	If the contents of the accumulator ≠ 0, branch to the specified program-memory address. TMS320C2x devices: Modify the current AR and ARP as specified. TMS320C20x and TMS320C5x devices: Modify the current AR and ARP as specified when the –p porting switch is used.
BSAR [*shift*]				√	**Barrel Shift** In a single cycle, execute a 1- to 16-bit right arithmetic barrel shift of the accumulator. The sign extension is determined by the sign-extension mode bit in ST1.
BV *pma*	√				**Branch on Overflow**
BV *pma* [, {*ind*} [, *next ARP*]]		√	√	√	If the OV flag is set, branch to the specified program-memory address and clear the OV flag. TMS320C2x, TMS320C20x, and TMS320C5x devices: Modify the current AR and ARP as specified. TMS320C20x and TMS320C5x devices: To modify the AR and ARP, use the –p porting switch.

Syntax	1x	2x	2xx	5x	Description
BZ *pma* **BZ** *pma* [, {*ind*} [, *next ARP*]]	√	√	√	√	**Branch if Accumulator = Zero** If the contents of the accumulator = 0, branch to the specified program-memory address. TMS320C2x, TMS320C20x and TMS320C5x devices: Modify the current AR and ARP as specified. TMS320C20x and TMS320C5x devices: To modify the AR and ARP, use the –p porting switch.
CALA	√	√	√		**Call Subroutine Indirect** The contents of the accumulator specify the address of a subroutine. Increment the PC, push the PC onto the stack, then load the 12 (TMS320C1x) or 16 (TMS320C2x/C20x) LSBs of the accumulator into the PC.
CALA[*D*]				√	**Call Subroutine Indirect With Optional Delay** The contents of the accumulator specify the address of a subroutine. Increment the PC and push it onto the stack; then load the 16 LSBs of the accumulator into the PC. If you specify a delayed branch (CALAD), the next two instruction words (two 1-word instructions or one 2-word instruction) are fetched and executed before the call.
CALL *pma* **CALL** *pma* [,{*ind*} [, *next ARP*]]	√	√	√		**Call Subroutine** The contents of the addressed program-memory location specify the address of a subroutine. Increment the PC by 2, push the PC onto the stack, then load the specified program-memory address into the PC. TMS320C2x and TMS320C20x devices: Modify the current AR and ARP as specified.
CALL[*D*] *pma* [, {*ind*} [, *next ARP*]]				√	**Call Unconditionally With Optional Delay** The contents of the addressed program-memory location specify the address of a subroutine. Increment the PC and push the PC onto the stack; then load the specified program-memory address (symbolic or numeric) into the PC. Modify the current AR and ARP as specified. If you specify a delayed branch (CALLD), the next two instruction words (two 1-word instructions or one 2-word instruction) are fetched and executed before the call.

Syntax	1x	2x	2xx	5x	Description
CC *pma, cond$_1$* [*, cond$_2$*] [, ...]			√		**Call Conditionally** If the specified conditions are met, control is passed to the pma. Not all combinations of conditions are meaningful.
CC[*D*] *pma*, *cond$_1$* [*, cond$_2$*] [, ...]				√	**Call Conditionally With Optional Delay** If the specified conditions are met, control is passed to the pma. Not all combinations of conditions are meaningful. If you specify a delayed branch (CCD), the next two instruction words (two 1-word instructions or one 2-word instruction) are fetched and executed before the call.
CLRC *control bit*			√	√	**Clear Control Bit** Set the specified control bit to a logic 0. Maskable interrupts are enabled immediately after the CLRC instruction executes.
CMPL		√	√	√	**Complement Accumulator** Complement the contents of the accumulator (1s complement).
CMPR *CM*		√	√	√	**Compare Auxiliary Register With AR0** Compare the contents of the current auxiliary register to AR0, based on the following cases: If CM = 00_2, test whether AR(ARP) = AR0. If CM = 01_2, test whether AR(ARP) < AR0. If CM = 10_2, test whether AR(ARP) > AR0. If CM = 11_2, test whether AR(ARP) ≠ AR0. If the result is true, load a 1 into the TC status bit; otherwise, load a 0 into the TC bit. The comparison does not affect the tested registers. TMS320C5x devices: Compare the contents of the auxiliary register with the ARCR.
CNFD		√	√	√	**Configure Block as Data Memory** Configure on-chip RAM block B0 as data memory. Block B0 is mapped into data-memory locations 512h–767h. TMS320C5x devices: Block B0 is mapped into data-memory locations 512h–1023h.

Syntax	1x	2x	2xx	5x	Description
CNFP		√	√	√	**Configure Block as Program Memory** Configure on-chip RAM block B0 as program memory. Block B0 is mapped into program-memory locations 65280h–65535h. TMS320C5x devices: Block B0 is mapped into data-memory locations 65024h–65535h.
CONF *2-bit constant*		√			**Configure Block as Program Memory** Configure on-chip RAM block B0/B1/B2/B3 as program memory. For information on the memory mapping of B0/B1/B2/B3, see the *TMS320C2x User's Guide*.
CPL [#*lk,*] *dma* **CPL** [#*lk,*] {*ind*} [*, next ARP*]				√ √	**Compare DBMR or Immediate With Data Value** Compare two quantities: If the two quantities are equal, set the TC bit to 1; otherwise, clear the TC bit.
CRGT				√	**Test for ACC > ACCB** Compare the contents of the ACC with the contents of the ACCB, then load the larger signed value into both registers and modify the carry bit according to the comparison result. If the contents of ACC are greater than or equal to the contents of ACCB, set the carry bit to 1.
CRLT				√	**Test for ACC < ACCB** Compare the contents of the ACC with the contents of the ACCB, then load the smaller signed value into both registers and modify the carry bit according to the comparison result. If the contents of ACC are less than the contents of ACCB, clear the carry bit.
DINT	√	√	√	√	**Disable Interrupts** Disable all interrupts; set the INTM to 1. Maskable interrupts are disabled immediately after the DINT instruction executes. DINT does not disable the unmaskable interrupt $\overline{RS}$; DINT does not affect the IMR.
DMOV *dma* **DMOV** {*ind*} [*, next ARP*]	√ √	√ √	√ √	√ √	**Data Move in Data Memory** Copy the contents of the addressed data-memory location into the next higher address. DMOV moves data only within on-chip RAM blocks. TMS320C2x, TMS320C20x, and TMS320C5x devices: The on-chip RAM blocks are B0 (when configured as data memory), B1, and B2.

Syntax	1x	2x	2xx	5x	Description
EINT	√	√	√	√	**Enable Interrupts** Enable all interrupts; clear the INTM to 0. Maskable interrupts are enabled immediately after the EINT instruction executes.
EXAR				√	**Exchange ACCB With ACC** Exchange the contents of the ACC with the contents of the ACCB.
FORT *1-bit constant*		√			**Format Serial Port Registers** Load the FO with a 0 or a 1. If FO = 0, the registers are configured to receive/transmit 16-bit words. If FO = 1, the registers are configured to receive/transmit 8-bit bytes.
IDLE		√	√	√	**Idle Until Interrupt** Forces an executing program to halt execution and wait until it receives a reset or an interrupt. The device remains in an idle state until it is interrupted.
IDLE2				√	**Idle Until Interrupt—Low-Power Mode** Removes the functional clock input from the internal device; this allows for an extremely low-power mode. The IDLE2 instruction forces an executing program to halt execution and wait until it receives a reset or unmasked interrupt.
IN *dma, PA*	√	√	√	√	**Input Data From Port**
IN {*ind*}*, PA* [*, next ARP*]	√	√	√	√	Read a 16-bit value from one of the external I/O ports into the addressed data-memory location. TMS320C1x devices: This is a 2-cycle instruction. During the first cycle, the port address is sent to address lines A2/PA2–A0/PA0; $\overline{\text{DEN}}$ goes low, strobing in the data that the addressed peripheral places on data bus D15–D0. TMS320C2x devices: The $\overline{\text{IS}}$ line goes low to indicate an I/O access, and the $\overline{\text{STRB}}$, R/$\overline{\text{W}}$, and READY timings are the same as for an external data-memory read. TMS320C20x and TMS320C5x devices: The $\overline{\text{IS}}$ line goes low to indicate an I/O access, and the $\overline{\text{STRB}}$, $\overline{\text{RD}}$, and READY timings are the same as for an external data-memory read.

Syntax	1x	2x	2xx	5x	Description
INTR *K*			√	√	**Soft Interrupt** Transfer program control to the program-memory address specified by *K* (an integer from 0 to 31). This instruction allows you to use your software to execute any interrupt service routine. The interrupt vector locations are spaced apart by two addresses (0h, 2h, 4h, ... , 3Eh), allowing a two-word branch instruction to be placed at each location.
LAC *dma* [, *shift*] **LAC** {*ind*} [, *shift* [, *next ARP*]]	√ √	√ √	√ √	√ √	**Load Accumulator With Shift** Load the contents of the addressed data-memory location into the accumulator. If a shift is specified, left shift the value before loading it into the accumulator. During shifting, low-order bits are zero filled, and high-order bits are sign extended if SXM = 1.
LACB				√	**Load Accumulator With ACCB** Load the contents of the accumulator buffer into the accumulator.
LACC *dma* [, $shift_1$] **LACC** {*ind*} [, $shift_1$ [, *next ARP*]] **LACC** *#lk* [, $shift_2$]		√ √ √	√ √ √	√ √ √	**Load Accumulator With Shift** Load the contents of the addressed data-memory location or the 16-bit constant into the accumulator. If a shift is specified, left shift the value before loading it into the accumulator. During shifting, low-order bits are zero filled, and high-order bits are sign extended if SXM = 1.
LACK *8-bit constant*	√	√	√	√	**Load Accumulator Immediate Short** Load an 8-bit constant into the accumulator. The 24 MSBs of the accumulator are zeroed.
LACL *dma* **LACL** {*ind*} [, *next ARP*] **LACL** *#k*			√ √ √	√ √ √	**Load Low Accumulator and Clear High Accumulator** Load the contents of the addressed data-memory location or zero-extended 8-bit constant into the 16 LSBs of the accumulator. The MSBs of the accumulator are zeroed. The data is treated as a 16-bit unsigned number. TMS320C20x: A constant of 0 clears the contents of the accumulator to 0 with no sign extension.

Syntax	1x	2x	2xx	5x	Description
LACT *dma* **LACT** {*ind*} [, *next ARP*]		√ √	√ √	√ √	**Load Accumulator With Shift Specified by T Register** Left shift the contents of the addressed data-memory location by the value specified in the 4 LSBs of the T register; load the result into the accumulator. If a shift is specified, left shift the value before loading it into the accumulator. During shifting, low-order bits are zero filled, and high-order bits are sign extended if SXM = 1.
LALK *#lk* [, *shift*]		√	√	√	**Load Accumulator Long Immediate With Shift** Load a 16-bit immediate value into the accumulator. If a shift is specified, left shift the constant before loading it into the accumulator. During shifting, low-order bits are zero filled, and high-order bits are sign extended if SXM = 1.
LAMM *dma* **LAMM** {*ind*} [, *next ARP*]				√ √	**Load Accumulator With Memory-Mapped Register** Load the contents of the addressed memory-mapped register into the low word of the accumulator. The 9 MSBs of the data-memory address are cleared, regardless of the current value of DP or the 9 MSBs of AR (ARP).
LAR *AR, dma* **LAR** *AR,* {*ind*} [, *next ARP*] **LAR** *AR, #k* **LAR** *AR, #lk*	√ √	√ √	√ √ √ √	√ √ √ √	**Load Auxiliary Register** TMS320C1x and TMS320C2x devices: Load the contents of the addressed data-memory location into the designated auxiliary register. TMS320C25, TMS320C20x, and TMS320C5x devices: Load the contents of the addressed data-memory location or an 8-bit or 16-bit immediate value into the designated auxiliary register.
LARK *AR, 8-bit constant*	√	√	√	√	**Load Auxiliary Register Immediate Short** Load an 8-bit positive constant into the designated auxiliary register.
LARP *1-bit constant* **LARP** *3-bit constant*	√	√	√	√	**Load Auxiliary Register Pointer** TMS320C1x devices: Load a 1-bit constant into the auxiliary register pointer (specifying AR0 or AR1). TMS320C2x, TMS320C20x, and TMS320C5x devices: Load a 3-bit constant into the auxiliary register pointer (specifying AR0–AR7).

Syntax	1x	2x	2xx	5x	Description
LDP *dma*	√	√	√	√	**Load Data-Memory Page Pointer**
LDP {*ind*} [, *next ARP*]	√	√	√	√	TMS320C1x devices: Load the LSB of the contents of the addressed data-memory location into the DP register. All high-order bits are ignored. DP = 0 defines page 0 (words 0–127), and DP = 1 defines page 1 (words 128–143/255).
LDP #*k*			√	√	TMS320C2x, TMS320C20x, and TMS320C5x devices: Load the 9 LSBs of the addressed data-memory location or a 9-bit immediate value into the DP register. The DP and 7-bit data-memory address are concatenated to form 16-bit data-memory addresses.
LDPK *1-bit constant*	√				**Load Data-Memory Page Pointer Immediate**
LDPK *9-bit constant*		√	√	√	TMS320C1x devices: Load a 1-bit immediate value into the DP register. DP = 0 defines page 0 (words 0–127), and DP = 1 defines page 1 (words 128–143/255).
					TMS320C2x, TMS320C20x, and TMS320C5x devices: Load a 9-bit immediate into the DP register. The DP and 7-bit data-memory address are concatenated to form 16-bit data-memory addresses. DP ≥ 8 specifies external data memory. DP = 4 through 7 specifies on-chip RAM blocks B0 or B1. Block B2 is located in the upper 32 words of page 0.
LMMR *dma, #lk*				√	**Load Memory-Mapped Register**
LMMR {*ind*}, #*lk* [, *next ARP*]				√	Load the contents of the memory-mapped register pointed at by the 7 LSBs of the direct or indirect data-memory value into the long immediate addressed data-memory location. The 9 MSBs of the data-memory address are cleared, regardless of the current value of DP or the 9 MSBs of AR (ARP).
LPH *dma*		√	√	√	**Load High P Register**
LPH {*ind*} [, *next ARP*]		√	√	√	Load the contents of the addressed data-memory location into the 16 MSBs of the P register; the LSBs are not affected.
LRLK *AR, lk*		√	√	√	**Load Auxiliary Register Long Immediate**
					Load a 16-bit immediate value into the designated auxiliary register.
LST *dma*	√	√	√	√	**Load Status Register**
LST {*ind*} [, *next ARP*]	√	√	√	√	Load the contents of the addressed data-memory location into the ST (TMS320C1x) or into ST0 (TMS320C2x/2xx/5x).

Syntax	1x	2x	2xx	5x	Description
LST *#n, dma*		√	√	√	**Load Status Register n**
LST *#n,* {*ind*} [*, next ARP*]		√	√	√	Load the contents of the addressed data-memory location into ST*n*.
LST1 *dma*		√	√	√	**Load ST1**
LST1 {*ind*} [*, next ARP*]		√	√	√	Load the contents of the addressed data-memory location into ST1.
LT *dma*	√	√	√	√	**Load T Register**
LT {*ind*} [*, next ARP*]	√	√	√	√	Load the contents of the addressed data-memory location into the T register (TMS320C1x/2x/2xx) or TREG0 (TMS320C5x).
LTA *dma*	√	√	√	√	**Load T Register and Accumulate Previous Product**
LTA {*ind*} [*, next ARP*]	√	√	√	√	Load the contents of the addressed data-memory location into T register (TMS320C1x/2x/2xx) or TREG0 (TMS320C5x) and add the contents of the P register to the accumulator. TMS320C2x, TMS320C20x, and TMS320C5x devices: Before the add, shift the contents of the P register as specified by the PM status bits.
LTD *dma*	√	√	√	√	**Load T Register, Accumulate Previous Product, and Move Data**
LTD {*ind*} [*, next ARP*]	√	√	√	√	Load the contents of the addressed data-memory location into the T register (TMS320C1x/2x/2xx) or TREG0 (TMS320C5x), add the contents of the P register to the accumulator, and copy the contents of the specified location into the next higher address (both data-memory locations must reside in on-chip data RAM). TMS320C2x, TMS320C20x, and TMS320C5x devices: Before the add, shift the contents of the P register as specified by the PM status bits.
LTP *dma*		√	√	√	**Load T Register, Store P Register in Accumulator**
LTP {*ind*} [*, next ARP*]		√	√	√	Load the contents of the addressed data-memory location into the T register (TMS320C1x/2x/2xx) or TREG0 (TMS320C5x). Store the contents of the product register into the accumulator.
LTS *dma*		√	√	√	**Load T Register, Subtract Previous Product**
LTS {*ind*} [*, next ARP*]		√	√	√	Load the contents of the addressed data-memory location into the T register (TMS320C1x/2x/2xx) or TREG0 (TMS320C5x). Shift the contents of the product register as specified by the PM status bits, and subtract the result from the accumulator.

Syntax	1x	2x	2xx	5x	Description
MAC *pma, dma*		√	√	√	**Multiply and Accumulate**
MAC *pma,* {*ind*} [*, next ARP*]		√	√	√	Multiply a data-memory value by a program-memory value and add the previous product (shifted as specified by the PM status bits) to the accumulator.
MACD *dma, pma*		√	√	√	**Multiply and Accumulate With Data Move**
MACD *pma,* {*ind*} [*, next ARP*]		√	√	√	Multiply a data-memory value by a program-memory value and add the previous product (shifted as specified by the PM status bits) to the accumulator. If the data-memory address is in on-chip RAM block B0, B1, or B2, copy the contents of the address to the next higher address.
MADD *dma*				√	**Multiply and Accumulate With Data Move and Dynamic Addressing**
MADD {*ind*} [*, next ARP*]				√	Multiply a data-memory value by a program-memory value and add the previous product (shifted as defined by the PM status bits) into the accumulator. The program-memory address is contained in the BMAR; this allows for dynamic addressing of coefficient tables. MADD functions the same as MADS, with the addition of data move for on-chip RAM blocks.
MADS *dma*				√	**Multiply and Accumulate With Dynamic Addressing**
MADS {*ind*} [*, next ARP*]				√	Multiply a data-memory value by a program-memory value and add the previous product (shifted as defined by the PM status bits) into the accumulator. The program-memory address is contained in the BMAR; this allows for dynamic addressing of coefficient tables.
MAR *dma*	√	√	√	√	**Modify Auxiliary Register**
MAR {*ind*} [*, next ARP*]	√	√	√	√	Modify the current AR or ARP as specified. MAR acts as NOP in indirect addressing mode.
MPY *dma*	√	√	√	√	**Multiply**
MPY {*ind*} [*, next ARP*]	√	√	√	√	TMS320C1x and TMS320C2x devices: Multiply the contents of the T register by the contents of the addressed data-memory location; place the result in the P register. TMS320C20x and TMS320C5x devices: Multiply the contents of the T register (TMS320C20x) or TREG0 (TMS320C5x) by the contents of the addressed data-memory location or a 13-bit or 16-bit immediate value; place the result in the P register.
MPY *#k*			√	√	
MPY *#lk*			√	√	

Syntax	1x	2x	2xx	5x	Description
MPYA *dma*		√	√	√	**Multiply and Accumulate Previous Product**
MPYA {*ind*} [, *next ARP*]		√	√	√	Multiply the contents of the T register (TMS320C2x/2xx) or TREG0 (TMS320C5x) by the contents of the addressed data-memory location; place the result in the P register. Add the previous product (shifted as specified by the PM status bits) to the accumulator.
MPYK *13-bit constant*	√	√	√	√	**Multiply Immediate**
					Multiply the contents of the T register (TMS320C2x/2xx) or TREG0 (TMS320C5x) by a signed 13-bit constant; place the result in the P register.
MPYS *dma*		√	√	√	**Multiply and Subtract Previous Product**
MPYS {*ind*} [, *next ARP*]		√	√	√	Multiply the contents of the T register (TMS320C2x/2xx) or TREG0 (TMS320C5x) by the contents of the addressed data-memory location; place the result in the P register. Subtract the previous product (shifted as specified by the PM status bits) from the accumulator.
MPYU *dma*		√	√	√	**Multiply Unsigned**
MPYU {*ind*} [, *next ARP*]		√	√	√	Multiply the unsigned contents of the T register (TMS320C2x/2xx) or TREG0 (TMS320C5x) by the unsigned contents of the addressed data-memory location; place the result in the P register.
NEG		√	√	√	**Negate Accumulator**
					Negate (2s complement) the contents of the accumulator.
NMI			√	√	**Nonmaskable Interrupt**
					Force the program counter to the nonmaskable interrupt vector location 24h. NMI has the same effect as a hardware nonmaskable interrupt.
NOP	√	√	√	√	**No Operation**
					Perform no operation.
NORM		√	√	√	**Normalize Contents of Accumulator**
NORM {*ind*}		√	√	√	Normalize a signed number in the accumulator.
OPL [#*lk*,] *dma*				√	**OR With DBMR or Long Immediate**
OPL [#*lk*,] {*ind*} [, *next ARP*]				√	If a long immediate is specified, OR it with the value at the specified data-memory location; otherwise, the second operand of the OR operation is the contents of the DBMR. The result is written back into the data-memory location previously holding the first operand.

Syntax	1x	2x	2xx	5x	Description
OR *dma*	√	√	√	√	**OR With Accumulator**
OR {*ind*} [, *next ARP*]	√	√	√	√	TMS320C1x and TMS320C2x devices: OR the 16 LSBs of the accumulator with the contents of the addressed data-memory location. The 16 MSBs of the accumulator are ORed with 0s.
OR #*lk* [, *shift*]			√	√	TMS320C20x and TMS320C5x devices: OR the 16 LSBs of the accumulator or a 16-bit immediate value with the contents of the addressed data-memory location. If a shift is specified, left-shift before ORing. Low-order bits below and high-order bits above the shifted value are treated as 0s.
ORB				√	**OR ACCB With Accumulator** OR the contents of the ACCB with the contents of the accumulator. ORB places the result in the accumulator.
ORK #*lk* [, *shift*]		√	√	√	**OR Immediate With Accumulator with Shift** OR a 16-bit immediate value with the contents of the accumulator. If a shift is specified, left-shift the constant before ORing. Low-order bits below and high-order bits above the shifted value are treated as 0s.
OUT *dma, PA*	√	√	√	√	**Output Data to Port**
OUT {*ind*}, *PA* [, *next ARP*]	√	√	√	√	Write a 16-bit value from a data-memory location to the specified I/O port. TMS320C1x devices: The first cycle of this instruction places the port address onto address lines A2/PA2–A0/PA0. During the same cycle, $\overline{WE}$ goes low and the data word is placed on the data bus D15–D0. TMS320C2x, TMS320C20x, and TMS320C5x devices: The $\overline{IS}$ line goes low to indicate an I/O access; the $\overline{STRB}$, R/$\overline{W}$, and READY timings are the same as for an external data-memory write.
PAC	√	√	√	√	**Load Accumulator With P Register** Load the contents of the P register into the accumulator. TMS320C2x, TMS320C20x, and TMS320C5x devices: Before the load, shift the P register as specified by the PM status bits.

Syntax	1x	2x	2xx	5x	Description
POP	√	√	√	√	**Pop Top of Stack to Low Accumulator** Copy the contents of the top of the stack into the 12 (TMS320C1x) or 16 (TMS320C2x/2xx/5x) LSBs of the accumulator and then pop the stack one level. The MSBs of the accumulator are zeroed.
POPD *dma* **POPD** {*ind*} [, *next ARP*]		√ √	√ √	√ √	**Pop Top of Stack to Data Memory** Transfer the value on the top of the stack into the addressed data-memory location and then pop the stack one level.
PSHD *dma* **PSHD** {*ind*} [, *next ARP*]		√ √	√ √	√ √	**Push Data-Memory Value Onto Stack** Copy the addressed data-memory location onto the top of the stack. The stack is pushed down one level before the value is copied.
PUSH	√	√	√	√	**Push Low Accumulator Onto Stack** Copy the contents of the 12 (TMS320C1x) or 16 (TMS320C2x/2xx/5x) LSBs of the accumulator onto the top of the hardware stack. The stack is pushed down one level before the value is copied.
RC		√	√	√	**Reset Carry Bit** Reset the C status bit to 0.
RET	√	√	√		**Return From Subroutine** Copy the contents of the top of the stack into the PC and pop the stack one level.
RET[*D*]				√	**Return From Subroutine With Optional Delay** Copy the contents of the top of the stack into the PC and pop the stack one level. If you specify a delayed branch (RETD), the next two instruction words (two 1-word instructions or one 2-word instruction) are fetched and executed before the return.
RETC $cond_1$ [, $cond_2$] [, ...]			√		**Return Conditionally** If the specified conditions are met, RETC performs a standard return. Not all combinations of conditions are meaningful.

Syntax	1x	2x	2xx	5x	Description
RETC[*D*] *cond*$_1$ [, *cond*$_2$] [, ...]				√	**Return Conditionally With Optional Delay** If the specified conditions are met, RETC performs a standard return. Not all combinations of conditions are meaningful. If you specify a delayed branch (RETCD), the next two instruction words (two 1-word instructions or one 2-word instruction) are fetched and executed before the return.
RETE				√	**Enable Interrupts and Return From Interrupt** Copy the contents of the top of the stack into the PC and pop the stack one level. RETE automatically clears the global interrupt enable bit and pops the shadow registers (stored when the interrupt was taken) back into their corresponding strategic registers. The following registers are shadowed: ACC, ACCB, PREG, ST0, ST1, PMST, ARCR, INDX, TREG0, TREG1, TREG2.
RETI				√	**Return From Interrupt** Copy the contents of the top of the stack into the PC and pop the stack one level. RETI also pops the values in the shadow registers (stored when the interrupt was taken) back into their corresponding strategic registers. The following registers are shadowed: ACC, ACCB, PREG, ST0, ST1, PMST, ARCR, INDX, TREG0, TREG1, TREG2.
RFSM		√			**Reset Serial Port Frame Synchronization Mode** Reset the FSM status bit to 0.
RHM		√		√	**Reset Hold Mode** Reset the HM status bit to 0.
ROL		√	√	√	**Rotate Accumulator Left** Rotate the accumulator left one bit.
ROLB				√	**Rotate ACCB and Accumulator Left** Rotate the ACCB and the accumulator left by one bit; this results in a 65-bit rotation.
ROR		√	√	√	**Rotate Accumulator Right** Rotate the accumulator right one bit.
RORB				√	**Rotate ACCB and Accumulator Right** Rotate the ACCB and the accumulator right one bit; this results in a 65-bit rotation.

Syntax	1x	2x	2xx	5x	Description
ROVM	√	√	√	√	**Reset Overflow Mode** Reset the OVM status bit to 0; this disables overflow mode.
RPT *dma* **RPT** {*ind*} [, *next ARP*] **RPT** *#k* **RPT** *#lk*		√ √	√ √ √ √	√ √ √ √	**Repeat Next Instruction** TMS320C2x devices: Load the 8 LSBs of the addressed value into the RPTC; the instruction following RPT is executed the number of times indicated by RPTC + 1. TMS320C20x and TMS320C5x devices: Load the 8 LSBs of the addressed value or an 8-bit or 16-bit immediate value into the RPTC; the instruction following RPT is repeated *n* times, where *n* is RPTC+1.
RPTB *pma*				√	**Repeat Block** RPTB repeats a block of instructions the number of times specified by the memory-mapped BRCR without any penalty for looping. The BRCR must be loaded before RPTB is executed.
RPTK *#k*		√	√	√	**Repeat Instruction as Specified by Immediate Value** Load the 8-bit immediate value into the RPTC; the instruction following RPTK is executed the number of times indicated by RPTC + 1.
RPTZ *#lk*				√	**Repeat Preceded by Clearing the Accumulator and P Register** Clear the accumulator and product register and repeat the instruction following RPTZ *n* times, where *n* = *lk* +1.
RSXM		√	√	√	**Reset Sign-Extension Mode** Reset the SXM status bit to 0; this suppresses sign extension on shifted data values for the following arithmetic instructions: ADD, ADDT, ADLK, LAC, LACT, LALK, SBLK, SUB, and SUBT.
RTC		√	√	√	**Reset Test/Control Flag** Reset the TC status bit to 0.
RTXM		√			**Reset Serial Port Transmit Mode** Reset the TXM status bit to 0; this configures the serial port transmit section in a mode where it is controlled by an FSX.
RXF		√	√	√	**Reset External Flag** Reset XF pin and the XF status bit to 0.

Syntax	1x	2x	2xx	5x	Description
SACB				√	**Store Accumulator in ACCB** Copy the contents of the accumulator into the ACCB.
SACH *dma* [, *shift*] **SACH** {*ind*} [, *shift* [, *next ARP*]]	√ √	√ √	√ √	√ √	**Store High Accumulator With Shift** Copy the contents of the accumulator into a shifter. Shift the entire contents 0, 1, or 4 bits (TMS320C1x) or from 0 to 7 bits (TMS320C2x/2xx/5x), and then copy the 16 MSBs of the shifted value into the addressed data-memory location. The accumulator is not affected.
SACL *dma* **SACL** *dma* [, *shift*] **SACL** {*ind*} [, *shift* [, *next ARP*]]	√ √	 √ √	 √ √	 √ √	**Store Low Accumulator With Shift** TMS320C1x devices: Store the 16 LSBs of the accumulator into the addressed data-memory location. A shift value of 0 must be specified if the ARP is to be changed. TMS320C2x, TMS320C20x, and TMS320C5x devices: Store the 16 LSBs of the accumulator into the addressed data-memory location. If a shift is specified, shift the contents of the accumulator before storing. Shift values are 0, 1, or 4 bits (TMS320C20) or from 0 to 7 bits (TMS320C2x/2xx/5x).
SAMM *dma* **SAMM** {*ind*} [, *next ARP*]				√ √	**Store Accumulator in Memory-Mapped Register** Store the low word of the accumulator in the addressed memory-mapped register. The upper 9 bits of the data address are cleared, regardless of the current value of DP or the 9 MSBs of AR (ARP).
SAR *AR, dma* **SAR** *AR,* {*ind*} [*, next ARP*]	√ √	√ √	√ √	√ √	**Store Auxiliary Register** Store the contents of the specified auxiliary register in the addressed data-memory location.
SATH				√	**Barrel-Shift Accumulator as Specified by T Register 1** If bit 4 of TREG1 is a 1, barrel-shift the accumulator right by 16 bits; otherwise, the accumulator is unaffected.
SATL				√	**Barrel-Shift Low Accumulator as Specified by T Register 1** Barrel-shift the accumulator right by the value specified in the 4 LSBs of TREG1.
SBB				√	**Subtract ACCB From Accumulator** Subtract the contents of the ACCB from the accumulator. The result is stored in the accumulator; the accumulator buffer is not affected.

Syntax	1x	2x	2xx	5x	Description
SBBB				√	**Subtract ACCB From Accumulator With Borrow** Subtract the contents of the ACCB and the logical inversion of the carry bit from the accumulator. The result is stored in the accumulator; the accumulator buffer is not affected. Clear the carry bit if the result generates a borrow.
SBLK *#lk* [*, shift*]		√	√	√	**Subtract From Accumulator Long Immediate With Shift** Subtract the immediate value from the accumulator. If a shift is specified, left shift the value before subtracting. During shifting, low-order bits are zero filled, and high-order bits are sign extended if SXM = 1.
SBRK *#k*		√	√	√	**Subtract From Auxiliary Register Short Immediate** Subtract the 8-bit immediate value from the designated auxiliary register.
SC		√	√	√	**Set Carry Bit** Set the C status bit to 1.
SETC *control bit*			√	√	**Set Control Bit** Set the specified control bit to a logic 1. Maskable interrupts are disabled immediately after the SETC instruction executes.
SFL		√	√	√	**Shift Accumulator Left** Shift the contents of the accumulator left one bit.
SFLB				√	**Shift ACCB and Accumulator Left** Shift the concatenation of the accumulator and the ACCB left one bit. The LSB of the ACCB is cleared to 0, and the MSB of the ACCB is shifted into the carry bit.
SFR		√	√	√	**Shift Accumulator Right** Shift the contents of the accumulator right one bit. If SXM = 1, SFR produces an arithmetic right shift. If SXM = 0, SFR produces a logic right shift.
SFRB				√	**Shift ACCB and Accumulator Right** Shift the concatenation of the accumulator and the ACCB right 1 bit. The LSB of the ACCB is shifted into the carry bit. If SXM = 1, SFRB produces an arithmetic right shift. If SXM = 0, SFRB produces a logic right shift.
SFSM		√			**Set Serial Port Frame Synchronization Mode** Set the FSM status bit to 1.

Syntax	1x	2x	2xx	5x	Description
SHM		√		√	**Set Hold Mode** Set the HM status bit to 1.
SMMR *dma, #lk* **SMMR** {*ind*}, *#lk* [, *next ARP*]				√ √	**Store Memory-Mapped Register** Store the memory-mapped register value, pointed at by the 7 LSBs of the data-memory address, into the long immediate addressed data-memory location. The 9 MSBs of the data-memory address of the memory-mapped register are cleared, regardless of the current value of DP or the upper 9 bits of AR(ARP).
SOVM	√	√	√	√	**Set Overflow Mode** Set the OVM status bit to 1; this enables overflow mode. (The ROVM instruction clears OVM.)
SPAC	√	√	√	√	**Subtract P Register From Accumulator** Subtract the contents of the P register from the contents of the accumulator. TMS320C2x, TMS320C20x, and TMS320C5x devices: Before the subtraction, shift the contents of the P register as specified by the PM status bits.
SPH *dma* **SPH** {*ind*} [, *next ARP*]		√ √	√ √	√ √	**Store High P Register** Store the high-order bits of the P register (shifted as specified by the PM status bits) at the addressed data-memory location.
SPL *dma* **SPL** {*ind*} [, *next ARP*]		√ √	√ √	√ √	**Store Low P Register** Store the low-order bits of the P register (shifted as specified by the PM status bits) at the addressed data-memory location.
SPLK *#lk, dma* **SPLK** *#lk,* {*ind*} [, *next ARP*]			√	√ √	**Store Parallel Long Immediate** Write a full 16-bit pattern into a memory location. The parallel logic unit (PLU) supports this bit manipulation independently of the ALU, so the accumulator is unaffected.
SPM *2-bit constant*		√	√	√	**Set P Register Output Shift Mode** Copy a 2-bit immediate value into the PM field of ST1. This controls shifting of the P register as shown below: PM = 00_2 Multiplier output is not shifted. PM = 01_2 Multiplier output is left shifted one place and zero filled. PM = 10_2 Multiplier output is left shifted four places and zero filled. PM = 11_2 Multiplier output is right shifted six places and sign extended; the LSBs are lost.

Syntax	1x	2x	2xx	5x	Description
SQRA *dma*		√	√	√	**Square and Accumulate Previous Product**
SQRA {*ind*} [, *next ARP*]		√	√	√	Add the contents of the P register (shifted as specified by the PM status bits) to the accumulator. Then load the contents of the addressed data-memory location into the T register (TMS320C2x/2xx) or TREG0 (TMS320C5x), square the value, and store the result in the P register.
SQRS *dma*		√	√	√	**Square and Subtract Previous Product**
SQRS {*ind*} [, *next ARP*]		√	√	√	Subtract the contents of the P register (shifted as specified by the PM status bits) to the accumulator. Then load the contents of the addressed data-memory location into the T register (TMS320C2x/2xx) or TREG0 (TMS320C5x), square the value, and store the result in the P register.
SST *dma*	√	√	√	√	**Store Status Register**
SST {*ind*} [, *next ARP*]	√	√	√	√	Store the contents of the ST (TMS320C1x) or ST0 (TMS320C2x/2xx/5x) in the addressed data-memory location.
SST #*n, dma*			√	√	**Store Status Register n**
SST #*n,* {*ind*} [, *next ARP*]			√	√	Store ST*n* in data memory.
SST1 *dma*		√	√	√	**Store Status Register ST1**
SST1 {*ind*} [, *next ARP*]		√	√	√	Store the contents of ST1 in the addressed data-memory location.
SSXM		√	√	√	**Set Sign-Extension Mode**
					Set the SXM status bit to 1; this enables sign extension.
STC		√	√	√	**Set Test/Control Flag**
					Set the TC flag to 1.
STXM		√			**Set Serial Port Transmit Mode**
					Set the TXM status bit to 1.

Syntax	1x	2x	2xx	5x	Description
SUB *dma* [, *shift*]	√	√	√	√	**Subtract From Accumulator With Shift**
SUB {*ind*} [, *shift* [, *next ARP*]]	√	√	√	√	TMS320C1x and TMS320C2x devices: Subtract the contents of the addressed data-memory location from the accumulator. If a shift is specified, left shift the value before subtracting. During shifting, low-order bits are zero filled, and high-order bits are sign extended if SXM = 1.
SUB #*k*			√	√	
SUB #*lk* [, $shift_2$]			√	√	
					TMS320C20x and TMS320C5x devices: Subtract the contents of the addressed data-memory location or an 8- or 16-bit constant from the accumulator. If a shift is specified, left shift the data before subtracting. During shifting, low-order bits are zero filled, and high-order bits are sign extended if SXM = 1.
SUBB *dma*		√	√	√	**Subtract From Accumulator With Borrow**
SUBB {*ind*} [, *next ARP*]		√	√	√	Subtract the contents of the addressed data-memory location and the value of the carry bit from the accumulator. The carry bit is affected in the normal manner.
SUBC *dma*	√	√	√	√	**Conditional Subtract**
SUBC {*ind*} [, *next ARP*]	√	√	√	√	Perform conditional subtraction. SUBC can be used for division.
SUBH *dma*	√	√	√	√	**Subtract From High Accumulator**
SUBH {*ind*} [, *next ARP*]	√	√	√	√	Subtract the contents of the addressed data-memory location from the 16 MSBs of the accumulator. The 16 LSBs of the accumulator are not affected.
SUBK #*k*		√	√	√	**Subtract From Accumulator Short Immediate**
					Subtract an 8-bit immediate value from the accumulator. The data is treated as an 8-bit positive number; sign extension is suppressed.
SUBS *dma*	√	√	√	√	**Subtract From Low Accumulator With Sign Extension Suppressed**
SUBS {*ind*} [, *next ARP*]	√	√	√	√	Subtract the contents of the addressed data-memory location from the accumulator. The data is treated as a 16-bit unsigned number; sign extension is suppressed.

Syntax	1x	2x	2xx	5x	Description
SUBT *dma* **SUBT** {*ind*} [, *next ARP*]		√ √	√ √	√ √	**Subtract From Accumulator With Shift Specified by T Register** Left shift the data-memory value as specified by the 4 LSBs of the T register (TMS320C2x/2xx) or TREG1 (TMS320C5x), and subtract the result from the accumulator. If a shift is specified, left shift the data-memory value before subtracting. During shifting, low-order bits are zero filled, and high-order bits are sign extended if SXM = 1.
SXF		√	√	√	**Set External Flag** Set the XF pin and the XF status bit to 1.
TBLR *dma* **TBLR** {*ind*} [, *next ARP*]	√ √	√ √	√ √	√ √	**Table Read** Transfer a word from program memory to a data-memory location. The program-memory address is in the 12 (TMS320C1x) or 16 (TMS320C2x/2xx/5x) LSBs of the accumulator.
TBLW *dma* **TBLW** {*ind*} [, *next ARP*]	√ √	√ √	√ √	√ √	**Table Write** Transfer a word from data-memory to a program-memory location. The program-memory address is in the 12 (TMS320C1x) or 16 (TMS320C2x/2xx/5x) LSBs of the accumulator.
TRAP		√	√	√	**Software Interrupt** The TRAP instruction is a software interrupt that transfers program control to program-memory address 30h (TMS320C2x) or 22h (TMS320C20x/5x) and pushes the PC + 1 onto the hardware stack. The instruction at address 30h or 22h may contain a branch instruction to transfer control to the TRAP routine. Putting the PC + 1 on the stack enables an RET instruction to pop the return PC.
XC *n, cond*$_1$ [, *cond*$_2$] [, ...]				√	**Execute Conditionally** Execute conditionally the next *n* instruction words where $1 \leq n \leq 2$. Not all combinations of conditions are meaningful.

Syntax	1x	2x	2xx	5x	Description
XOR *dma* **XOR** {*ind*} [, *next ARP*] **XOR** #*lk* [, *shift*]	√ √	√ √	√ √ √	√ √ √	**Exclusive-OR With Accumulator** TMS320C1x and TMS320C2x devices: Exclusive-OR the contents of the addressed data-memory location with 16 LSBs of the accumulator. The MSBs are not affected. TMS320C20x and TMS320C5x devices: Exclusive-OR the contents of the addressed data-memory location or a 16-bit immediate value with the accumulator. If a shift is specified, left shift the value before performing the exclusive-OR operation. Low-order bits below and high-order bits above the shifted value are treated as 0s.
XORB				√	**Exclusive-OR of ACCB With Accumulator** Exclusive-OR the contents of the accumulator with the contents of the ACCB. The results are placed in the accumulator.
XORK #*lk* [, *shift*]		√	√	√	**Exclusive-OR Immediate With Accumulator With Shift** Exclusive-OR a 16-bit immediate value with the accumulator. If a shift is specified, left shift the value before peforming the exclusive-OR operation. Low-order bits below and high-order bits above the shifted value are treated as 0s.
XPL [#*lk*,] *dma* **XPL** [#*lk*,] {*ind*} [, *next ARP*]				√ √	**Exclusive-OR of Long Immediate or DBMR With Addressed Data-Memory Value** If a long immediate value is specified, exclusive OR it with the addressed data-memory value; otherwise, exclusive OR the DBMR with the addressed data-memory value. Write the result back to the data-memory location. The accumulator is not affected.
ZAC	√	√	√	√	**Zero Accumulator** Clear the contents of the accumulator to 0.
ZALH *dma* **ZALH** {*ind*} [, *next ARP*]	√ √	√ √	√ √	√ √	**Zero Low Accumulator and Load High Accumulator** Clear the 16 LSBs of the accumulator to 0 and load the contents of the addressed data-memory location into the 16 MSBs of the accumulator.

Syntax	1x	2x	2xx	5x	Description
ZALR *dma* **ZALR** {*ind*} [, *next ARP*]		√ √	√ √	√ √	**Zero Low Accumulator, Load High Accumulator With Rounding** Load the contents of the addressed data-memory location into the 16 MSBs of the accumulator. The value is rounded by 1/2 LSB; that is, the 15 LSBs of the accumulator (0–14) are cleared and bit 15 is set to 1.
ZALS *dma* **ZALS** {*ind*} [, *next ARP*]	√ √	√ √	√ √	√ √	**Zero Accumulator, Load Low Accumulator With Sign Extension Suppressed** Load the contents of the addressed data-memory location into the 16 LSBs of the accumulator. The 16 MSBs are zeroed. The data is treated as a 16-bit unsigned number.
ZAP				√	**Zero the Accumulator and Product Register** The accumulator and product register are zeroed. The ZAP instruction speeds up the preparation for a repeat multiply/accumulate.
ZPR				√	**Zero the Product Register** The product register is cleared.

Appendix B

Submitting ROM Codes to TI

The size of a printed circuit board is a consideration in many DSP applications. To make full use of the board space, Texas Instruments offers a ROM code option that reduces the chip count and provides a single-chip solution. This option allows you to use a code-customized processor for a specific application while taking advantage of:

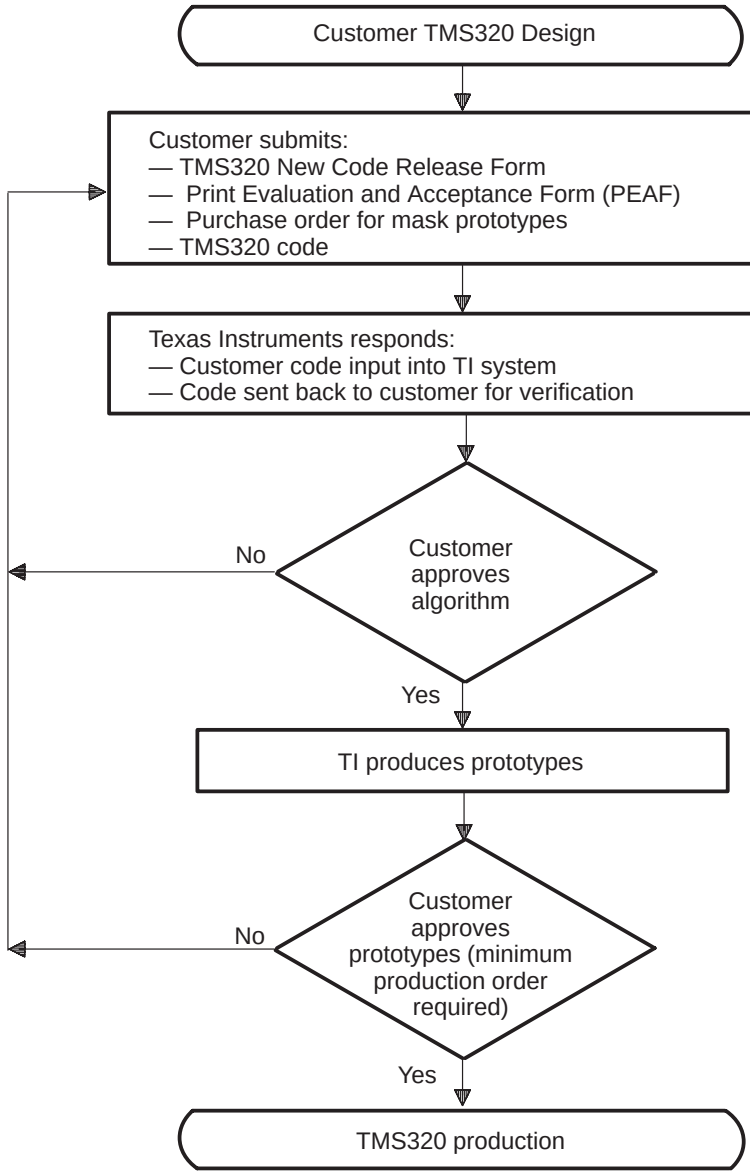
- ❏ Greater memory expansion
- ❏ Lower system cost
- ❏ Less hardware and wiring
- ❏ Smaller PCB

If a routine or algorithm is used often, it 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 some ROM-coded TMS320 DSP devices when accesses to either on-chip or off-chip memory are 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 code can be submitted 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 can once again be used in the microprocessor mode. This shortens the field-upgrade time and prevents the possibility of inventory obsolescence.

Figure B–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 final delivery.

Figure B–1. TMS320 ROM Code Procedural Flow Chart



The TMS320 ROM code may be submitted in one of the following forms:

- Attachment to e–mail
- 3-1/2-in floppy: COFF format from macro-assembler/linker

When code is submitted to TI for masking, the code is reformatted to accommodate the TI mask-generation system. System-level verification by the customer is, therefore, necessary to ensure the reformatting remains transparent and does 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) and the addition of data in the reserved locations of the ROM for device ROM test. 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 that states:

> 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 that states:

> 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 C

Design Considerations for Using the XDS510 Emulator

This appendix assists you in meeting the design requirements of the Texas Instruments XDS510™ emulator for IEEE-1149.1 designs and discusses the XDS510 cable (manufacturing part number 2617698-0001). This cable is identified by a label on the cable pod marked *JTAG 3/5V* and supports both standard 3-V and 5-V target system power inputs.

The term *JTAG,* as used in this book, refers to TI scan-based emulation, which is based on the IEEE 1149.1 standard.

For more information concerning the IEEE 1149.1 standard, contact IEEE Customer Service:

Address: IEEE Customer Service
445 Hoes Lane, PO Box 1331
Piscataway, NJ 08855-1331

Phone: (800) 678–IEEE in the US and Canada
(908) 981–1393 outside the US and Canada

FAX: (908) 981–9667 Telex: 833233

Topic **Page**

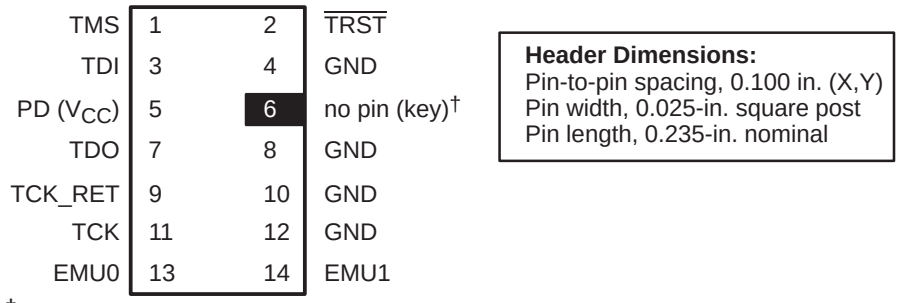
C.1 Designing Your Target System's Emulator Connector (14-Pin Header)

JTAG target devices support emulation through a dedicated emulation port. This port is accessed directly by the emulator and provides emulation functions that are a superset of those specified by IEEE 1149.1. To communicate with the emulator, *your target system must have a 14-pin header* (two rows of seven pins) with the connections that are shown in Figure C–1. Table C–1 describes the emulation signals.

Although you can use other headers, the recommended unshrouded, straight header has these DuPont connector systems part numbers:

- 65610–114
- 65611–114
- 67996–114
- 67997–114

Figure C–1. 14-Pin Header Signals and Header Dimensions



† While the corresponding female position on the cable connector is plugged to prevent improper connection, the cable lead for pin 6 is present in the cable and is grounded, as shown in the schematics and wiring diagrams in this appendix.

Table C–1. 14-Pin Header Signal Descriptions

Signal	Description	Emulator† State	Target† State
EMU0	Emulation pin 0	I	I/O
EMU1	Emulation pin 1	I	I/O
GND	Ground		
PD(V_{CC})	Presence detect. Indicates that the emulation cable is connected and that the target is powered up. PD must be tied to V_{CC} in the target system.	I	O
TCK	Test clock. TCK is a 10.368-MHz clock source from the emulation cable pod. This signal can be used to drive the system test clock.	O	I
TCK_RET	Test clock return. Test clock input to the emulator. May be a buffered or unbuffered version of TCK.	I	O
TDI	Test data input	O	I
TDO	Test data output	I	O
TMS	Test mode select	O	I
$\overline{\text{TRST}}$‡	Test reset	O	I

† I = input; O = output

‡ Do not use pullup resistors on $\overline{\text{TRST}}$: it has an internal pulldown device. In a low-noise environment, $\overline{\text{TRST}}$ can be left floating. In a high-noise environment, an additional pulldown resistor may be needed. (The size of this resistor should be based on electrical current considerations.)

C.2 Bus Protocol

The IEEE 1149.1 specification covers the requirements for the test access port (TAP) bus slave devices and provides certain rules, summarized as follows:

- The TMS and TDI inputs are sampled on the rising edge of the TCK signal of the device.
- The TDO output is clocked from the falling edge of the TCK signal of the device.

When these devices are daisy-chained together, the TDO of one device has approximately a half TCK cycle setup time before the next device's TDI signal. This timing scheme minimizes race conditions that would occur if both TDO and TDI were timed from the same TCK edge. The penalty for this timing scheme is a reduced TCK frequency.

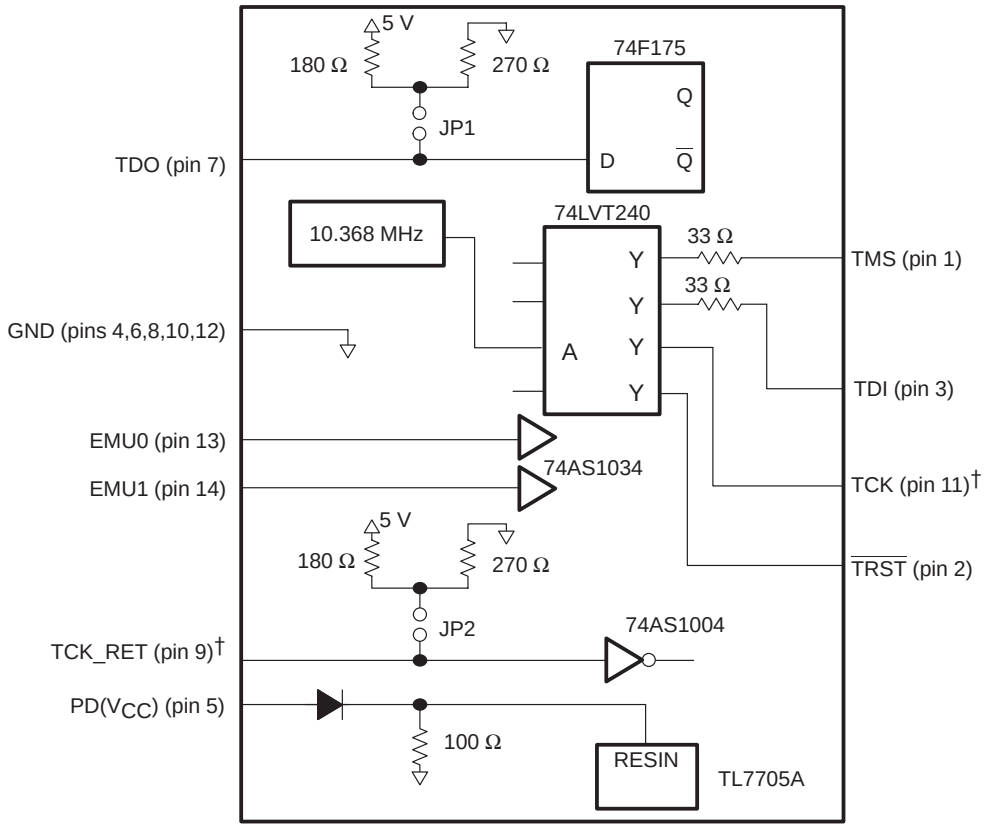
The IEEE 1149.1 specification does not provide rules for bus master (emulator) devices. Instead, it states that the device expects a bus master to provide bus slave compatible timings. The XDS510 provides timings that meet the bus slave rules.

C.3 Emulator Cable Pod

Figure C–2 shows a portion of the emulator cable pod. The functional features of the pod are:

- ❑ TDO and TCK_RET can be parallel-terminated inside the pod if required by the application. By default, these signals are not terminated.
- ❑ TCK is driven with a 74LVT240 device. Because of the high-current drive (32-mA I_{OL}/I_{OH}), this signal can be parallel-terminated. If TCK is tied to TCK_RET, you can use the parallel terminator in the pod.
- ❑ TMS and TDI can be generated from the falling edge of TCK_RET, according to the IEEE 1149.1 bus slave device timing rules.
- ❑ TMS and TDI are series terminated to reduce signal reflections.
- ❑ A 10.368-MHz test clock source is provided. You can also provide your own test clock for greater flexibility.

Figure C–2. Emulator Cable Pod Interface



† The emulator pod uses TCK_RET as its clock source for internal synchronization. TCK is provided as an optional target system test clock source.

C.4 Emulator Cable Pod Signal Timing

Figure C–3 shows the signal timings for the emulator cable pod. Table C–2 defines the timing parameters illustrated in the figure. These timing parameters are calculated from values specified in the standard data sheets for the emulator and cable pod and are for reference only. Texas Instruments does not test or guarantee these timings.

The emulator pod uses TCK_RET as its clock source for internal synchronization. TCK is provided as an optional target system test clock source.

Figure C–3. Emulator Cable Pod Timings

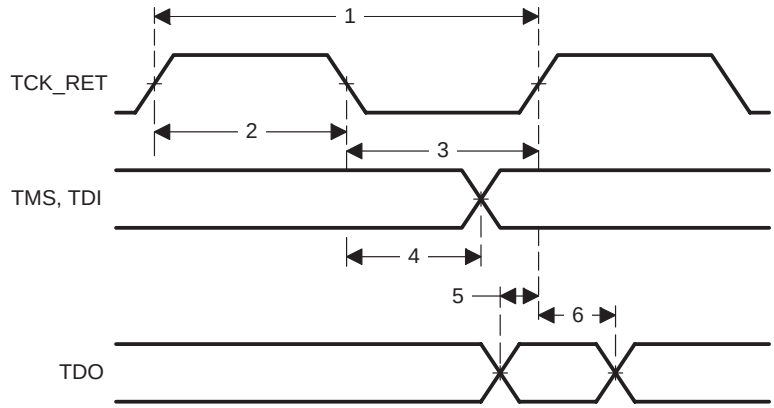


Table C–2. Emulator Cable Pod Timing Parameters

No.	Parameter	Description	Min	Max	Unit
1	$t_{c(TCK)}$	Cycle time, TCK_RET	35	200	ns
2	$t_{w(TCKH)}$	Pulse duration, TCK_RET high	15		ns
3	$t_{w(TCKL)}$	Pulse duration, TCK_RET low	15		ns
4	$t_{d(TMS)}$	Delay time, TMS or TDI valid for TCK_RET low	6	20	ns
5	$t_{su(TDO)}$	Setup time, TDO to TCK_RET high	3		ns
6	$t_{h(TDO)}$	Hold time, TDO from TCK_RET high	12		ns

C.5 Emulation Timing Calculations

The examples in this section help you calculate emulation timings in your system. For actual target timing parameters, see the appropriate data sheet for the device you are emulating.

The examples use the following assumptions:

$t_{su(TTMS)}$	Setup time, target TMS or TDI to TCK high	10 ns
$t_{d(TTDO)}$	Delay time, target TDO from TCK low	15 ns
$t_{d(bufmax)}$	Delay time, target buffer maximum	10 ns
$t_{d(bufmin)}$	Delay time, target buffer minimum	1 ns
$t_{bufskew}$	Skew time, target buffer between two devices in the same package: $[t_{d(bufmax)} - t_{d(bufmin)}] \times 0.15$	1.35 ns
$t_{TCKfactor}$	Duty cycle, assume a 40/60% duty cycle clock	0.4 (40%)

Also, the examples use the following values from Table C–2 on page C-6:

$t_{d(TMSmax)}$	Delay time, emulator TMS or TDI from TCK_RET low, maximum	20 ns
$t_{su(TDOmin)}$	Setup time, TDO to emulator TCK_RET high, minimum	3 ns

There are two key timing paths to consider in the emulation design:

- The TCK_RET-to-TMS or TDI path, called $t_{pd(TCK\_RET\text{-}TMS/TDI)}$ (propagation delay time)
- The TCK_RET-to-TDO path, called $t_{pd(TCK\_RET\text{-}TDO)}$

In the examples, the worst-case path delay is calculated to determine the maximum system test clock frequency.

Example C–1. Key Timing for a Single-Processor System Without Buffers

- The following example calculates key timing for a single-processor system without buffers.

$$t_{pd\,(TCK\_RET\text{-}TMS/TDI)} = \frac{\left[t_{d\,(TMSmax)} + t_{su\,(TTMS)}\right]}{t_{TCKfactor}}$$

$$= \frac{(20\text{ ns} + 10\text{ ns})}{0.4}$$

$$= 75\text{ ns, or } 13.3\text{ MHz}$$

$$t_{pd\,(TCK\_RET–TDO)} = \frac{\left[t_{d\,(TTDO)} + t_{su\,(TDOmin)}\right]}{t_{TCKfactor}}$$

$$= \frac{(15\text{ ns} + 3\text{ ns})}{0.4}$$

$$= 45\text{ ns, or } 22.2\text{ MHz}$$

In the preceding example, the TCK_RET-to-TMS/TDI path is the limiting factor because it requires more time to complete.

- The following example calculates key timing for a single- or multiple-processor system with buffered input and output:

$$t_{pd\,(TCK\_RET\text{-}TMS/TDI)} = \frac{\left[t_{d\,(TMSmax)} + t_{su\,(TTMS)} + t_{bufskew}\right]}{t_{TCKfactor}}$$

$$= \frac{(20\text{ ns} + 10\text{ ns} + 1.35\text{ ns})}{0.4}$$

$$= 78.4\text{ ns, or } 12.7\text{ MHz}$$

$$t_{pd\,(TCK\_RET–TDO)} = \frac{\left[t_{d\,(TTDO)} + t_{su\,(TDOmin)} + t_{d\,(bufmax)}\right]}{t_{TCKfactor}}$$

$$= \frac{(15\text{ ns} + 3\text{ ns} + 10\text{ ns})}{0.4}$$

$$= 70\text{ ns, or } 14.3\text{ MHz}$$

In the preceding example, the TCK_RET-to-TMS/TDI path is the limiting factor becaise it requires more time to complete.

In a multiprocessor application, it is necessary to ensure that the EMU0 and EMU1 lines can go from a logic-low level to a logic-high level in less than 10 μs, this parameter is called rise time, t_r. This can be calculated as follows:

$$
\begin{aligned}
t_r &= 5(R_{pullup} \times N_{devices} \times C_{load\_per\_device}) \\
&= 5(4.7\ k\Omega \times 16 \times 15\ pF) \\
&= 5(4.7 \times 10^3\ \Omega \times 16 \times 15 = no^{-12}\ F) \\
&= 5(1128 \times 10^{-9}\ ) \\
&= 5.64\ \mu s
\end{aligned}
$$

C.6 Connections Between the Emulator and the Target System

It is extremely important to provide high-quality signals between the emulator and the JTAG target system. You must supply the correct signal buffering, test clock inputs, and multiple processor interconnections to ensure proper emulator and target system operation.

Signals applied to the EMU0 and EMU1 pins on the JTAG target device can be either input or output. In general, these two pins are used as both input and output in multiprocessor systems to handle global run/stop operations. EMU0 and EMU1 signals are applied only as inputs to the XDS510 emulator header.

C.6.1 Buffering Signals

If the distance between the emulation header and the JTAG target device is greater than 6 inches, the emulation signals must be buffered. If the distance is less than 6 inches, no buffering is necessary. Figure C–4 shows the simpler, no-buffering situation.

The distance between the header and the JTAG target device must be no more than 6 inches. The EMU0 and EMU1 signals must have pullup resistors connected to V_{CC} to provide a signal rise time of less than 10 μs. A 4.7-kΩ resistor is suggested for most applications.

Figure C–4. Emulator Connections Without Signal Buffering

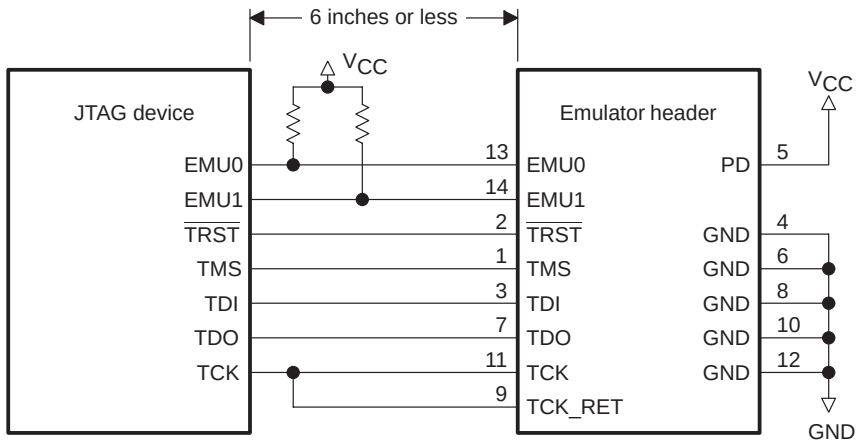
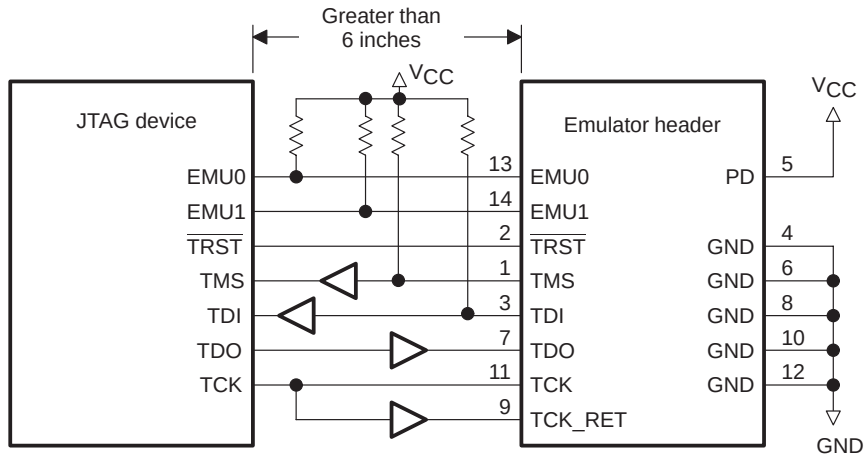


Figure C–5 shows the connections necessary for buffered transmission signals. The distance between the emulation header and the processor is greater than 6 inches. Emulation signals TMS, TDI, TDO, and TCK_RET are buffered through the same device package.

Figure C–5. Emulator Connections With Signal Buffering



The EMU0 and EMU1 signals must have pullup resistors connected to V_{CC} to provide a signal rise time of less than 10 μs. A 4.7-kΩ resistor is suggested for most applications.

The input buffers for TMS and TDI should have pullup resistors connected to V_{CC} to hold these signals at a known value when the emulator is not connected. A resistor value of 4.7 kΩ or greater is suggested.

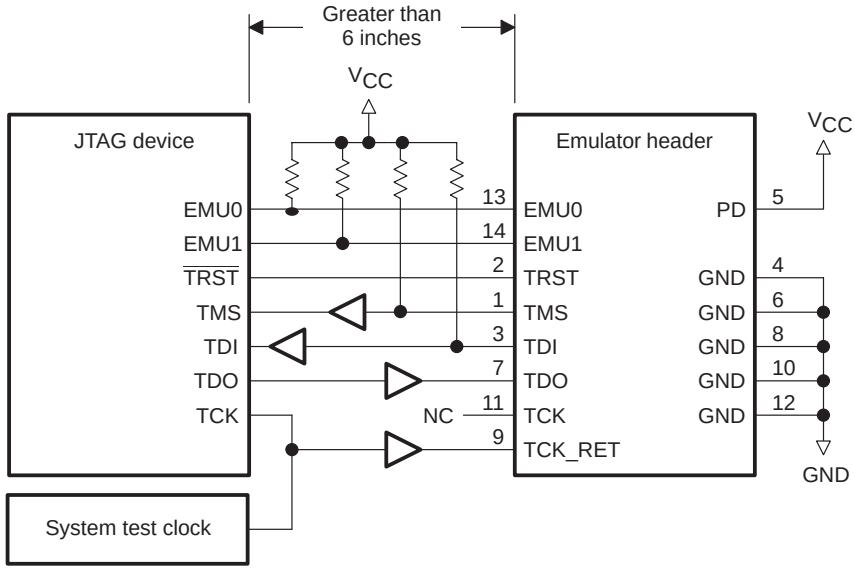
To have high-quality signals (especially the processor TCK and the emulator TCK_RET signals), you may have to employ special care when routing the printed wiring board trace. You also may have to use termination resistors to match the trace impedance. The emulator pod provides optional internal parallel terminators on the TCK_RET and TDO. TMS and TDI provide fixed series termination.

Because $\overline{\text{TRST}}$ is an asynchronous signal, it should be buffered as needed to ensure sufficient current to all target devices.

C.6.2 Using a Target-System Clock

Figure C–6 shows an application with the system test clock generated in the target system. In this application, the emulator's TCK signal is left unconnected.

Figure C–6. Target-System-Generated Test Clock



Note: When the TMS and TDI lines are buffered, pullup resistors must be used to hold the buffer inputs at a known level when the emulator cable is not connected.

There are two benefits in generating the test clock in the target system:

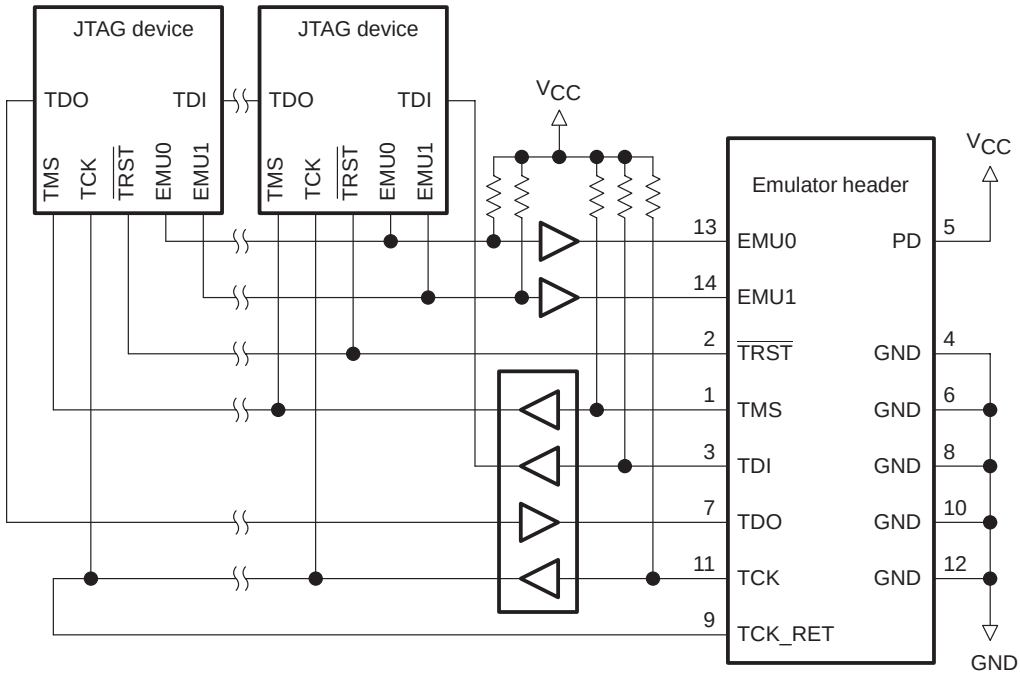
- The emulator provides only a single 10.368-MHz test clock. If you allow the target system to generate your test clock, you can set the frequency to match your system requirements.
- In some cases, you may have other devices in your system that require a test clock when the emulator is not connected. The system test clock also serves this purpose.

C.6.3 Configuring Multiple Processors

Figure C–7 shows a typical daisy-chained multiprocessor configuration that meets the minimum requirements of the IEEE 1149.1 specification. The emulation signals are buffered to isolate the processors from the emulator and provide adequate signal drive for the target system. One of the benefits of this interface is that you can slow down the test clock to eliminate timing problems. Follow these guidelines for multiprocessor support:

- The processor TMS, TDI, TDO, and TCK signals must be buffered through the same physical device package for better control of timing skew.
- The input buffers for TMS, TDI, and TCK must have pullup resistors connected to V_{CC} to hold these signals at a known value when the emulator is not connected. A resistor value of 4.7 kΩ or greater is suggested.
- Buffering EMU0 and EMU1 is optional but highly recommended to provide isolation. These are not critical signals and do not have to be buffered through the same physical package as TMS, TCK, TDI, and TDO.

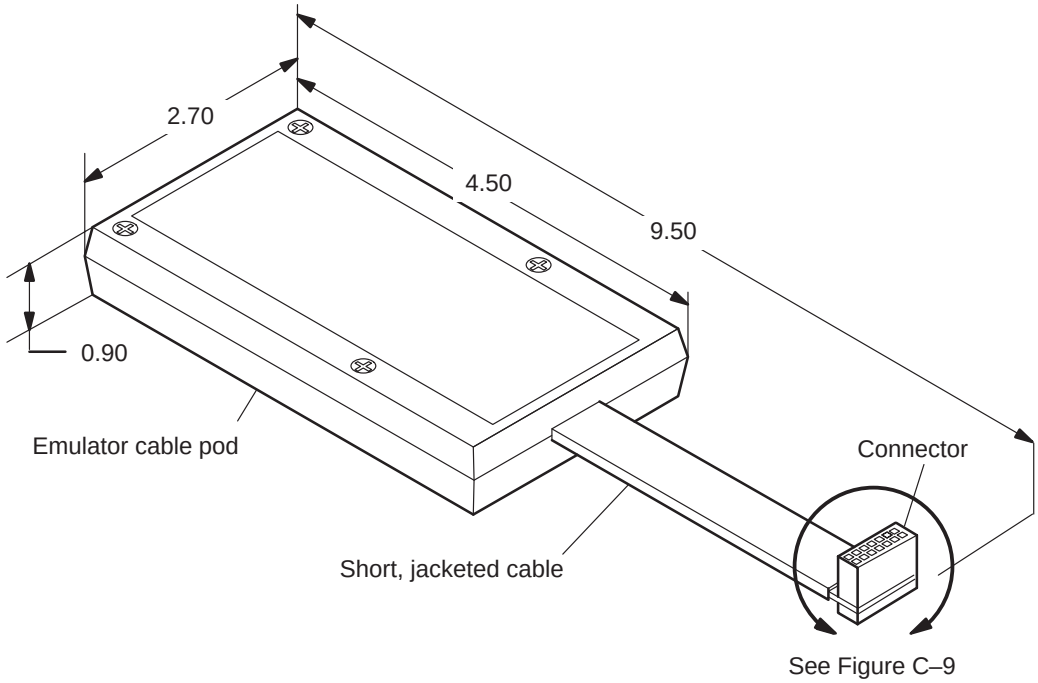
Figure C–7. Multiprocessor Connections



C.7 Physical Dimensions for the 14-Pin Emulator Connector

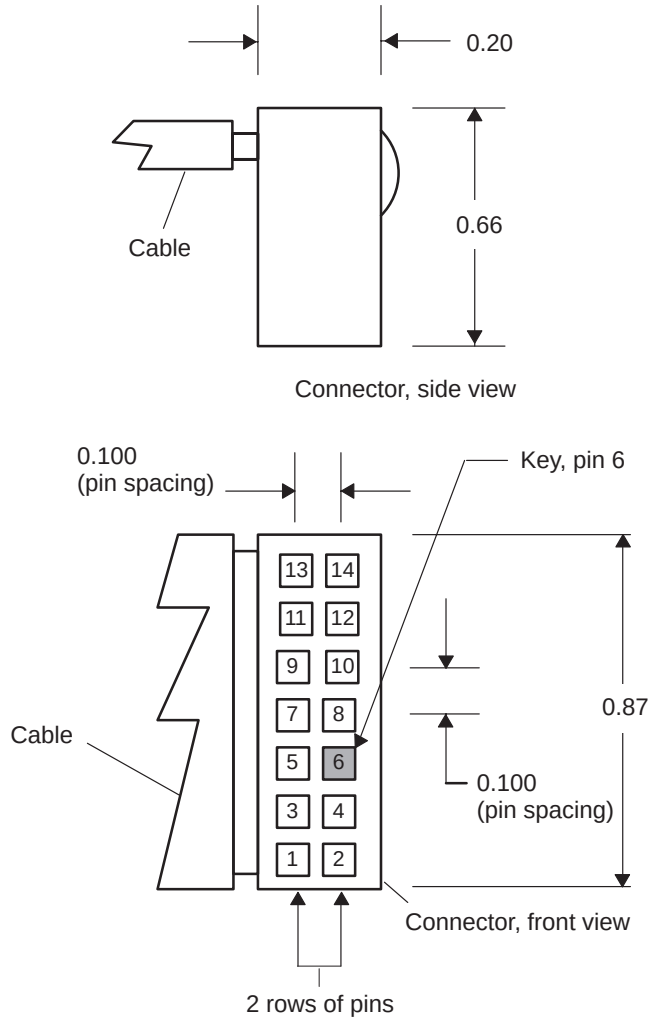
The JTAG emulator target cable consists of a 3-foot section of jacketed cable that connects to the emulator, an active cable pod, and a short section of jacketed cable that connects to the target system. The overall cable length is approximately 3 feet 10 inches. Figure C–8 and Figure C–9 show the physical dimensions for the target cable pod and short cable. The cable pod box is nonconductive plastic with four recessed metal screws.

Figure C–8. Pod/Connector Dimensions



Note: All dimensions are in inches and are nominal dimensions, unless otherwise specified. Pin-to-pin spacing on the connector is 0.100 inches in both the X and Y planes.

Figure C–9. 14-Pin Connector Dimensions



Note: All dimensions are in inches and are nominal dimensions, unless otherwise specified. Pin-to-pin spacing on the connector is 0.100 inches in both the X and Y planes.

C.8 Emulation Design Considerations

This section describes the use and application of the scan path linker (SPL), which can simultaneously add all four secondary JTAG scan paths to the main scan path. It also describes the use of the emulation pins and the configuration of multiple processors.

C.8.1 Using Scan Path Linkers

You can use the TI ACT8997 scan path linker (SPL) to divide the JTAG emulation scan path into smaller, logically connected groups of 4 to 16 devices. As described in the *Advanced Logic and Bus Interface Logic Data Book*, the SPL is compatible with the JTAG emulation scanning. The SPL is capable of adding any combination of its four secondary scan paths into the main scan path.

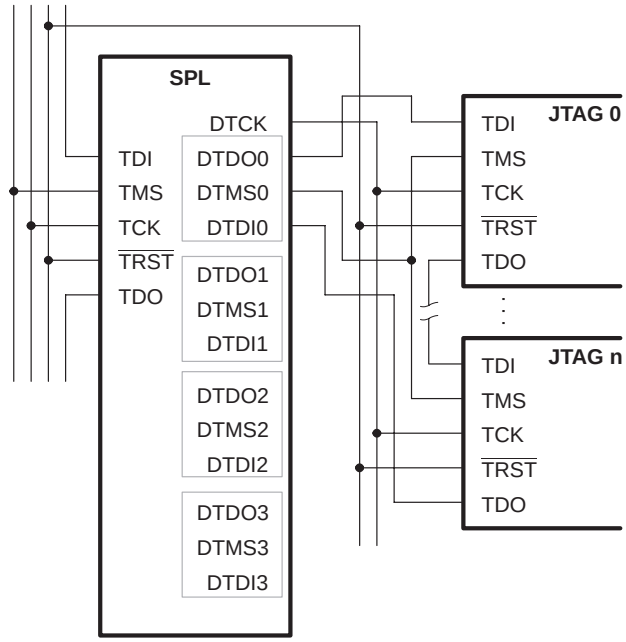
A system of multiple, secondary JTAG scan paths has better fault tolerance and isolation than a single scan path. Since an SPL has the capability of adding all secondary scan paths to the main scan path simultaneously, it can support global emulation operations, such as starting or stopping a selected group of processors.

TI emulators do not support the nesting of SPLs (for example, an SPL connected to the secondary scan path of another SPL). However, you can have multiple SPLs on the main scan path.

Scan path selectors are not supported by this emulation system. The TI ACT8999 scan path selector is similar to the SPL, but it can add only one of its secondary scan paths at a time to the main JTAG scan path. Thus, global emulation operations are not assured with the scan path selector.

You can insert an SPL on a backplane so that you can add up to four device boards to the system without the jumper wiring required with nonbackplane devices. You connect an SPL to the main JTAG scan path in the same way you connect any other device. Figure C–10 shows how to connect a secondary scan path to an SPL.

Figure C–10. Connecting a Secondary JTAG Scan Path to a Scan Path Linker



The $\overline{\text{TRST}}$ signal from the main scan path drives all devices, even those on the secondary scan paths of the SPL. The TCK signal on each target device on the secondary scan path of an SPL is driven by the SPL's DTCK signal. The TMS signal on each device on the secondary scan path is driven by the respective DTMS signals on the SPL.

DTDO0 on the SPL is connected to the TDI signal of the first device on the secondary scan path. DTDI0 on the SPL is connected to the TDO signal of the last device in the secondary scan path. Within each secondary scan path, the TDI signal of a device is connected to the TDO signal of the device before it. If the SPL is on a backplane, its secondary JTAG scan paths are on add-on boards; if signal degradation is a problem, you may need to buffer both the $\overline{\text{TRST}}$ and DTCK signals. Although degradation is less likely for DTMS*n* signals, you may also need to buffer them for the same reasons.

C.8.2 Emulation Timing Calculations for a Scan Path Linker (SPL)

The examples in this section help you to calculate the key emulation timings in the SPL secondary scan path of your system. For actual target timing parameters, see the appropriate device data sheet for your target device.

The examples use the following assumptions:

$t_{su(TTMS)}$	Setup time, target TMS/TDI to TCK high	10 ns
$t_{d(TTDO)}$	Delay time, target TDO from TCK low	15 ns
$t_{d(bufmax)}$	Delay time, target buffer, maximum	10 ns
$t_{d(bufmin)}$	Delay time, target buffer, minimum	1 ns
$t_{(bufskew)}$	Skew time, target buffer, between two devices in the same package: $[t_{d(bufmax)} - t_{d(bufmin)}] \times 0.15$	1.35 ns
$t_{(TCKfactor)}$	Duty cycle, TCK assume a 40/60% clock	0.4 (40%)

Also, the examples use the following values from the SPL data sheet:

$t_{d(DTMSmax)}$	Delay time, SPL DTMS/DTDO from TCK low, maximum	31 ns
$t_{su(DTDLmin)}$	Setup time, DTDI to SPL TCK high, minimum	7 ns
$t_{d(DTCKHmin)}$	Delay time, SPL DTCK from TCK high, minimum	2 ns
$t_{d(DTCKLmax)}$	Delay time, SPL DTCK from TCK low, maximum	16 ns

There are two key timing paths to consider in the emulation design:

- ❏ The TCK-to-DTMS/DTDO path, called $t_{pd(TCK-DTMS)}$
- ❏ The TCK-to-DTDI path, called $t_{pd(TCK-DTDI)}$

In the following two examples, the worst-case path delay is calculated to determine the maximum system test clock frequency.

Example C–2. Key Timing for a Single-Processor System Without Buffering (SPL)

- The following example calculates key timing for a single-processor system without buffering (SPL):

$$t_{pd\,(TCK\text{-}DTMS)} = \frac{\left[t_{d\,(DTMSmax)} + t_{d\,(DTCKHmin)} + t_{su\,(TTMS)}\right]}{t_{TCKfactor}}$$

$$= \frac{(31\text{ ns} + 2\text{ ns} + 10\text{ ns})}{0.4}$$

$$= 107.5\text{ ns, or } 9.3\text{ MHz}$$

$$t_{pd\,(TCK\text{-}DTDI)} = \frac{\left[t_{d\,(TTDO)} + t_{d\,(DTCKLmax)} + t_{su\,(DTDLmin)}\right]}{t_{TCKfactor}}$$

$$= \frac{(15\text{ ns} + 16\text{ ns} + 7\text{ ns})}{0.4}$$

$$= 9.5\text{ ns, or } 10.5\text{ MHz}$$

In the preceding example, the TCK-to-DTMS/DTDL path is the limiting factor.

- The following example calculates key timing for a single- or multiprocessor-system with buffered input and output (SPL):

$$t_{pd\,(TCK\text{-}TDMS)} = \frac{\left[t_{d\,(DTMSmax)} + t_{(DTCKHmin)} + t_{su\,(TTMS)} + t_{(bufskew)}\right]}{t_{TCKfactor}}$$

$$= \frac{(31\text{ ns} + 2\text{ ns} + 10\text{ ns} + 1.35\text{ ns})}{0.4}$$

$$= 110.9\text{ ns, or } 9.0\text{ MHz}$$

$$t_{pd\,(TCK–DTDI)} = \frac{\left[t_{d\,(TTDO)} + t_{d\,(DTCKLmax)} + t_{su\,(DTDLmin)} + t_{d\,(bufskew)}\right]}{t_{TCKfactor}}$$

$$= \frac{(15\text{ ns} + 15\text{ ns} + 7\text{ ns} + 10\text{ ns})}{0.4}$$

$$= 120\text{ ns, or } 8.3\text{ MHz}$$

In the preceding example, the TCK-to-DTDI path is the limiting factor.

C.8.3 Using Emulation Pins

The EMU0/1 pins of TI devices are bidirectional, 3-state output pins. When in an inactive state, these pins are at high impedance. When the pins are active, they provide one of two types of output:

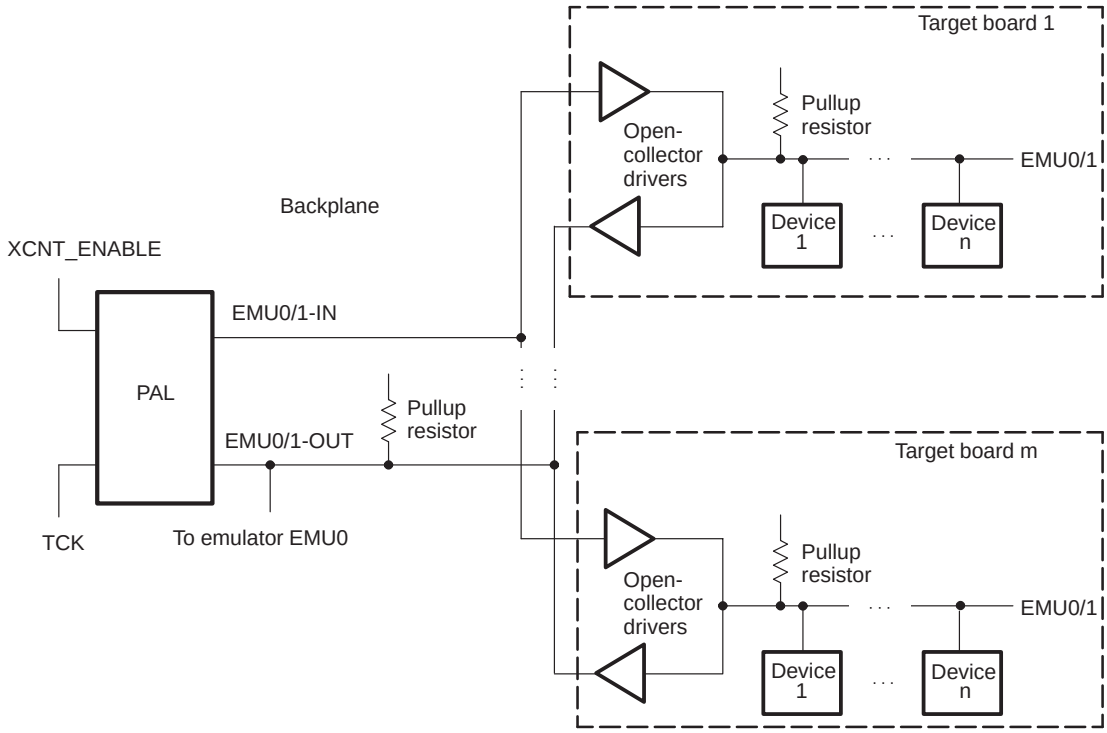
- Signal event. The EMU0/1 pins can be configured via software to signal internal events. In this mode, driving one of these pins low can cause devices to signal such events. To enable this operation, the EMU0/1 pins function as open-collector sources. External devices such as logic analyzers can also be connected to the EMU0/1 signals in this manner. If such an external source is used, it must also be connected via an open-collector source.

- External count. The EMU0/1 pins can be configured via software as totem-pole outputs for driving an external counter. If the output of more than one device is configured for totem-pole operation, then these devices can be damaged. The emulation software detects and prevents this condition. However, the emulation software has no control over external sources on the EMU0/1 signal. Therefore, all external sources must be inactive when any device is in the external count mode.

TI devices can be configured by software to halt processing if their EMU0/1 pins are driven low. This feature combined with the signal event output, allows one TI device to halt all other TI devices on a given event for system-level debugging.

If you route the EMU0/1 signals between multiple boards, they require special handling because they are more complex than normal emulation signals. Figure C–11 shows an example configuration that allows any processor in the system to stop any other processor in the system. Do not tie the EMU0/1 pins of more than 16 processors together in a single group without using buffers. Buffers provide the crisp signals that are required during a RUNB (run benchmark) debugger command or when the external analysis counter feature is used.

Figure C–11. EMU0/1 Configuration to Meet Timing Requirements of Less Than 25 ns



Notes: 1) The low time on EMU0/1-IN must be at least one TCK cycle and less than 10 μs. Software sets the EMU0/1-OUT pin to a high state.

2) To enable the open-collector driver and pullup resistor on EMU1 to provide rise/fall times of less than 25 ns, the modification shown in this figure is suggested. Rise times of more than 25 ns can cause the emulator to detect false edges during the RUNB command or when the external counter selected from the debugger analysis menu is used.

These seven important points apply to the circuitry shown in Figure C–11 and the timing shown in Figure C–12:

- ❏ Open-collector drivers isolate each board. The EMU0/1 pins are tied together on each board.
- ❏ At the board edge, the EMU0/1 signals are split to provide both input and output connections. This is required to prevent the open-collector drivers from acting as latches that can be set only once.
- ❏ The EMU0/1 signals are bused down the backplane. Pullup resistors must be installed as required.

- The bused EMU0/1 signals go into a programmable logic array device PAL®, whose function is to generate a low pulse on the EMU0/1-IN signal when a low level is detected on the EMU0/1-OUT signal. This pulse must be longer than one TCK period to affect the devices but less than 10 μs to avoid possible conflicts or retriggering once the emulation software clears the device's pins.
- During a RUNB debugger command or other external analysis count, the EMU0/1 pins on the target device become totem-pole outputs. The EMU1 pin is a ripple carry-out of the internal counter. EMU0 becomes a *processor-halted* signal. During a RUNB or other external analysis count, the EMU0/1-IN signal to all boards must remain in the high (disabled) state. You must provide some type of external input (XCNT_ENABLE) to the PAL® to disable the PAL® from driving EMU0/1-IN to a low state.
- If you use sources other than TI processors (such as logic analyzers) to drive EMU0/1, their signal lines must be isolated by open-collector drivers and be inactive during RUNB and other external analysis counts.
- You must connect the EMU0/1-OUT signals to the emulation header or directly to a test bus controller.

Figure C–12. Suggested Timings for the EMU0 and EMU1 Signals

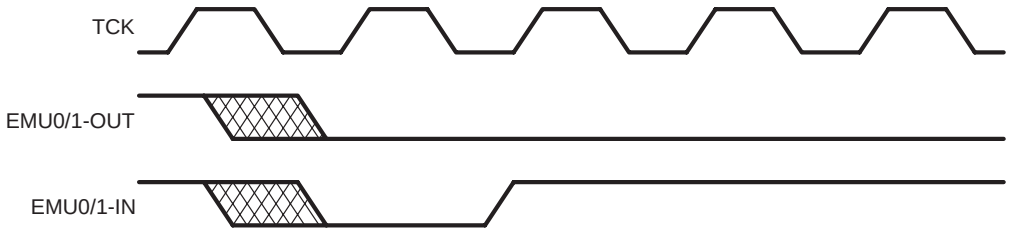
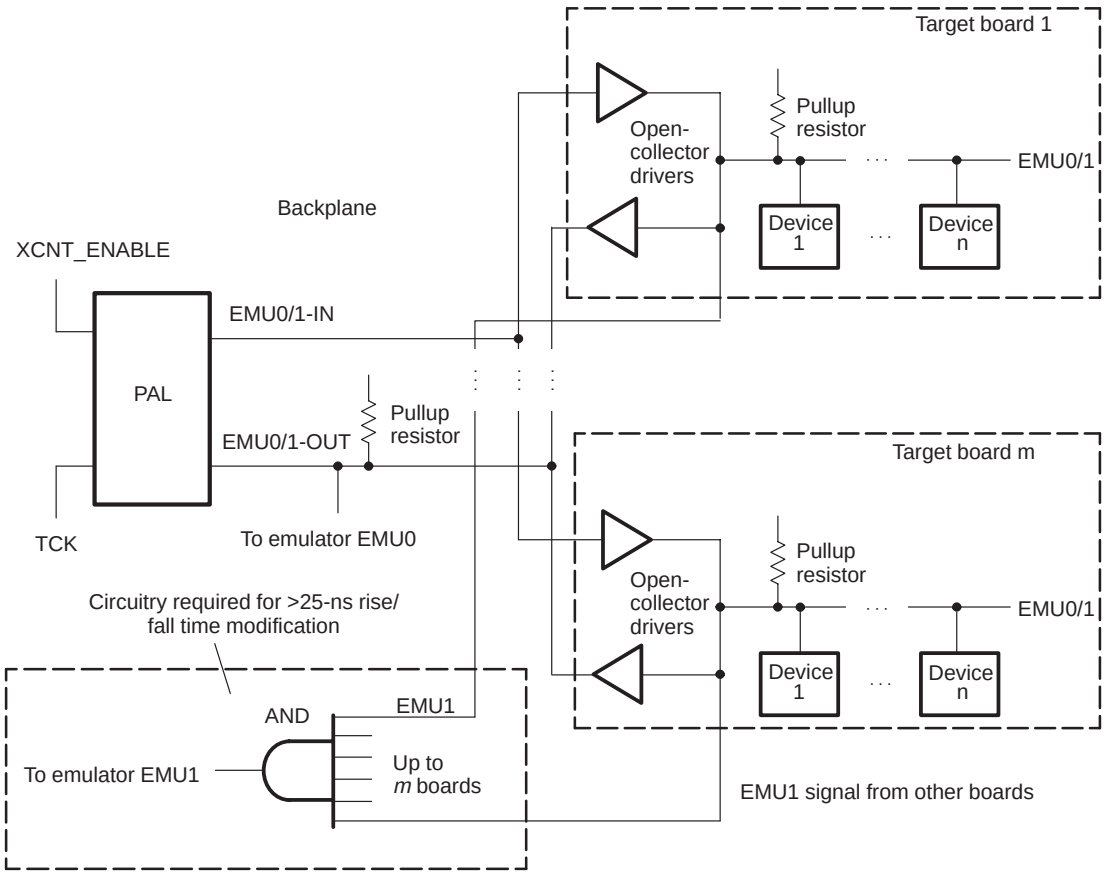


Figure C–13. EMU0/1 Configuration With Additional AND Gate to Meet Timing Requirements of Greater Than 25 ns

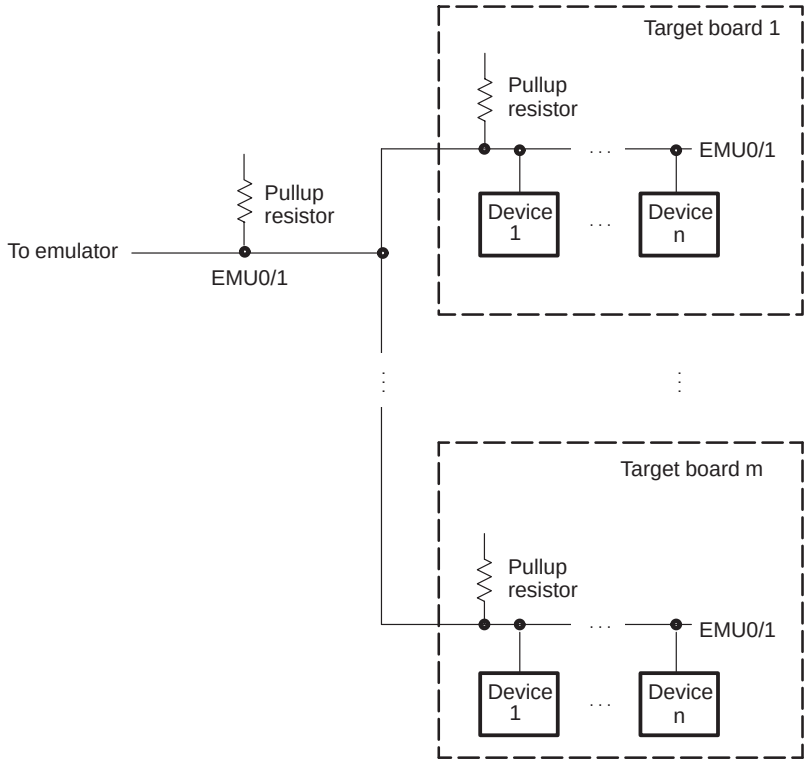


Notes: 1) The low time on EMU0/1-IN must be at least one TCK cycle and less than 10 μs. Software sets the EMU0/1-OUT pin to a high state.

2) To enable the open-collector driver and pullup resistor on EMU1 to provide rise/fall time of greater than 25 ns, the modification shown in this figure is suggested. Rise times of more than 25 ns can cause the emulator to detect false edges during the RUNB command or when the external counter selected from the debugger analysis menu is used.

You do not need to have devices on one target board stop devices on another target board using the EMU0/1 signals (see the circuit in Figure C–14). In this configuration, the global-stop capability is lost. It is important not to overload EMU0/1 with more than 16 devices.

Figure C–14. EMU0/1 Configuration Without Global Stop

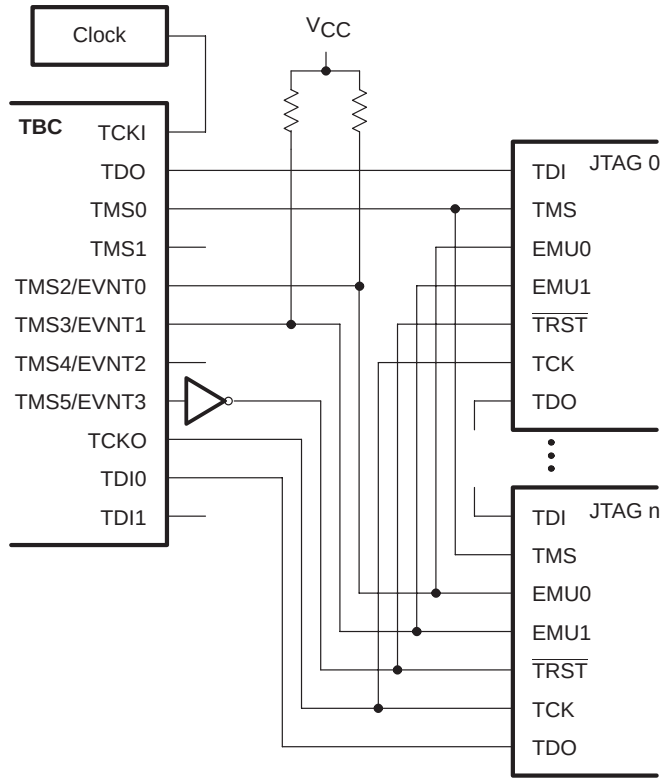


Note: The open-collector driver and pullup resistor on EMU1 must be able to provide rise/fall times of less than 25 ns. Rise times of more than 25 ns can cause the emulator to detect false edges during the RUNB command or when the external counter selected from the debugger analysis menu is used. If this condition cannot be met, then the EMU0/1 signals from the individual boards must be ANDed together (as shown in Figure C–14) to produce an EMU0/1 signal for the emulator.

C.8.4 Performing Diagnostic Applications

For systems that require built-in diagnostics, it is possible to connect the emulation scan path directly to a TI ACT8990 test bus controller (TBC) instead of the emulation header. The TBC is described in the Texas Instruments *Advanced Logic and Bus Interface Logic Data Book*. Figure C–15 shows the scan path connections of *n* devices to the TBC.

Figure C–15. TBC Emulation Connections for n JTAG Scan Paths



In the system design shown in Figure C–15, the TBC emulation signals TCKI, TDO, TMS0, TMS2/EVNT0, TMS3/EVNT1, TMS5/EVNT3, TCKO, and TDI0 are used, and TMS1, TMS4/EVNT2, and TDI1 are not connected. The target devices' EMU0 and EMU1 signals are connected to V_{CC} through pullup resistors and tied to the TBC's TMS2/EVNT0 and TMS3/EVNT1 pins, respectively. The TBC's TCKI pin is connected to a clock generator. The TCK signal for the main JTAG scan path is driven by the TBC's TCKO pin.

On the TBC, the TMS0 pin drives the TMS pins on each device on the main JTAG scan path. TDO on the TBC connects to TDI on the first device on the main JTAG scan path. TDI0 on the TBC is connected to the TDO signal of the last device on the main JTAG scan path. Within the main JTAG scan path, the TDI signal of a device is connected to the TDO signal of the device before it. $\overline{\text{TRST}}$ for the devices can be generated either by inverting the TBC's TMS5/EVNT3 signal for software control or by logic on the board itself.

Appendix D

Glossary

A

A0–A15: Collectively, the external address bus; the 16 pins are used in parallel to address external data memory, program memory, or I/O space.

ACC: See *accumulator.*

ACCH: *Accumulator high word.* The upper 16 bits of the accumulator. See also *accumulator*.

ACCL: *Accumulator low word.* The lower 16 bits of the accumulator. See also *accumulator*.

accumulator: A 32-bit register that stores the results of operations in the central arithmetic logic unit (CALU) and provides an input for subsequent CALU operations. The accumulator also performs shift and rotate operations.

address: The location of program code or data stored in memory.

addressing mode: A method by which an instruction interprets its operands to acquire the data it needs. See also *direct addressing*; *immediate addressing*; *indirect addressing*.

analog-to-digital (A/D) converter: A circuit that translates an analog signal to a digital signal.

AR: See *auxiliary register*.

AR0–AR7: *Auxiliary registers 0 through 7.* See *auxiliary register*.

ARAU: See *auxiliary register arithmetic unit (ARAU).*

ARB: See *auxiliary register pointer buffer (ARB)*.

ARP: See *auxiliary register pointer (ARP).*

auxiliary register: One of eight 16-bit registers (AR7–AR0) used as pointers to addresses in data space. The registers are operated on by the auxiliary register arithmetic unit (ARAU) and are selected by the auxiliary register pointer (ARP).

auxiliary register arithmetic unit (ARAU): A 16-bit arithmetic unit used to increment, decrement, or compare the contents of the auxiliary registers. Its primary function is manipulating auxiliary register values for indirect addressing.

auxiliary register pointer (ARP): A 3-bit field in status register ST0 that points to the current auxiliary register.

auxiliary register pointer buffer (ARB): A 3-bit field in status register ST1 that holds the previous value of the auxiliary register pointer (ARP).

B

B0: An on-chip block of dual-access RAM that can be configured as either data memory or program memory, depending on the value of the CNF bit in status register ST1.

B1: An on-chip block of dual-access RAM available for data memory.

B2: An on-chip block of dual-access RAM available for data memory.

$\overline{\text{BIO}}$ pin: A general-purpose input pin that can be tested by conditional instructions that cause a branch when an external device drives $\overline{\text{BIO}}$ low.

bit-reversed indexed addressing: A method of indirect addressing that allows efficient I/O operations by resequencing the data points in a radix-2 fast Fourier transform (FFT) program. The direction of carry propagation in the ARAU is reversed.

boot loader: A built-in segment of code that transfers code from an external source to a 16-bit external program destination at reset.

$\overline{\text{BR}}$: *Bus request pin*. This pin is tied to the $\overline{\text{BR}}$ signal, which is asserted when a global data memory access is initiated.

branch: A switching of program control to a nonsequential program-memory address.

C

C bit: See *carry bit.*

CALU: See *central arithmetic logic unit (CALU).*

carry bit: Bit 9 of status register ST1; used by the CALU for extended arithmetic operations and accumulator shifts and rotates. The carry bit can be tested by conditional instructions.

central arithmetic logic unit (CALU): The 32-bit wide main arithmetic logic unit for the ’C24x CPU that performs arithmetic and logic operations. It accepts 32-bit values for operations, and its 32-bit output is held in the accumulator.

CLKIN: *Input clock signal*. A clock source signal supplied to the on-chip clock generator at the CLKIN/X2 pin or generated internally by the on-chip oscillator. The clock generator divides or multiplies CLKIN to produce the CPU clock signal, CLKOUT1.

CLKOUT: *Master clock output signal*. The output signal of the on-chip clock generator. The CLKOUT1 high pulse signifies the CPU’s logic phase (when internal values are changed), and the CLKOUT1 low pulse signifies the CPU’s latch phase (when the values are held constant).

clock mode (clock generator): One of the modes which sets the internal CPU clock frequency to a fraction or multiple of the frequency of the input clock signal CLKIN.

CNF bit: *DARAM configuration bit*. Bit 12 in status register ST1. CNF is used to determine whether the on-chip RAM block B0 is mapped to program space or data space.

codec: A device that codes in one direction of transmission and decodes in another direction of transmission.

COFF: *Common object file format.* A system of files configured according to a standard developed by AT&T. These files are relocatable in memory space.

context saving/restoring: Saving the system status when the device enters a subroutine (such as an interrupt service routine) and restoring the system status when exiting the subroutine. On the ’C24x, only the program counter value is saved and restored automatically; other context saving and restoring must be performed by the subroutine.

CPU: *Central processing unit*. The ’C24x CPU is the portion of the processor involved in arithmetic, shifting, and Boolean logic operations, as well as the generation of data- and program-memory addresses. The CPU includes the central arithmetic logic unit (CALU), the multiplier, and the auxiliary register arithmetic unit (ARAU).

CPU cycle: The time required for the CPU to go through one logic phase (during which internal values are changed) and one latch phase (during which the values are held constant).

current AR: See *current auxiliary register*.

current auxiliary register: The auxiliary register pointed to by the auxiliary register pointer (ARP). The auxiliary registers are AR0 (ARP = 0) through AR7 (ARP = 7). See also *auxiliary register*; *next auxiliary register*.

current data page: The data page indicated by the content of the data page pointer (DP). See also *data page*; *DP*.

D

D0–D15: Collectively, the external data bus; the 16 pins are used in parallel to transfer data between the ’C24x and external data memory, program memory, or I/O space.

DARAM: *Dual-access RAM*. RAM that can be accessed twice in a single CPU clock cycle. For example, your code can read from and write to DARAM in the same clock cycle.

DARAM configuration bit (CNF): See *CNF bit*.

data-address generation logic: Logic circuitry that generates the addresses for data memory reads and writes. This circuitry, which includes the auxiliary registers and the ARAU, can generate one address per machine cycle. See also *program-address generation logic.*

data page: One block of 128 words in data memory. Data memory contains 512 data pages. Data page 0 is the first page of data memory (addresses 0000h–007Fh); data page 511 is the last page (addresses FF80h–FFFFh). See also *data page pointer (DP)*; *direct addressing*.

data page 0: Addresses 0000h–007Fh in data memory; contains the memory-mapped registers, a reserved test/emulation area for special information transfers, and the scratch-pad RAM block (B2).

data page pointer (DP): A 9-bit field in status register ST0 that specifies which of the 512 data pages is currently selected for direct address generation. When an instruction uses direct addressing to access a data-memory value, the DP provides the nine MSBs of the data-memory address, and the instruction provides the seven LSBs.

data-read address bus (DRAB): A 16-bit internal bus that carries the address for each read from data memory.

data read bus (DRDB): A 16-bit internal bus that carries data from data memory to the CALU and the ARAU.

data-write address bus (DWAB): A 16-bit internal bus that carries the address for each write to data memory.

data write bus (DWEB): A 16-bit internal bus that carries data to both program memory and data memory.

decode phase: The phase of the pipeline in which the instruction is decoded. See also *pipeline*; *instruction-fetch phase*; *operand-fetch phase; instruction-execute phase*.

direct addressing: One of the methods used by an instruction to address data-memory. In direct addressing, the data-page pointer (DP) holds the nine MSBs of the address (the current data page), and the instruction word provides the seven LSBs of the address (the offset). See also *indirect addressing*.

DP: See *data page pointer (DP).*

DRAB: See *data-read address bus (DRAB).*

DRDB: See *data read bus (DRDB).*

$\overline{\text{DS}}$: *Data memory select pin*. The ’C24x asserts $\overline{\text{DS}}$ to indicate an access to external data memory (local or global).

DSWS: *Data-space wait-state bit(s).* A value in the wait-state generator control register (WSGR) that determines the number of wait states applied to reads from and writes to off-chip data space.

dual-access RAM: See *DARAM*.

dummy cycle: A CPU cycle in which the CPU intentionally reloads the program counter with the same address.

DWAB: See *data-write address bus (DWAB).*

DWEB: See *data write bus (DWEB).*

E

execute phase: The fourth phase of the pipeline; the phase in which the instruction is executed. See also *pipeline*; *instruction-fetch phase*; *instruction-decode phase*; *operand-fetch phase*.

external interrupt: A hardware interrupt triggered by an external event sending an input through an interrupt pin.

F

FIFO buffer: *First-in, first-out buffer.* A portion of memory in which data is stored and then retrieved in the same order in which it was stored. The synchronous serial port has two four-word-deep FIFO buffers: one for its transmit operation and one for its receive operation.

flash memory: Electronically erasable and programmable, nonvolatile (read-only) memory.

G

general-purpose input/output pins: Pins that can be used to accept input signals or send output signals. These pins are the input pin $\overline{\text{BIO}}$, the output pin XF, and the GPIO pins.

global data space: One of the four ’C24x address spaces. The global data space can be used to share data with other processors within a system and can serve as additional data space. See also *local data space*.

GREG: *Global memory allocation register.* A memory-mapped register used for specifying the size of the global data memory. Addresses not allocated by the GREG for global data memory are available for local data memory.

H

hardware interrupt: An interrupt triggered through physical connections with on-chip peripherals or external devices.

I

immediate addressing: One of the methods for obtaining data values used by an instruction; the data value is a constant embedded directly into the instruction word; data memory is not accessed.

immediate operand/immediate value: A constant given as an operand in an instruction that is using immediate addressing.

IMR: See *interrupt mask register (IMR)*.

indirect addressing: One of the methods for obtaining data values used by an instruction. When an instruction uses indirect addressing, data memory is addressed by the current auxiliary register. See also *direct addressing*.

input clock signal: See *CLKIN*.

input shifter: A 16- to 32-bit left barrel shifter that shifts incoming 16-bit data from 0 to 16 positions left relative to the 32-bit output.

instruction-decode phase: The second phase of the pipeline; the phase in which the instruction is decoded. See also *pipeline*; *instruction-fetch phase*; *operand-fetch phase; instruction-execute phase*.

instruction-execute phase: The fourth phase of the pipeline; the phase in which the instruction is executed. See also *pipeline*; *instruction-fetch phase*; *instruction-decode phase*; *operand-fetch phase*.

instruction-fetch phase: The first phase of the pipeline; the phase in which the instruction is fetched from program-memory. See also *pipeline*; *instruction-decode phase*; *operand-fetch phase; instruction-execute phase*.

instruction register (IR): A 16-bit register that contains the instruction being executed.

instruction word: A 16-bit value representing all or half of an instruction. An instruction that is fully represented by 16 bits uses one instruction word. An instruction that must be represented by 32 bits uses two instruction words (the second word is a constant).

internal interrupt: A hardware interrupt caused by an on-chip peripheral.

interrupt: A signal sent to the CPU that (when not masked or disabled) forces the CPU into a subroutine called an interrupt service routine (ISR). This signal can be triggered by an external device, an on-chip peripheral, or an instruction (INTR, NMI, or TRAP).

interrupt acknowledge signal ($\overline{\text{IACK}}$): A signal that indicates an interrupt has been received and that the program counter is fetching the interrupt vector that will force the processor into the appropriate interrupt service routine.

interrupt flag register (IFR): A 16-bit memory-mapped register that indicates pending interrupts. Read the IFR to identify pending interrupts and write to the IFR to clear selected interrupts. Writing a 1 to any IFR flag bit clears that bit to 0.

interrupt latency: The delay between the time an interrupt request is made and the time it is serviced.

interrupt mask register (IMR): A 16-bit memory-mapped register used to mask external and internal interrupts. Writing a 1 to any IMR bit position enables the corresponding interrupt (when INTM = 0).

interrupt mode bit (INTM): Bit 9 in status register ST0; either enables all maskable interrupts that are not masked by the IMR or disables all maskable interrupts.

interrupt service routine (ISR): A module of code that is executed in response to a hardware or software interrupt.

interrupt trap: See *interrupt service routine (ISR)*.

interrupt vector: A branch instruction that leads the CPU to an interrupt service routine (ISR).

interrupt vector location: An address in program memory where an interrupt vector resides. When an interrupt is acknowledged, the CPU branches to the interrupt vector location and fetches the interrupt vector.

INTM bit: See *interrupt mode bit (INTM)*.

I/O-mapped register: One of the on-chip registers mapped to addresses in I/O (input/output) space. These registers, which include the registers for the on-chip peripherals, must be accessed with the IN and OUT instructions. See also *memory-mapped register*.

IR: See *instruction register (IR)*.

$\overline{\text{IS}}$: *I/O space select pin*. The ’C24x asserts $\overline{\text{IS}}$ to indicate an access to external I/O space.

ISR: See *interrupt service routine (ISR)*.

ISWS: *I/O-space wait-state bit(s).* A value in the wait-state generator control register (WSGR) that determines the number of wait states applied to reads from and writes to off-chip I/O space.

L

latch phase: The phase of a CPU cycle during which internal values are held constant. See also *logic phase*; *CLKOUT1*.

local data space: The portion of data-memory addresses that are not allocated as global by the global memory allocation register (GREG). If none of the data-memory addresses are allocated for global use, all of data space is local. See also *global data space*.

logic phase: The phase of a CPU cycle during which internal values are changed. See also *latch phase*; *CLKOUT1*.

long-immediate value: A 16-bit constant given as an operand of an instruction that is using immediate addressing.

LSB: *Least significant bit.* The lowest order bit in a word. When used in plural form (LSBs), refers to a specified number of low-order bits, beginning with the lowest order bit and counting to the left. For example, the four LSBs of a 16-bit value are bits 0 through 3. See also *MSB*.

M

machine cycle: See *CPU cycle*.

maskable interrupt: A hardware interrupt that can be enabled or disabled through software. See also *nonmaskable interrupt*.

master clock output signal: See *CLKOUT1*.

master phase: See *logic phase*.

memory-mapped register: One of the on-chip registers mapped to addresses in data memory. See also *I/O-mapped register*.

microcomputer mode: A mode in which the on-chip ROM or flash memory is enabled. This mode is selected with the MP/$\overline{\text{MC}}$ pin. See also *MP/$\overline{\text{MC}}$ pin*; *microprocessor mode*.

microprocessor mode: A mode in which the on-chip ROM or flash memory is disabled. This mode is selected with the MP/$\overline{\text{MC}}$ pin. See also *MP/$\overline{\text{MC}}$ pin*; *microcomputer mode*.

microstack (MSTACK): A register used for temporary storage of the program counter (PC) value when an instruction needs to use the PC to address a second operand.

MIPS: Million instructions per second.

MP/$\overline{\text{MC}}$ pin: A pin that indicates whether the processor is operating in microprocessor mode or microcomputer mode. MP/$\overline{\text{MC}}$ high selects microprocessor mode; MP/$\overline{\text{MC}}$ low selects microcomputer mode.

MSB: *Most significant bit.* The highest order bit in a word. When used in plural form (MSBs), refers to a specified number of high-order bits, beginning with the highest order bit and counting to the right. For example, the eight MSBs of a 16-bit value are bits 15 through 8. See also *LSB*.

MSTACK: See *microstack*.

multiplier: A part of the CPU that performs 16-bit $\times$ 16-bit multiplication and generates a 32-bit product. The multiplier operates using either signed or unsigned 2s-complement arithmetic.

N

next AR: See *next auxiliary register*.

next auxiliary register: The register that is pointed to by the auxiliary register pointer (ARP) when an instruction that modifies ARP is finished executing. See also *auxiliary register*; *current auxiliary register*.

$\overline{\text{NMI}}$: A hardware interrupt that uses the same logic as the maskable interrupts but cannot be masked. It is often used as a soft reset. See also *maskable interrupt*; *nonmaskable interrupt*.

nonmaskable interrupt: An interrupt that can be neither masked by the interrupt mask register (IMR) nor disabled by the INTM bit of status register ST0.

NPAR: *Next program address register.* Part of the program-address generation logic. This register provides the address of the next instruction to the program counter (PC), the program address register (PAR), the micro stack (MSTACK), or the stack.

O

operand: A value to be used or manipulated by an instruction; specified in the instruction.

operand-fetch phase: The third phase of the pipeline; the phase in which an operand or operands are fetched from memory. See also *pipeline*; *instruction-fetch phase*; *instruction-decode phase; instruction-execute phase*.

output shifter: 32- to 16-bit barrel left shifter. Shifts the 32-bit accumulator output from 0 to 7 bits left for quantization management, and outputs either the 16-bit high or low half of the shifted 32-bit data to the data write bus (DWEB).

OV bit: *Overflow flag bit.* Bit 12 of status register ST0; indicates whether the result of an arithmetic operation has exceeded the capacity of the accumulator.

overflow (in a register): A condition in which the result of an arithmetic operation exceeds the capacity of the register used to hold that result.

overflow mode: The mode in which an overflow in the accumulator causes the accumulator to be loaded with a preset value. If the overflow is in the positive direction, the accumulator is loaded with its most positive number. If the overflow is in the negative direction, the accumulator is filled with its most negative number.

OVM bit: *Overflow mode bit.* Bit 11 of status register ST0; enables or disables overflow mode. See also *overflow mode*.

P

PAB: See *program address bus (PAB).*

PAR: *Program address register.* A register that holds the address currently being driven on the program address bus for as many cycles as it takes to complete all memory operations scheduled for the current machine cycle.

PC: See *program counter (PC).*

PCB: *Printed circuit board*.

pending interrupt: A maskable interrupt that has been successfully requested but is awaiting acknowledgement by the CPU.

pipeline: A method of executing instructions in an assembly line fashion. The ’C24x pipeline has four independent phases. During a given CPU cycle, four different instructions can be active, each at a different stage of completion. See also *instruction-fetch phase*; *instruction-decode phase*; *operand-fetch phase; instruction-execute phase*.

PLL: Phase lock loop circuit.

PM bits: See *product shift mode bits (PM).*

power-down mode: The mode in which the processor enters a dormant state and dissipates considerably less power than during normal operation. This mode is initiated by the execution of an IDLE instruction. During a power-down mode, all internal contents are maintained so that operation continues unaltered when the power-down mode is terminated. The contents of all on-chip RAM also remains unchanged.

PRDB: See *program read bus (PRDB).*

PREG: See *product register (PREG).*

product register (PREG): A 32-bit register that holds the results of a multiply operation.

product shifter: A 32-bit shifter that performs a 0-, 1-, or 4-bit left shift, or a 6-bit right shift of the multiplier product based on the value of the product shift mode bits (PM).

product shift mode: One of four modes (no-shift, shift-left-by-one, shift-left-by-four, or shift-right-by-six) used by the product shifter.

product shift mode bits (PM): Bits 0 and 1 of status register ST1; they identify which of four shift modes (no-shift, left-shift-by-one, left-shift-by-four, or right-shift-by-six) will be used by the product shifter.

program address bus (PAB): A 16-bit internal bus that provides the addresses for program-memory reads and writes.

program-address generation logic: Logic circuitry that generates the addresses for program memory reads and writes, and an operand address in instructions that require two registers to address operands. This circuitry can generate one address per machine cycle. See also *data-address generation logic.*

program control logic: Logic circuitry that decodes instructions, manages the pipeline, stores status of operations, and decodes conditional operations.

program counter (PC): A register that indicates the location of the next instruction to be executed.

program read bus (PRDB): A 16-bit internal bus that carries instruction code and immediate operands, as well as table information, from program memory to the CPU.

$\overline{\text{PS}}$: *Program select pin*. The ’C24x asserts $\overline{\text{PS}}$ to indicate an access to external program memory.

PSLWS: *Lower program-space wait-state bits.* A value in the wait-state generator control register (WSGR) that determines the number of wait states applied to reads from and writes to off-chip lower program space (addresses 0000h–7FFFh). See also *PSUWS*.

PSUWS: *Upper program-space wait-state bits.* A value in the wait-state generator control register (WSGR) that determines the number of wait states applied to reads from and writes to off-chip upper program space (addresses 8000h–FFFFh). See also *PSLWS*.

R

$\overline{\text{RD}}$: *Read select pin*. The ’C24x asserts $\overline{\text{RD}}$ to request a read from external program, data, or I/O space. $\overline{\text{RD}}$ can be connected directly to the output enable pin of an external device.

READY: *External device ready pin*. Used to create wait states externally. When this pin is driven low, the ’C24x waits one CPU cycle and then tests READY again. After READY is driven low, the ’C24x does not continue processing until READY is driven high.

repeat counter (RPTC): A 16-bit register that counts the number of times a single instruction is repeated. RPTC is loaded by an RPT instruction.

reset: A way to bring the processor to a known state by setting the registers and control bits to predetermined values and signaling execution to start at address 0000h.

reset pin ($\overline{\text{RS}}$): A pin that causes a reset.

reset vector: The interrupt vector for reset.

return address: The address of the instruction to be executed when the CPU returns from a subroutine or interrupt service routine.

RPTC: See *repeat counter (RPTC).*

$\overline{\text{RS}}$: *Reset pin*. When driven low, causes a reset on any ’C24x device.

R/$\overline{\text{W}}$: *Read/write pin.* Indicates the direction of transfer between the ’C24x and external program, data, or I/O space.

S

SARAM: *Single-access RAM*. RAM that can be accessed (read from or written to) once in a single CPU cycle.

scratch-pad RAM: Another name for DARAM block B2 in data space (32 words).

short-immediate value: An 8-, 9-, or 13-bit constant given as an operand of an instruction that is using immediate addressing.

sign bit: The MSB of a value when it is seen by the CPU to indicate the sign (negative or positive) of the value.

sign extend: Fill the unused high order bits of a register with copies of the sign bit in that register.

sign-extension mode (SXM) bit: Bit 10 of status register ST1; enables or disables sign extension in the input shifter. It also differentiates between logic and arithmetic shifts of the accumulator.

single-access RAM: See *SARAM*.

slave phase: See *latch phase*.

software interrupt: An interrupt caused by the execution of an INTR, NMI, or TRAP instruction.

software stack: A program control feature that allows you to extend the hardware stack into data memory with the PSHD and POPD instructions. The stack can be directly stored and recovered from data memory, one word at time. This feature is useful for deep subroutine nesting or protection against stack overflow.

ST0 and ST1: See *status registers ST0 and ST1*.

stack: A block of memory reserved for storing return addresses for subroutines and interrupt service routines. The ’C24x stack is 16 bits wide and eight levels deep.

status registers ST0 and ST1: Two 16-bit registers that contain bits for determining processor modes, addressing pointer values, and indicating various processor conditions and arithmetic logic results. These registers can be stored into and loaded from data memory, allowing the status of the machine to be saved and restored for subroutines.

$\overline{\text{STRB}}$: *External access active strobe*. The ’C24x asserts $\overline{\text{STRB}}$ during accesses to external program, data, or I/O space.

SXM bit: See *sign-extension mode bit (SXM).*

T

TC bit: *Test/control flag bit.* Bit 11 of status register ST1; stores the results of test operations done in the central arithmetic logic unit (CALU) or the auxiliary register arithmetic unit (ARAU). The TC bit can be tested by conditional instructions.

temporary register (TREG): A 16-bit register that holds one of the operands for a multiply operation; the dynamic shift count for the LACT, ADDT, and SUBT instructions; or the dynamic bit position for the BITT instruction.

TOS: *Top of stack.* Top level of the 8-level last-in, first-out hardware stack.

TREG: See *temporary register (TREG).*

TTL: *Transistor-to-transistor logic.*

V

vector: See *interrupt vector.*

vector location: See *interrupt vector location.*

W

wait state: A CLKOUT1 cycle during which the CPU waits when reading from or writing to slower external memory.

wait-state generator: An on-chip peripheral that generates a limited number of wait states for a given off-chip memory space (program, data, or I/O). Wait states are set in the wait-state generator control register (WSGR).

$\overline{\text{WE}}$: *Write enable pin.* The ’C24x asserts $\overline{\text{WE}}$ to request a write to external program, data, or I/O space.

WSGR: *Wait-state generator control register.* This register, which is mapped to I/O memory, controls the wait-state generator.

X

XF bit: *XF-pin status bit.* Bit 4 of status register ST1 that is used to read or change the logic level on the XF pin.

XF pin: *External flag pin*. A general-purpose output pin whose status can be read or changed by way of the XF bit in status register ST1.

Z

zero fill: A way to fill the unused low or high order bits in a register by inserting 0s.

Appendix E

Summary of Updates in This Document

This appendix provides a summary of the updates in this version of the document. Updates within paragraphs appear in a **bold typeface**.

Rev. B Page:	Rev. C Page	Change or Add:
– – – –	– – – –	Changed the title on the cover and title page to: TMS320F/C24x DSP Controllers, CPU and Instruction Set Reference Guide. In addition to being revised, this version has been reorganized, and hence, differs significantly from the previous version (SPRU160B). Only major reorganizational changes are noted in this appendix.
3–1	3–1	Chapter 3 in revision *B* was *Central Processing Unit*. Chapter 3 in revision *C* is now *Memory and I/O Spaces.*
4–1	4–1	Chapter 4 in revision *B* was *Memory and I/O Spaces*. Chapter 4 in revision *C* is now *Central Processing Unit.*
6–1	6–1	Chapter 6 in revision *B* was *System Functions*. Chapter 6 in revision *C* is now *Addressing Modes.*
7–1	7–1	Chapter 7 in revision *B* was *Addressing Modes*. Chapter 7 in revision *C* is now *Assembly Language Instructions.*
8–1	8–1	Chapter 8 in revision *B* was *Assembly Language Instructions*. There is no Chapter 8 in revision *C*.

Index

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

R

S

T