TMS320C27x DSP CPU and Instruction Set Reference Guide

Preliminary

Literature Number: SPRU220D
March 1999

PRINTED WITH SOY INK

TEXAS INSTRUMENTS

Printed on Recycled Paper

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.

Copyright © 1999, Texas Instruments Incorporated

Preface

Read This First

About This Manual

This manual describes the central processing unit (CPU) and the assembly language instructions of the TMS320C27x 16-bit fixed-point digital signal processors (DSPs). It also describes emulation features available on these DSPs. A summary of the chapters and appendixes follows:

Chapter 1 **Architectural Overview**
This chapter introduces the T320C2700 DSP core that is at the heart of each TMS320C27x DSP. The chapter includes a memory map and a high-level description of the memory interface that connects the core with memory and peripheral devices.

Chapter 2 **Central Processing Unit**
This chapter describes the architecture, registers, and primary functions of the CPU. The chapter includes detailed descriptions of the flag and control bits in the most important CPU registers, status registers ST0 and ST1.

Chapter 3 **Interrupts and Reset**
This chapter describes the interrupts and how they are handled by the CPU. The chapter also explains the effects of a reset on the CPU and includes discussion of the automatic context save performed by the CPU prior to servicing an interrupt.

Chapter 4 **Pipeline**
This chapter describes the phases and operation of the instruction pipeline. The chapter is primarily for readers interested in increasing the efficiency of their programs by preventing pipeline delays.

Chapter 5 **Addressing Modes**
This chapter explains the modes by which the assembly language instructions accept data and access register and memory locations. The chapter includes a description of how addressing-mode information is encoded in opcodes.

Chapter 6 **Assembly Language Instructions**
This chapter provides summaries of the instruction set and detailed descriptions (including examples) for the instructions. The chapter includes an explanation of how 32-bit accesses are aligned to even addresses.

Chapter 7 **Emulation Features**
This chapter describes the TMS320C27x emulation features that can be used with only a JTAG port and two additional emulation pins.

Chapter 8 **Assembly Language Programming Tips**
This chapter introduces instructions and programming techniques that may help you manage your programs for the TMS320C27x.

Appendix A **Register Quick Reference**
This appendix is a concise central resource for information about the status and control registers of the CPU. The chapter includes figures that summarize the bit fields of the registers.

Appendix B **Submitting ROM Codes to TI**
This appendix describes the procedures for getting code-customized ROM in a Texas Instruments (TI™) DSP.

Appendix C **Design Considerations for Using XDS510 Emulator**
This appendix assists you in meeting the design requirements of the TI XDS510™ emulator with respect to designs that use the IEEE standard 1149.1.

Appendix D **Glossary**
This appendix explains abbreviations, acronyms, and special terminology used throughout this document.

Notational Conventions

This document uses the following conventions:

- The device number TMS320C27x is very often abbreviated as ’27x.
- Program examples are shown in a `special typeface`. Here is a sample line of program code:

  ```
  PUSH IER
  ```

- Portions of an instruction syntax that are in **bold** should be entered as shown; portions of a syntax that are in *italics* are variables indicating information that should be entered. Here is an example of an instruction syntax:

  MOV AR*x***, *–SP[***6bit***]**

  MOV is the instruction mnemonic. This instruction has two operands, indicated by **AR***x* and ***–SP[***6bit***]**. Where the variable *x* appears, you type a value from 0 to 5; where the *6bit* appears, you type a 6-bit constant. The rest of the instruction, including the square brackets, must be entered as shown.

- When braces enclose an operand, as in {operand}, the operand is optional. If you use an optional operand, you specify the information within the braces; you do not enter the braces themselves. In the following syntax, the operand << *shift* is optional:

  MOV ACC, *–SP[*6bit***]** {**<<** *shift* }

  For example, you could use either of the following instructions:

  ```
  MOV ACC, *–SP[5]
  MOV ACC, *–SP[5]<< 4
  ```

- In most cases, hexadecimal numbers are shown with a subscript of 16. For example, the hexadecimal number 40 would be shown as 40_{16}. An exception to this rule is a hexadecimal number in a code example; these hexadecimal numbers have the suffix h. For example, the number 40 in the following code is a hexadecimal 40.

  ```
  MOVB AR0,#40h
  ```

  Similarly, binary numbers usually are shown with a subscript of 2. For example, the binary number 4 would be shown as 0100_2. Binary numbers in example code have the suffix b. For example, the following code uses a binary 4.

  ```
  MOVB AR0,#0100b
  ```

- Bus signals and bits are sometimes represented with the following notations:

Notation	Description	Example
Bus(n:m)	Signals n through m of bus	PRDB(31:0) represents the 32 signals of the program-read data bus (PRDB).
Register(n:m)	Bits n through m of register	T(3:0) represents the 4 least significant bits of the T register.
Register(n)	Bit n of register	IER(4) represents bit 4 of the interrupt enable register (IER).

- Concatenated values are represented with the following notation:

Notation	Description	Example
x:y	x concatenated with y	AR1:AR0 is the concatenation of the 16-bit registers AR1 and AR0. AR0 is the low word. AR1 is the high word.

- If a signal is from an active-low pin, the name of the signal is qualified with an overbar (for example, $\overline{\text{INT1}}$). If a signal is from an active-high pin or from hardware inside the the DSP (in which case, the polarity is irrelevant), the name of the signal is left unqualified (for example, DLOGINT).

Related Documentation From Texas Instruments

The following books describe the TMS320C27x DSP 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.

TMS320C27xx Code Generation Tools Getting Started Guide (literature number SPRU213) describes how to install the TMS320C27xx assembly language tools and the C compiler for the TMS320C27xx device. The installation for MS-DOS™, SunOS™, and HP-UX™ 9.0x systems are covered.

TMS320C27xx Assembly Language Tools User's Guide (literature number SPRU211) describes the assembly language tools (assembler and other tools used to develop assembly language code), assembler directives, macros, common object file format, and symbolic debugging directives for the TMS320C27xx device.

TMS320C27xx Optimizing C Compiler User's Guide (literature number SPRU212) describes the TMS320C27xx C compiler. This C compiler accepts ANSI standard C source code and produces TMS320 assembly language source code for the TMS320C27xx device.

TMS320C27xx C Source Debugger User's Guide (literature number SPRU214) tells you how to invoke the TMS320C27xx 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.

TMS320C27xx Translation Assistant User's Guide (literature number SPRU278) describes the TMS320C27xx translation utility and how it fits in with the rest of the TMS320C27xx code development tools. It tells you how to use the translation assistant to translate code you already have for TMS320C2xx devices into code that will run on TMS320C27xx devices.

Trademarks

320 Hotline On-line is a trademark of Texas Instruments Incorporated.

HP-UX is a trademark of Hewlett-Packard Company.

IBM and PC are trademarks of International Business Machines Corporation.

Intel is a trademark of Intel Corporation.

MS-DOS is a registered trademark of Microsoft Corporation.

PAL® is a registered trademark of Advanced Micro Devices, Inc.

SunOS is a trademark of Sun Microsystems, Inc.

TI and XDS510 are trademarks of Texas Instruments Incorporated.

If You Need Assistance . . .

❑ World-Wide Web Sites

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

❑ North America, South America, Central America

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

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

❑ Europe, Middle East, Africa

European Product Information Center (EPIC) Hotlines:

Multi-Language Support	+33 1 30 70 11 69	Fax: +33 1 30 70 10 32

Email: epic@ti.com

Deutsch	+49 8161 80 33 11	or +33 1 30 70 11 68
English		+33 1 30 70 11 65
Francais		+33 1 30 70 11 64
Italiano		+33 1 30 70 11 67
EPIC Modem BBS		+33 1 30 70 11 99
European Factory Repair		+33 4 93 22 25 40
Europe Customer Training Helpline		Fax: +49 81 61 80 40 10

❑ Asia-Pacific

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

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

❑ Japan

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

❑ Documentation

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

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

Email: dsph@ti.com

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

Contents

1 Architectural Overview 1-1
Introduces the architecture and memory map of the T320C2700 DSP core.

1.1 Introduction to the Core 1-2
1.2 Components of the Core 1-3
1.2.1 Central Processing Unit (CPU) 1-3
1.2.2 Emulation Logic 1-4
1.2.3 Signals 1-5
1.3 Memory Map 1-6
1.3.1 Possible Maps for Block B0 1-7
1.3.2 Possible Maps for Block B1 1-9
1.3.3 Accessing Program Space and Data Space 1-10
1.4 Memory Interface 1-11
1.4.1 Address and Data Buses 1-11
1.4.2 Special Bus Operations 1-12
1.4.3 Alignment of 32-Bit Accesses to Even Addresses 1-13

2 Central Processing Unit 2-1
Describes the architecture, registers, and primary functions of the TMS320C27x CPU.

2.1 CPU Architecture 2-2
2.2 CPU Registers 2-4
2.2.1 Accumulator (ACC, AH, AL) 2-5
2.2.2 Multiplicand Register (T) 2-7
2.2.3 Product Register (P, PH, PL) 2-7
2.2.4 Data Page Pointer (DP) 2-8
2.2.5 Stack Pointer (SP) 2-9
2.2.6 Auxiliary Registers (AR0–AR5, XAR6, XAR7) 2-10
2.2.7 Program Counter (PC) 2-11
2.2.8 Status Registers (ST0, ST1) 2-12
2.2.9 Interrupt-Control Registers (IFR, IER, DBGIER) 2-12
2.3 Status Register ST0 2-13
2.4 Status Register ST1 2-19
2.5 Program Flow 2-24
2.5.1 Interrupts 2-24
2.5.2 Branches, Calls, and Returns 2-24
2.5.3 Repeating a Single Instruction 2-24
2.5.4 Instruction Pipeline 2-25
2.6 Multiply Operations 2-26
2.7 Shift Operations 2-27

3 Interrupts and Reset 3-1

Describes the TMS320C27x interrupts and how they are handled by the CPU. Also explains the effects of a hardware reset.

3.1 Interrupts Overview 3-2
3.2 Interrupt Vectors and Priorities 3-3
3.3 Maskable Interrupts: $\overline{\text{INT1}}$–$\overline{\text{INT14}}$, DLOGINT, and $\overline{\text{RTOSINT}}$ 3-5
3.3.1 Interrupt Flag Register (IFR) 3-6
3.3.2 Interrupt Enable Register (IER) and Debug Interrupt Enable Register (DBGIER) 3-7
3.4 Standard Operation for Maskable Interrupts 3-10
3.5 Nonmaskable Interrupts 3-16
3.5.1 INTR Instruction 3-16
3.5.2 TRAP Instruction 3-17
3.5.3 Hardware Interrupt $\overline{\text{NMI}}$ 3-20
3.6 Illegal-Instruction Trap 3-21
3.7 Hardware Reset ($\overline{\text{RS}}$) 3-22

4 Pipeline 4-1

Describes the phases and operation of the instruction pipeline.

4.1 Pipelining of Instructions 4-2
4.1.1 Decoupled Pipeline Segments 4-4
4.1.2 Instruction-Fetch Mechanism 4-4
4.1.3 Address Counters FC, IC, and PC 4-5
4.2 Visualizing Pipeline Activity 4-7
4.3 Freezes in Pipeline Activity 4-10
4.3.1 Wait States 4-10
4.3.2 Instruction-Not-Available Condition 4-10
4.4 Pipeline Protection 4-12
4.4.1 Protection During Reads and Writes to the Same Data-Space Location 4-12
4.4.2 Protection Against Register Conflicts 4-13
4.5 Avoiding Unprotected Operations 4-16
4.5.1 Unprotected Program-Space Reads and Writes 4-16
4.5.2 An Access to One Location That Affects Another Location 4-16
4.5.3 Write Followed By Read Protection Mode 4-17

5 Addressing Modes 5-1

Describes the operation and use of the TMS320C27x addressing modes.

5.1 Immediate Addressing Modes 5-2
5.1.1 Immediate-Constant Addressing Mode 5-2
5.1.2 Immediate-Pointer Addressing Mode 5-2
5.2 Direct Addressing Modes 5-4
5.2.1 DP Direct Addressing Mode 5-4
5.2.2 PAGE0 Direct Addressing Mode 5-5
5.3 Register Addressing Mode 5-6

5.4 Indirect Addressing Modes That Use the Stack Pointer 5-7
5.4.1 Stack-Pointer Indirect Addressing Mode 5-7
5.4.2 PAGE0 Stack Addressing Mode 5-8
5.5 Indirect Addressing Modes That Use Auxiliary Registers 5-10
5.5.1 Auxiliary-Register Indirect Addressing Mode 5-11
5.5.2 ARP Indirect Addressing Mode 5-13
5.5.3 XAR7 Indirect Addressing Mode 5-15
5.5.4 Circular Addressing Mode 5-16
A caution for those accessing the circular buffer with 32-bit operations 5-17
5.6 Summary of Indirect-Addressing Operands 5-18
5.7 Addressing-Mode Information in Opcodes 5-21
5.7.1 Opcodes for Immediate Addressing Modes 5-21
5.7.2 Opcodes for Register, Direct, and Indirect Addressing Modes 5-23

6 Assembly Language Instructions 6-1
Describes the assembly language instructions.. Includes summaries and detailed descriptions of the instructions.

6.1 Instruction Set Summaries 6-2
6.1.1 Alphabetical Summary by Mnemonic 6-2
6.1.2 Summary by Operation Type 6-9
6.1.3 Summary by Opcode 6-20
6.2 Alignment of 32-Bit Accesses to Even Addresses 6-31
6.3 How to Use the Instruction Descriptions 6-34
6.3.1 Syntax 6-34
6.3.2 Operands 6-36
6.3.3 Opcode 6-36
6.3.4 Description 6-37
6.3.5 Execution 6-37
6.3.6 Status Bits 6-38
6.3.7 Repeat 6-38
6.3.8 Words 6-38
6.3.9 Examples 6-38
6.4 Instruction Descriptions 6-39

7 Emulation Features 7-1
Explains features supported by the T320C2700 core for testing and debugging programs.

7.1 Overview of Emulation Features 7-2
7.2 Debug Interface 7-3
7.3 Debug Terminology 7-6
7.4 Execution Control Modes 7-7
7.4.1 Stop Mode 7-7
7.4.2 Real-Time Mode 7-9
Caution about breakpoints within time-critical interrupt service routines 7-11
7.4.3 Summary of Stop Mode and Real-Time Mode 7-11

7.5 Aborting Interrupts With the ABORTI Instruction 7-14
7.6 DT-DMA Mechanism 7-15
7.7 Analysis Breakpoints, Watchpoints, and Counter(s) 7-18
7.7.1 Analysis Breakpoints 7-18
7.7.2 Watchpoints 7-18
7.7.3 Benchmark Counter/Event Counter(s) 7-19
7.8 Data Logging 7-21
7.8.1 Creating a Data Logging Transfer Buffer 7-21
7.8.2 Accessing the Emulation Registers Properly 7-24
7.8.3 Data Log Interrupt (DLOGINT) 7-25
7.8.4 Examples of Data Logging 7-26
7.9 Sharing Analysis Resources 7-28
7.10 Diagnostics and Recovery 7-29

8 Assembly Language Programming Tips 8-1
Introduces special assembly language instructions and provides advice to help you with key operations of the TMS320C27xx.

8.1 Initializing the Processor After Reset 8-2
8.2 Performing Special Branch Operations 8-4
8.2.1 Branching to an Address Determined at Run Time 8-4
8.2.2 Creating a Fast Function Call and Return 8-5
8.2.3 Branching Based on a Single Condition 8-5
BANZ instruction 8-5
B and SB instructions 8-6
8.2.4 Branching Based on Multiple Conditions 8-8
8.3 Managing Interrupts 8-9
8.3.1 Assigning Interrupt Vectors 8-9
8.3.3 Using the Interrupt Acknowledge (IACK) Instruction 8-10
8.3.4 Changing the Values Saved During the Automatic Context Save 8-11
Review of the automatic context save 8-11
Reading and modifying the correct values 8-13
8.3.5 Using Nested Interrupts 8-14
8.3.6 Checking Interrupt Flags 8-16
8.4 Using the LOOPZ and LOOPNZ Instructions 8-17
8.4.1 Examples of the Loop Instructions 8-17
8.4.2 Preventing an Endless Loop 8-19
8.5 Managing Memory 8-20
8.5.1 Moving Blocks of Data 8-20
Moving blocks of data within data space 8-20
Moving blocks of data between program space and data space 8-21
8.5.2 Accessing Individual Bytes 8-22
8.5.3 Accessing an Array With a Base Pointer and an Index 8-23
8.5.4 Aligning Long Data to Even Addresses 8-25
8.5.5 Allocating and Deallocating Temporary Space on the Stack 8-26
8.5.6 Safely Mixing 16- and 32-Bit Values on the Stack 8-26
8.6 Implementing a Circular Buffer With Circular Addressing Mode 8-30

A Register Quick Reference A-1

Is a concise, central resource for information about the status and control registers of the TMS320C27x CPU.

A.1 Reset Values of and Instructions for Accessing the Registers A-2
A.2 Register Figures A-3
Status register ST0 . . . . . A-4
Status register ST1 . . . . . A-5
Interrupt flag register (IFR) A-6
Interrupt enable register (IER) A-7
Debug interrupt enable register (DBGIER) A-8

B Submitting ROM Codes to TI B-1

Explains the process for submitting custom program code to TI for designing masks for the on-chip ROM on a TMS320 DSP.

B.1 Scope B-2
B.2 Procedure B-3
B.2.1 Customer Required Information B-4
B.2.2 TI Performs ROM Receipt B-4
B.2.3 Customer Approves ROM Receipt B-5
B.2.4 TI Orders Masks, Manufactures, and Ships Prototypes B-5
B.2.5 Customer Approves Prototype B-5
B.2.6 Customer Release to Production B-5
B.3 Code Submittal B-6
B.4 Ordering B-7

C Design Considerations for Using XDS510 Emulator C-1

Describes the JTAG emulator cable, how to construct a 14-pin connector on your target system, and how to connect the target system to the emulator.

C.1 Designing Your Target System's Emulator Connector (14-Pin Header) C-2
C.2 Bus Protocol C-4
C.3 Emulator Cable Pod C-5
C.4 Emulator Cable Pod Signal Timing C-6
C.5 Emulation Timing Calculations C-7
C.6 Connections Between the Emulator and the Target System C-10
C.6.1 Buffering Signals C-10
C.6.2 Using a Target-System Clock C-12
C.6.3 Configuring Multiple Processors C-13
C.7 Physical Dimensions for the 14-Pin Emulator Connector C-14
C.8 Emulation Design Considerations C-16
C.8.1 Using Scan Path Linkers C-16
C.8.2 Emulation Timing Calculations for a Scan Path Linker (SPL) C-18
C.8.3 Using Emulation Pins C-20
C.8.4 Performing Diagnostic Applications C-24

D Glossary D-1

Figures

1–1 High-Level Conceptual Diagram of the Core 1-3
1–2 TMS320C27x High-Level Memory Map 1-6
1–3 Possible Maps for Block B0 1-8
1–4 Possible Maps for Block B1 1-9
2–1 Conceptual Block Diagram of the CPU 2-3
2–2 Individually Accessible Portions of the Accumulator 2-6
2–3 Individually Accessible Halves of the P Register 2-7
2–4 Pages of Data Memory 2-9
2–5 Address Reach of the Stack Pointer 2-9
2–6 Address Reach of the Auxiliary Registers 2-11
2–7 AR6 and AR7 2-11
2–8 Status Register ST0 2-12
2–9 Status Register ST1 2-12
2–10 Conceptual Diagram of Components Involved in Multiplication 2-26
3–1 Interrupt Flag Register (IFR) 3-6
3–2 Interrupt Enable Register (IER) 3-8
3–3 Debug Interrupt Enable Register (DBGIER) 3-9
3–4 Standard Operation for Maskable Interrupts 3-11
3–5 Functional Flow Chart for an Interrupt Initiated by the TRAP Instruction 3-17
5–1 Accessing Memory in Immediate-Pointer Addressing Mode 5-2
5–2 Pages of Data Memory 5-4
5–3 Accessing Data Memory in Stack-Pointer Indirect Addressing Mode 5-7
5–4 Accessing Data Memory in PAGE0 Stack Addressing Mode 5-8
5–5 Accessing Data Memory With the Auxiliary Registers 5-10
6–1 Relationship Among Auxiliary Register, Offset, and Accessed Byte During Move to or From AX.LSB or AX.MSB 6-161
7–1 JTAG Header to Interface a Target to the Scan Controller 7-3
7–2 Stop Mode Execution States 7-8
7–3 Real-time Mode Execution States 7-10
7–4 Stop Mode Versus Real-Time Mode 7-12
7–5 Process for Handling a DT-DMA Request 7-16
7–6 ADDRL (at Data-Space Address 00 0838_{16}) 7-22
7–7 ADDRH (at Data-Space Address 00 0839_{16}) 7-22
7–8 REFL (at Data-Space Address 00 $084A_{16}$) 7-22
7–9 REFH (at Data-Space Address 00 $084B_{16}$) 7-22
7–10 Valid Combinations of Analysis Resources 7-28

8–1 Accessing Values Saved to the Stack During Interrupt Processing 8-12
B–1 TMS320 ROM Code Prototype and Production Flowchart B-3
C–1 14-Pin Header Signals and Header Dimensions C-2
C–2 Emulator Cable Pod Interface C-5
C–3 Emulator Cable Pod Timings C-6
C–4 Emulator Connections Without Signal Buffering C-10
C–5 Emulator Connections With Signal Buffering C-11
C–6 Target-System-Generated Test Clock C-12
C–7 Multiprocessor Connections C-13
C–8 Pod/Connector Dimensions C-14
C–9 14-Pin Connector Dimensions C-15
C–10 Connecting a Secondary JTAG Scan Path to a Scan Path Linker C-17
C–11 EMU0/1 Configuration to Meet Timing Requirements of Less Than 25 ns C-21
C–12 Suggested Timings for the EMU0 and EMU1 Signals C-22
C–13 EMU0/1 Configuration With Additional AND Gate to Meet Timing Requirements of Greater Than 25 ns C-23
C–14 EMU0/1 Configuration Without Global Stop C-24
C–15 TBC Emulation Connections for n JTAG Scan Paths C-25

Tables

1–1 Summary of Bus Use During Data-Space and Program-Space Accesses 1-12
1–2 Special Bus Operations 1-13
2–1 CPU Register Summary 2-4
2–2 Available Operations for Shifting Values in the Accumulator 2-6
2–3 Product Shift Modes 2-8
2–4 Shift Operations 2-28
3–1 Interrupt Vectors and Priorities 3-3
3–2 Requirements for Enabling a Maskable Interrupt 3-6
3–3 Register Pairs Saved and SP Positions for Context Saves 3-13
3–4 Register Pairs Saved and SP Positions for Context Saves 3-19
3–5 Registers After Reset 3-22
5–1 Operands for Register Addressing Mode 5-6
5–2 Operands for Stack-Pointer Indirect Addressing Mode 5-8
5–3 Operands for Auxiliary-Register Indirect Addressing Mode 5-12
5–4 Operands for ARP Indirect Addressing Mode 5-14
5–5 Encoding of Register-, Direct-, and Indirect-Addressing Operands 5-23
6–1 Alphabetical Instruction Set Summary 6-3
6–2 Address Register Operations (AR0–AR7, XAR6, XAR7, DP, SP) 6-10
6–3 Push and Pop Stack Operations 6-10
6–4 AX (AH, AL) Operations 6-12
6–5 AX (AH, AL) Byte Operations 6-13
6–6 ACC Operations 6-13
6–7 ACC 32-Bit Operations 6-15
6–8 Operations on Memory or Register 6-15
6–9 Data Move Operations 6-16
6–10 Program Flow Operations 6-16
6–11 Math Operations 6-17
6–12 Control Operations 6-18
6–13 Emulation Operations 6-20
6–14 Instruction Set Summary by Opcode 6-20
6–15 Mechanisms That Generate 32-Bit-Wide Data Accesses 6-33
6–16 Conditions and Their Corresponding Opcode Segments and Flag Tests 6-78
6–17 Status Bits as Affected by the CLRC Instruction 6-85
6–18 Conditions and Their Corresponding Opcode Segments and Flag Tests 6-258
6–19 Status Bits as Affected by the SETC Instruction 6-265
7–1 14-Pin Header Signal Descriptions 7-4
7–2 Selecting Device Operating Modes By Using $\overline{\text{TRST}}$, EMU0, and EMU1 7-5
7–3 Interrupt Handling Information By Mode and State 7-13
7–4 Start Address and DMA Registers 7-23
7–5 End-Address Registers 7-24
7–6 Analysis Resources 7-28

8–1 Setting Pointers 8-2
8–2 Setting Key Status Bits 8-3
8–3 Conditions and Their Corresponding Flag Tests 8-7
A–1 Reset Values of the Status and Control Registers A-2
C–1 14-Pin Header Signal Descriptions C-3
C–2 Emulator Cable Pod Timing Parameters C-6

Examples

3–1 Typical ISR 3-15
4–1 Relationship Between Pipeline and Address Counters FC, IC, and PC 4-6
4–2 Diagramming Pipeline Activity 4-8
4–3 Simplified Diagram of Pipeline Activity 4-9
4–4 Conflict Between a Read From and a Write to Same Memory Location 4-13
4–5 Register Conflict 4-14
5–1 Circular Addressing Mode 5-16
5–2 Opcode Format for Immediate Addressing With an 8-Bit Constant 5-21
5–3 Opcode Format for Immediate Addressing With a 5-Bit Constant 5-22
5–4 Opcode Format for Immediate Addressing With a 16-Bit Constant 5-22
5–5 Opcode Format for Immediate Addressing With a 22-Bit Constant 5-22
5–6 Opcode Format for an Instruction Using Register Addressing Mode 5-24
5–7 Opcode Format for an Instruction Using DP Direct Addressing Mode 5-25
5–8 Opcode Format for an Instruction Using Auxiliary-Register Indirect Addressing Mode 5-25
5–9 Opcode Format for an Instruction Using Stack-Pointer Indirect Addressing Mode 5-26
6–1 32-Bit Read From Data-Memory 6-31
6–2 32-Bit Write to Data Memory 6-32
7–1 Initialization Code for Data Logging With Word Counter 7-26
7–2 Initialization Code for Data Logging With End Address 7-27
8–1 Assigned Interrupt Vectors 8-10
8–2 Enabling and Disabling Nested Interrupts 8-15
8–3 LOOPNZ Monitoring External Signal 8-18
8–4 LOOPZ With Indirect Addressing 8-18
8–5 Using a Timer to Prevent Endless Looping of a Loop Instruction 8-19
8–6 Using 3-Bit Indexes for an 8-Element Array 8-24
8–7 Using AR1 as the Index for a Large Array 8-25
8–8 32-Bit Write Operation Overwriting an Existing Value 8-27
8–9 Incrementing SP by 1 to Prevent Overwrite 8-28
8–10 Using ASP Instruction to Prevent Overwrite 8-29
8–11 Using a Circular Buffer 8-32
C–1 Key Timing for a Single-Processor System Without Buffers C-8
C–2 Key Timing for a Single- or Multiple-Processor System With Buffered Input and Output C-8
C–3 Key Timing for a Single-Processor System Without Buffering (SPL) C-19
C–4 Key Timing for a Single- or Multiprocessor-System With Buffered Input and Output (SPL) C-19

Chapter 1

Architectural Overview

Each TMS320C27x ('27x) DSP contains a core. The core is responsible for the central control activities of a '27x DSP, activities such as address generation, arithmetic operations, data transfers, and system monitoring. This chapter introduces the components of the core, the memory map, and the memory interface.

Topic **Page**

1.1	Introduction to the Core	1-2
1.2	Components of the Core	1-3
1.3	Memory Map	1-6
1.4	Memory Interface	1-11

1.1 Introduction to the Core

The core is a low-cost 16-bit fixed-point digital signal processor (DSP). This device draws from the best features of digital signal processing; reduced instruction set computing (RISC); and microcontroller architectures, firmware, and tool sets. The DSP features include a modified Harvard architecture and circular addressing. The RISC features are single-cycle instruction execution, register-to-register operations, and modified Harvard architecture (usable in Von Neumann mode). The microcontroller features include ease of use through an intuitive instruction set, byte packing and unpacking, and bit manipulation.

The modified Harvard architecture of the core enables instruction and data fetches to be performed in parallel. The CPU can read instructions and data while it writes data simultaneously to maintain the single-cycle instruction operation across the pipeline. The CPU does this over six separate address/data buses.

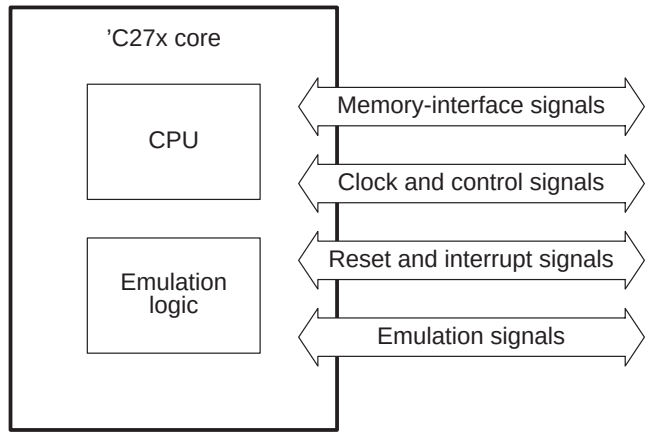
1.2 Components of the Core

As shown in Figure 1–1, the core contains:

- A CPU for generating data- and program-memory addresses; decoding and executing instructions; performing arithmetic, logical, and shift operations; and controlling data transfers among CPU registers, data memory, and program memory
- Emulation logic for monitoring and controlling various parts and functionalities of the DSP and for testing device operation
- Signals for interfacing with memory and peripherals, clocking and controlling the CPU and the emulation logic, showing the status of the CPU and the emulation logic, and using interrupts

The core does not contain memory, a clock generator, or peripheral devices. For information about interfacing to these items, see the data sheet that corresponds to your DSP.

Figure 1–1. High-Level Conceptual Diagram of the Core



1.2.1 Central Processing Unit (CPU)

The CPU is discussed in more detail in Chapter 2, but following is a list of its major features:

- Protected pipeline. The CPU implements an 8-phase pipeline that prevents a read from and a write to the same location from occurring out of order.
- Independent register space. The CPU contains registers that are not mapped to data space. These registers function as system-control

registers, math registers, and data pointers. The system-control registers are accessed by special instructions. The other registers are accessed by special instructions or by a special addressing mode (register addressing mode).

- ❑ Arithmetic logic unit (ALU). The 32-bit ALU performs 2s-complement arithmetic and Boolean logic operations.
- ❑ Address register arithmetic unit (ARAU). The ARAU generates data-memory addresses and increments or decrements pointers in parallel with ALU operations.
- ❑ Barrel shifter. This shifter performs all left and right shifts of data. It can shift data to the left by up to 16 bits and to the right by up to 16 bits.
- ❑ Multiplier. The multiplier performs 16-bit × 16-bit 2s-complement multiplication with a 32-bit result. The multiplication can be performed with two signed numbers, two unsigned numbers, or one signed number and one unsigned number.

1.2.2 Emulation Logic

The emulation logic includes the following features. For more details about these features, see Chapter 7, *Emulation Features*.

- ❑ Debug-and-test direct memory access (DT-DMA). A debug host can gain direct access to the content of registers and memory by taking control of the memory interface during unused cycles of the instruction pipeline.
- ❑ Data logging. The emulation logic enables application-initiated transfers of memory contents between the ’27x and a debug host.
- ❑ A counter for performance benchmarking
- ❑ Multiple debug events. Any of the following *debug events* can cause a break in program execution:
  - ■ A breakpoint initiated by the ESTOP0 or ESTOP1 instruction
  - ■ An access to a specified program-space or data-space location
  - ■ A request from the debug host or other hardware

  When a debug event causes the ’27x to enter the debug-halt state, the event is called a *break event*.
- ❑ Real-time mode of operation. When the ’27x is in this mode and a break event occurs, the main body of program code comes to a halt, but time-critical interrupts can still be serviced.

1.2.3 Signals

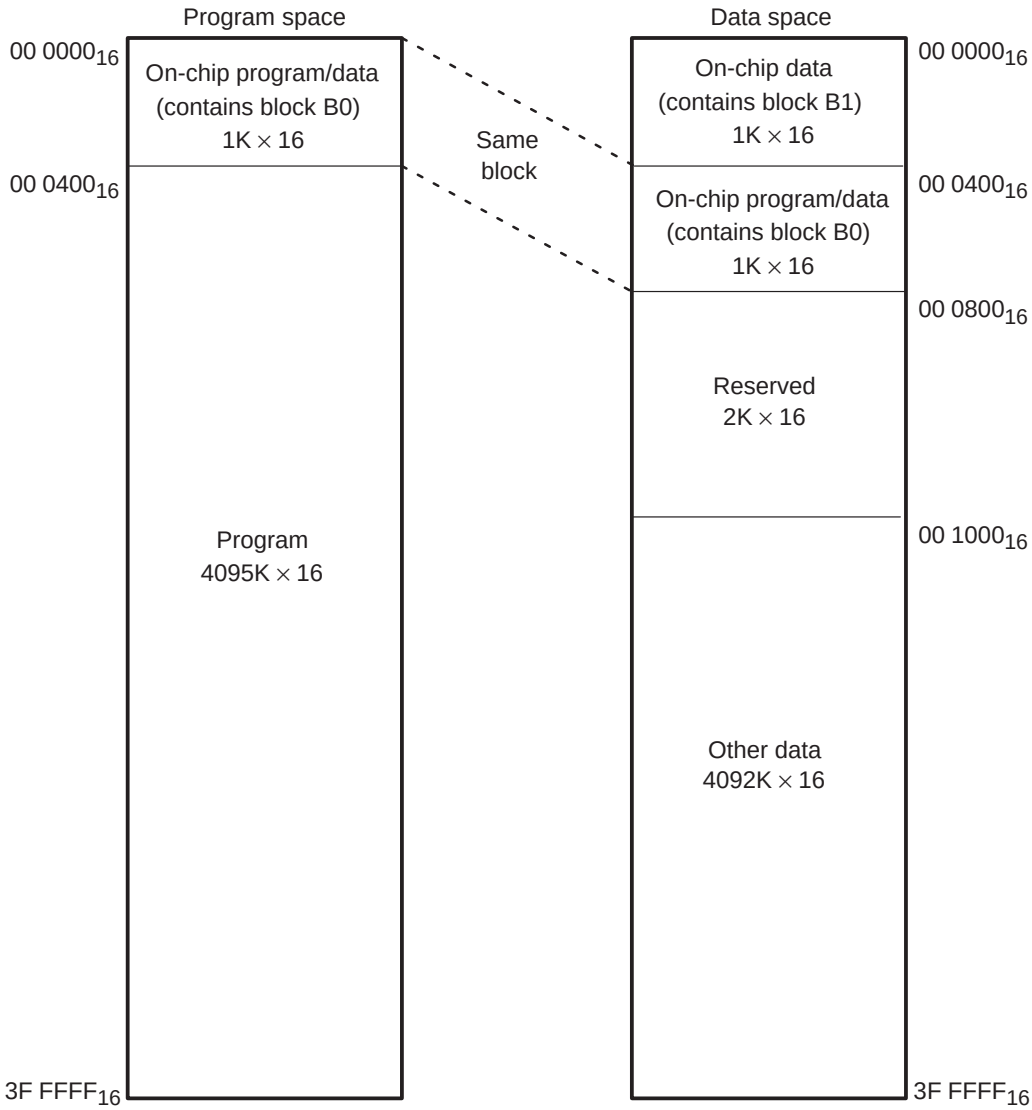
As shown in Figure 1–1 (page 1-3), the core has four main types of signals:

- Memory-interface signals. These signals transfer data among the core, memory, and peripherals; indicate program-memory accesses and data-memory accesses; and differentiate between accesses of different sizes (16-bit or 32-bit).
- Clock and control signals. These provide clocking for the CPU and the emulation logic, and they are used to control and monitor the CPU.
- Reset and interrupt signals. These are used for generating a hardware reset and interrupts, and for monitoring the status of interrupts.
- Emulation signals. These signals are used for testing and debugging.

1.3 Memory Map

The core contains no memory, but can access memory elsewhere on the ’27x DSP or outside the DSP. The ’27x uses 22-bit addresses, supporting a total address reach of 4M words (1 word = 16 bits) in both program space and data space. Figure 1–2 shows a high-level view of how addresses are allocated in program space and data space.

Figure 1–2. TMS320C27x High-Level Memory Map



The memory map in Figure 1–2 has been divided into the following segments. (For specific details about each of the map segments, see the data sheet for your DSP.)

- **On-chip program/data.** Addresses 00 0000_{16}–00 $03FF_{16}$ in program space and 00 0400_{16}–00 $07FF_{16}$ in data space are for use with a single on-chip block of program/data memory. For example, you access the same location with address 00 0041_{16} in program space or address 00 0441_{16} in data space. Within these addresses resides a block B0, which is 256, 512, or 1K words. For more details about B0, see section 1.3.1.

- **Program.** Addresses 00 0400_{16}–3F $FFFF_{16}$ in program space are available for additional program memory inside and/or outside the DSP.

- **On-chip data.** Addresses 00 0000_{16}–00 $03FF_{16}$ in data space are for a block of data-memory on board the DSP. This block, called B1, may be 256, 512, or 1K words. For an explanation of how B1 affects the memory map, see section 1.3.2 on page 1-9.

- **Reserved.** Addresses 00 0800_{16}–00 $0CFF_{16}$ in data space are reserved for special registers. Some of the addresses in the range 000800_{16}–$0009FF_{16}$ are available for accessing memory-mapped emulation registers.

- **Other data.** Addresses 00 $0D00_{16}$–3F $FFFF_{16}$ in data space are available for additional data memory inside and/or outside the DSP.

Sixty-four addresses in program space are set aside for a table of 32 interrupt vectors. The vectors can be mapped to the top or bottom of program space by way of the VMAP bit. As shown in Figure 1–3, if VMAP = 0, the vectors are mapped to addresses 00 0000_{16}–00 $003F_{16}$; they are also mapped to addresses 00 0400_{16}–00 $043F_{16}$ in data space. When VMAP = 1, the vectors are only mapped to program space at addresses 3F $FFC0_{16}$–3F $FFFF_{16}$. For more information about the vectors, see section 3.2, *Interrupt Vectors*, on page 3-3.

1.3.1 Possible Maps for Block B0

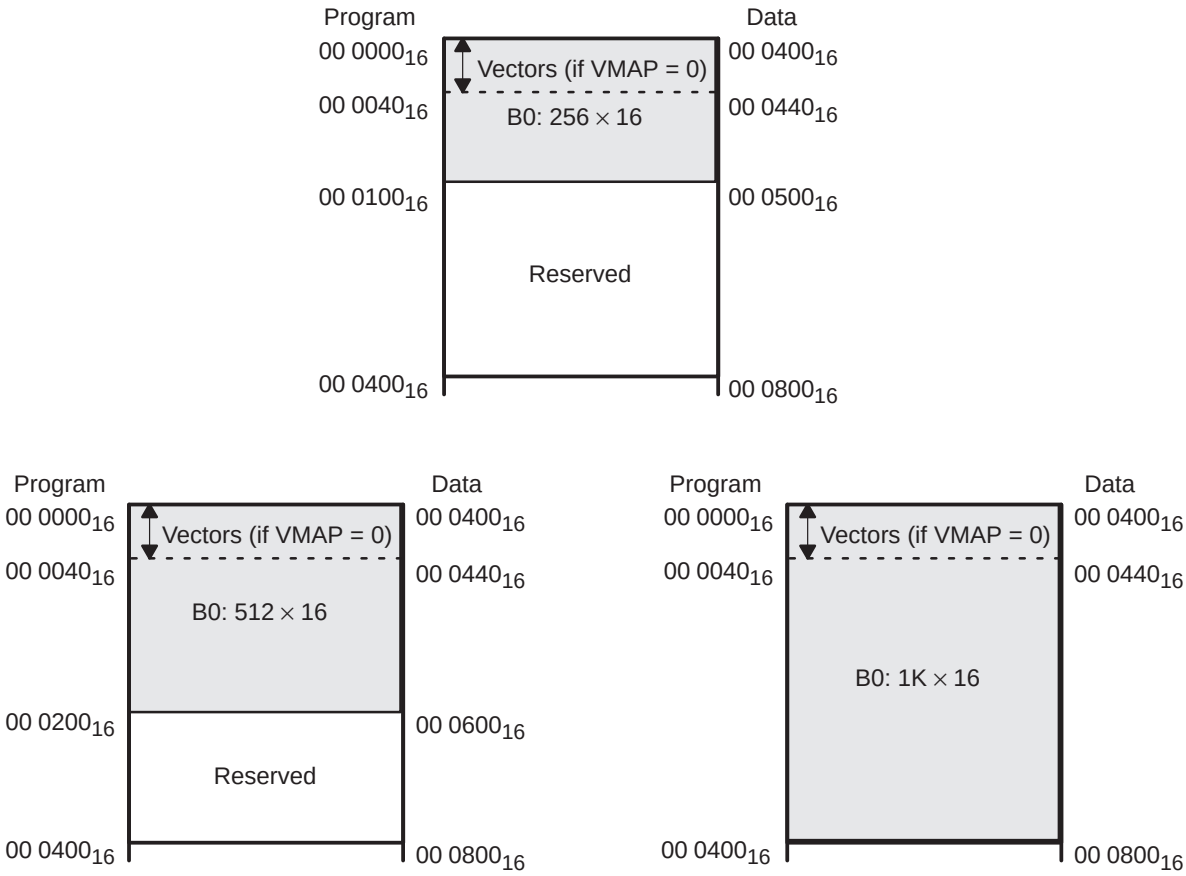
The possible maps for block B0 are shown in Figure 1–3. B0 can be 256, 512, or 1K words, depending on the chosen design. If B0 has fewer than 1024 (1K) words, some of the program/data addresses are reserved (not available for use).

The 1K block of memory shown in Figure 1–3, whether all used for B0 or partially reserved, can be addresses in program space or data space. For

example, you access the same location with address 00 0041_{16} in program space or address 00 0441_{16} in data space.

As shown in Figure 1–3, if VMAP = 0, the vectors are mapped to program-space addresses 00 0000_{16}–00 $003F_{16}$ and to data-space addresses 00 0400_{16}–00 $043F_{16}$. When VMAP = 1, the vectors are only mapped to program space at addresses 3F $FFC0_{16}$–3F $FFFF_{16}$. For more information about the vectors, see section 3.2, *Interrupt Vectors*, on page 3-3.

Figure 1–3. Possible Maps for Block B0

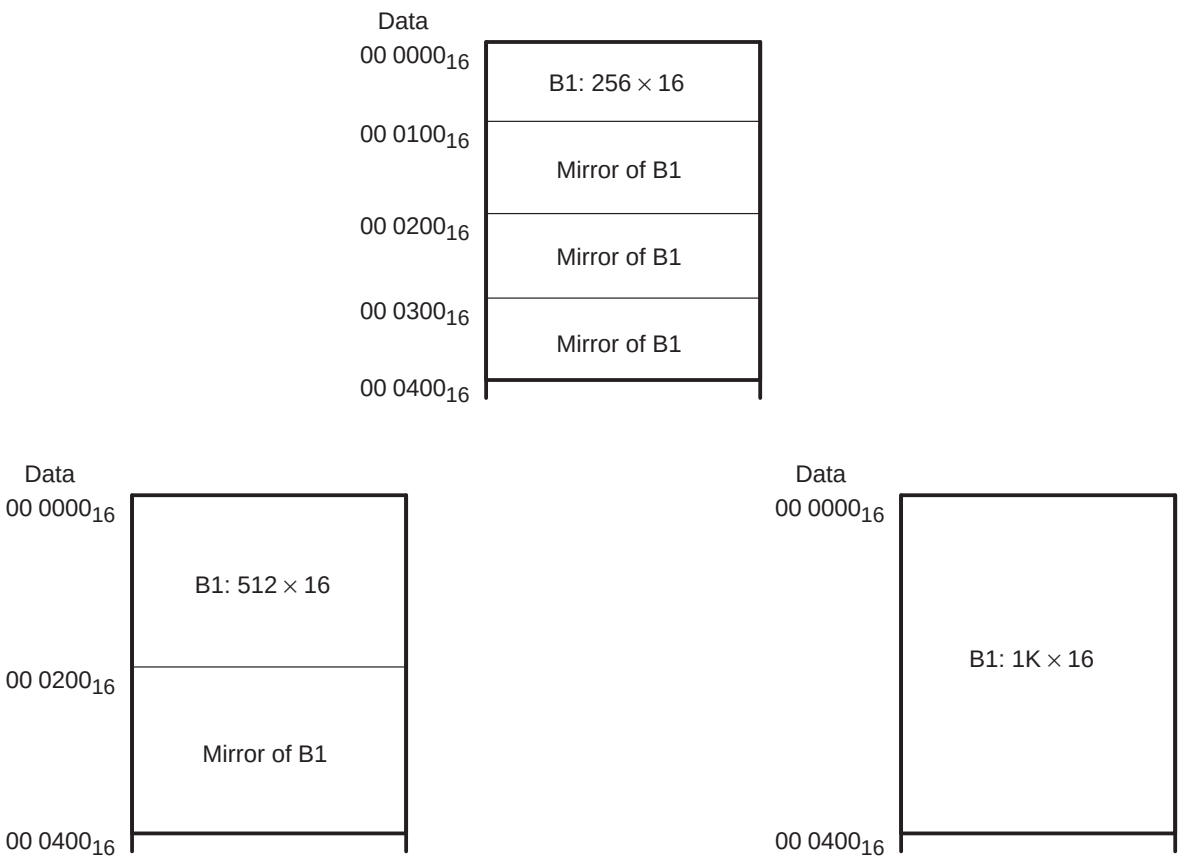


1.3.2 Possible Maps for Block B1

Figure 1–4 shows the possible maps for block B1, which resides only in data space. B1 is 256, 512, or 1K words, depending on the chosen design. If B1 has 256 words, mirrors of B1 are in the address ranges $00\ 0100_{16}$–$00\ 01FF_{16}$, $00\ 0200_{16}$–$00\ 02FF_{16}$, and $00\ 0300_{16}$–$00\ 03FF_{16}$. As an example of what this means, assume you want to access the first word of B1. You could access this word at any one of the following addresses: $00\ 0000_{16}$, $00\ 0100_{16}$, $00\ 0200_{16}$, or $00\ 0300_{16}$.

If B1 has 512 words, there is one mirror of B1 at addresses $00\ 0200_{16}$–$00\ 0400_{16}$. If B1 has 1K words, it uses all the available addresses.

Figure 1–4. Possible Maps for Block B1



1.3.3 Accessing Program Space and Data Space

The ’27x supports long call and branch operations and a 22-bit program counter (PC), enabling you to access program space as one linear memory space. The ’27x also supports offset branching, which enables code to be relocated at run time.

The ’27x supports a 16-bit stack pointer (SP) that points to data memory. Using this register, you can create a stack within the first 64K of data space (addresses 00 0000_{16}–00 $FFFF_{16}$). SP is incremented such that the stack grows from low memory to high memory and SP always points to the next empty location. After a hardware reset, SP points to address 00 0000_{16}. For more information about the stack, see section 2.2.5, *Stack Pointer (SP)*, on page 2-9.

Data space is supported by a rich set of addressing modes. Indirect addressing modes use 16-bit pointers (the SP and the auxiliary registers AR0–AR5) to access the low 64K of data space and 22-bit pointers (auxiliary registers XAR6 and XAR7) to access the full range of data space. Direct addressing modes access all of data space by concatenating a 16-bit data page number with a 6-bit offset that is embedded in an instruction. The addressing modes are described in Chapter 5.

Instructions are included to move data between program space and data space. The PREAD instruction can be used to transfer a program-memory value to data memory. The PWRITE instruction can transfer a data-memory value to program memory. If the source and destination memory blocks are mapped to both program space and data space, you can simply use syntaxes of the MOV instruction to transfer data between the blocks. Information about these instructions can be found in Chapter 6, *Assembly Language Instructions*.

1.4 Memory Interface

The ’27x memory map is accessible outside the core by the memory interface, which connects the core logic to memories, peripherals, or other interfaces. The memory interface includes separate buses for program space and data space. This means an instruction can be fetched from program memory while data memory is being accessed.

The interface also includes signals that indicate the type of read or write being requested by the CPU. These signals can select a specified memory block or peripheral for a given bus transaction. In addition to 16-bit and 32-bit accesses, the ’27x supports special byte-access instructions which can access the least significant byte (LSByte) or most significant byte (MSByte) of an addressed word. Strobe signals indicate when such an access is occurring on a data bus.

1.4.1 Address and Data Buses

The memory interface has three 22-bit address buses:

PAB *Program address bus.* The PAB carries addresses for reads and writes from program space.

DRAB *Data-read address bus.* The DRAB carries addresses for reads from data space.

DWAB *Data-write address bus.* The DWAB carries addresses for writes to data space.

The memory interface also has three 32-bit data buses:

PRDB *Program-read data bus.* The PRDB carries instructions or data during reads from program space.

DRDB *Data-read data bus.* The DRDB carries data during reads from data space.

DWDB *Data-/Program-write data bus.* The DWDB carries data during writes to data space or program space.

Table 1–1 summarizes how these buses are used during accesses.

Table 1–1. Summary of Bus Use During Data-Space and Program-Space Accesses

Access Type	Address Bus	Data Bus
Read from program space	PAB	PRDB
Read from data space	DRAB	DRDB
Write to program space	PAB	DWDB
Write to data space	DWAB	DWDB

A program-space read and a program-space write cannot happen simultaneously because both use the PAB. Similarly, a program-space write and a data-space write cannot happen simultaneously because both use the DWDB. Transactions that use different buses can happen simultaneously. For example, the CPU can read from program space (using PAB and PRDB), read from data space (using DRAB and DRDB), and write to data space (using DWAB and DWDB) at the same time.

1.4.2 Special Bus Operations

Typically, PAB and PRDB are used only for reading instructions from program space, and DWDB is used only for writing data to data space. However, the instructions in Table 1–2 are exceptions to this behavior. For more details about using these instructions, see Chapter 6, *Assembly Language Instructions*.

Table 1–2. Special Bus Operations

Instruction	Special Bus Operation
PREAD	This instruction reads a data value rather than an instruction from program space. It then transfers that value to data space or a register. For the read from program space, the core places the source address on the program address bus (PAB), sets the appropriate program-space select signals, and reads the data value from the program-read data bus (PRDB).
PWRITE	This instruction writes a data value to program space. The value is read from from data space or a register. For the write to program space, the core places the destination address on the program address bus (PAB), sets the appropriate program-space select signals, and writes the data value to the data-/program-write data bus (DWDB).
MAC	As part of its operation, this instruction multiplies two data values, one of which is read from program space. For the read from program space, the core places the program-space source address on the program address bus (PAB), sets the appropriate program-space select signals, and reads the program data value from the program read data bus (PRDB).

1.4.3 Alignment of 32-Bit Accesses to Even Addresses

The ’27x core expects memory wrappers or peripheral-interface logic to align any 32-bit read or write to an even address. If the address-generation logic generates an odd address, the CPU must begin reading or writing at the previous even address. This alignment does not affect the address values generated by the address-generation logic.

Most instruction fetches from program space are performed as 32-bit read operations and are aligned accordingly. However, alignment of instruction fetches are effectively invisible to a programmer. When instructions are stored to program space, they do not have to be aligned to even addresses. Instruction boundaries are decoded within the core.

You need to be concerned with alignment when using instructions that perform 32-bit reads from or writes to data space. Section 6.2, *Alignment of 32-bit Accesses to Even Addresses*, provides examples of alignment of data-space accesses and lists the operations that are subject to this alignment.

Chapter 2

Central Processing Unit

The central processing unit (CPU) is responsible for controlling the flow of a program and the processing of instructions. It performs arithmetic, Boolean-logic, multiply, and shift operations. When performing signed math, the CPU uses 2s-complement notation. This chapter describes the architecture, registers, and primary functions of the central processing unit (CPU).

Topic **Page**

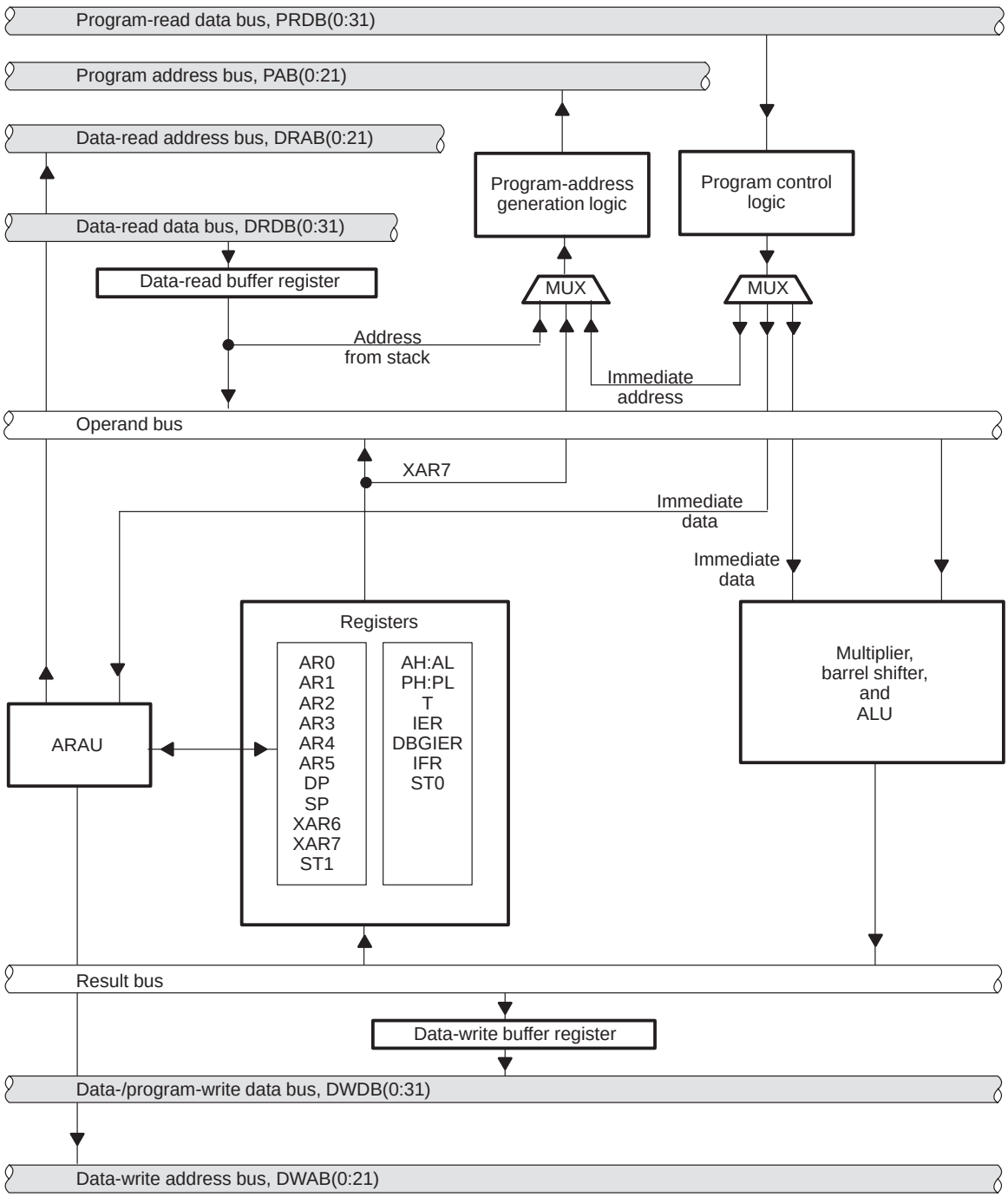
2.1	CPU Architecture	2-2
2.2	CPU Registers	2-4
2.3	Status Register ST0	2-13
2.4	Status Register ST1	2-19
2.5	Program Flow	2-24
2.6	Multiply Operations	2-26
2.7	Shift Operations	2-27

2.1 CPU Architecture

Figure 2–1 shows the major blocks and data paths of the ’27xx CPU. It does not reflect the actual silicon implementation. The shaded buses are memory-interface buses that are external to the CPU. The operand bus supplies the values for multiplier, shifter, and ALU operations, and the result bus carries the results to registers and memory. The main blocks of the CPU are:

- **Program control logic.** This logic stores a queue of instructions that have been fetched from program memory by way of the program-read bus (PRDB). It also decodes these instructions and passes commands and constant data to other parts of the CPU.
- **Program address generation logic.** This logic generates the addresses used to fetch instructions or data from program memory and places each address on the program address bus (PAB). As part of its operation, the program-address generation logic uses several counters, addresses supplied by the program control logic, and addresses from the extended auxiliary register XAR7.
- **Address register arithmetic unit (ARAU).** The ARAU generates addresses for values that must be fetched from data memory. For a data read, it places the address on the data-read address bus (DRAB); for a data write, it loads the data-write address bus (DWAB). The ARAU also increments or decrements the stack pointer (SP) and the auxiliary registers (AR0, AR1, AR2, AR3, AR4, AR5, XAR6, and XAR7).
- **Arithmetic logic unit (ALU).** The 32-bit ALU performs 2s-complement arithmetic and Boolean logic operations. Before doing its calculations, the ALU accepts data from registers, from data memory, or from the program control logic. The ALU saves results to a register or to data memory. Inputs from memory are received from the data-read data bus (DRDB). Results that go to data memory are sent to the data-/program-write data bus (DWDB).
- **Multiplier.** The multiplier performs 16-bit $\times$ 16-bit 2s-complement multiplication with a 32-bit result. In conjunction with the multiplier, the ’27xx uses the 16-bit multiplicand register (T), the 32-bit product register (P), and the 32-bit accumulator (ACC). The T register supplies one of the values to be multiplied. The result of the multiplication can be sent to the P register or to ACC.
- **Barrel shifter.** The shifter performs all the shifts on the ’27xx. It can shift data to the left by up to 16 bits and to the right by up to 16 bits. The amount and direction of the shift is determined by the particular instruction that causes the shift.

Figure 2–1. Conceptual Block Diagram of the CPU



2.2 CPU Registers

Table 2–1 lists the main CPU registers and their values after reset. Sections 2.2.1 through 2.2.9 describe the registers in more detail.

Table 2–1. CPU Register Summary

Register	Size	Description	Value After Reset
ACC	32 bits	Accumulator	$0000\ 0000_{16}$
AH	16 bits	High half of ACC	0000_{16}
AL	16 bits	Low half of ACC	0000_{16}
AR0	16 bits	Auxiliary register 0	0000_{16}
AR1	16 bits	Auxiliary register 1	0000_{16}
AR2	16 bits	Auxiliary register 2	0000_{16}
AR3	16 bits	Auxiliary register 3	0000_{16}
AR4	16 bits	Auxiliary register 4	0000_{16}
AR5	16 bits	Auxiliary register 5	0000_{16}
XAR6	22 bits	Extended auxiliary register 6	$00\ 0000_{16}$
AR6	16 bits	16 LSBs of XAR6	0000_{16}
XAR7	22 bits	Extended auxiliary register 7	$00\ 0000_{16}$
AR7	16 bits	16 LSBs of XAR7	0000_{16}
DP	16 bits	Data page pointer	0000_{16}

† The value in the PC after reset depends on the value in the VMAP bit at the time reset is initiated. The VMAP bit is bit 3 of ST1. The PC will be loaded with the reset vector at either address $00\ 0000_{16}$ or address $3F\ FFC0_{16}$.

‡ The value in ST1 after reset depends, in part, on the level on the VMAP input signal at the time reset is initiated. If the VMAP signal is low, bit 3 (VMAP) is cleared. If the VMAP signal is high, bit 3 is set.

Table 2–1. CPU Register Summary (Continued)

Register	Size	Description	Value After Reset
IFR	16 bits	Interrupt flag register	0000_{16}
IER	16 bits	Interrupt enable register	0000_{16}
DBGIER	16 bits	Debug interrupt enable register	0000_{16}
P	32 bits	Product register	0000 0000_{16}
PH	16 bits	High half of P	0000_{16}
PL	16 bits	Low half of P	0000_{16}
PC†	22 bits	Program counter	If VMAP bit = 0 PC = [00 0000_{16}] If VMAP bit = 1 PC = [3F $FFC0_{16}$]
SP	16 bits	Stack pointer	0000_{16}
ST0	16 bits	Status register 0	0000_{16}
ST1‡	16 bits	Status register 1	If VMAP input signal low ST1 = 0003_{16} If VMAP input signal high ST1 = $000B_{16}$
T	16 bits	Multiplicand register	0000_{16}

† The value in the PC after reset depends on the value in the VMAP bit at the time reset is initiated. The VMAP bit is bit 3 of ST1. The PC will be loaded with the reset vector at either address 00 0000_{16} or address 3F $FFC0_{16}$.

‡ The value in ST1 after reset depends, in part, on the level on the VMAP input signal at the time reset is initiated. If the VMAP signal is low, bit 3 (VMAP) is cleared. If the VMAP signal is high, bit 3 is set. Bits 1 (DBGM) and 0 (INTM) are always set by a reset.

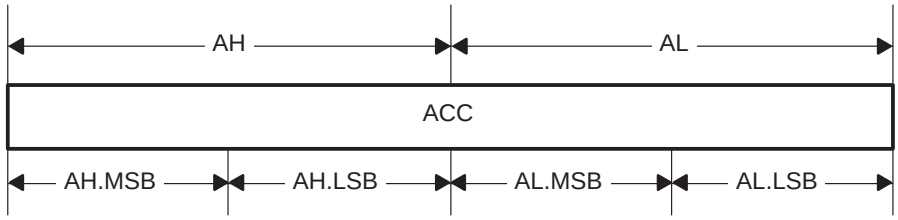
2.2.1 Accumulator (ACC, AH, AL)

The accumulator (ACC) is the main working register for the device. It is the destination for all ALU operations except those which operate directly on memory or registers. ACC supports single-cycle move, add, subtract, and compare operations from 32-bit-wide data memory. It can also accept the 32-bit result of a multiplication operation.

The halves and quarters of the ACC can also be accessed (see Figure 2–2). ACC can be treated as two independent 16-bit registers: AH (high 16 bits) and AL (low 16 bits). The bytes within AH and AL can also be accessed

independently. Special byte-move instructions load and store the most significant byte or least significant byte of AH or AL. This enables efficient byte packing and unpacking.

Figure 2–2. Individually Accessible Portions of the Accumulator



AH = ACC (31:16)
AH.MSB = ACC (31:24)
AH.LSB = ACC (23:16)

AL = ACC (15:0)
AL.MSB = ACC (15:8)
AL.LSB = ACC (7:0)

The accumulator has the following associated status bits. For the details on these bits, see section 2.3, *Status Register ST0*.

- ❏ Overflow mode bit (OVM)
- ❏ Sign-extension mode bit (SXM)
- ❏ Test/control flag bit (TC)
- ❏ Carry bit (C)
- ❏ Zero flag bit (Z)
- ❏ Negative flag bit (N)
- ❏ Latched overflow flag bit (V)
- ❏ Overflow counter bits (OVC)

Table 2–2 shows the ways to shift the content of AH, AL, or ACC.

Table 2–2. Available Operations for Shifting Values in the Accumulator

Register	Shift Direction	Shift Type	Instruction
ACC	Left	Logical	LSL
		Rotation	ROL
	Right	Arithmetic	SFR with SXM = 1
		Logical	SFR with SXM = 0
		Rotation	ROR
AH or AL	Left	Logical	LSL
	Right	Arithmetic	ASR
		Logical	LSR

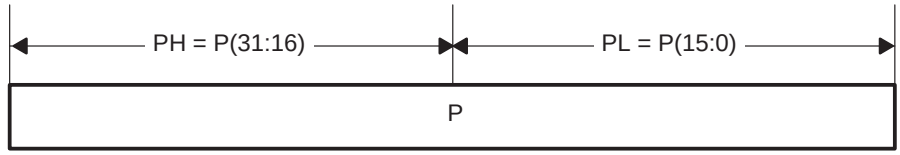
2.2.2 Multiplicand Register (T)

The multiplicand register (T register) is primarily used to store a 16-bit signed integer value prior to a multiply operation. During every multiplication, one of the multiplicands is taken from the T register. For some left and right barrel-shift operations, the shift amount (0 to 15) is determined by the four least significant bits of the T register. In these shift operations, the 12 most significant bits of T are ignored. The contents of T can be stored back to memory.

2.2.3 Product Register (P, PH, PL)

The product register (P register) is typically used to hold the 32-bit result of a 16×16 multiplication. It can also be loaded directly from a 16- or 32-bit data-memory location, a 16-bit constant, the 32-bit ACC, or a 16-bit addressable CPU register. The P register can be treated as a 32-bit register or as two independent 16-bit registers: PH (high 16 bits) and PL (low 16 bits); see Figure 2–3.

Figure 2–3. Individually Accessible Halves of the P Register



When an instruction accesses P, PH, or PL, all 32-bits are copied to the ALU-shifter block, where the barrel shifter may perform a left shift, a right shift, or no shift. Then the entire 32 bits (P) or the high or low 16 bits (PH or PL) are used in an ALU calculation or stored. The action of the shifter is determined by the product shift mode (PM) bits in status register ST0. Table 2–3 shows the possible PM values and the corresponding product shift modes. When the barrel shifter performs a left shift, the low order bits are filled with zeros. When the shifter performs a right shift, the P register value is sign extended. Instructions that use PH or PL as operands ignore the product shift mode.

Table 2–3. Product Shift Modes

PM Value	Product Shift Mode
000_2	Left shift by 1
001_2	No shift
010_2	Right shift by 1
011_2	Right shift by 2
100_2	Right shift by 3
101_2	Right shift by 4
110_2	Right shift by 5
111_2	Right shift by 6

2.2.4 Data Page Pointer (DP)

In the direct addressing modes, data memory is addressed in blocks of 64 words called *data pages*. The entire 4M words of data memory consists of 65 536 data pages labeled 0 through 65 535, as shown in Figure 2–4. In DP direct addressing mode, the 16-bit data page pointer (DP) holds the current data page number. You change the data page by loading the DP with a new number. For information about the direct addressing modes, see section 5.2 on page 5-4.

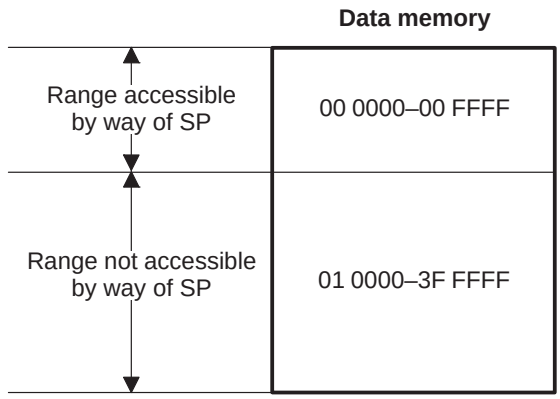
Figure 2–4. Pages of Data Memory

Data page	Offset	Data memory
00 0000 0000 0000 00 : 00 0000 0000 0000 00	00 0000 : 11 1111	Page 0: 00 0000–00 003F
00 0000 0000 0000 01 : 00 0000 0000 0000 01	00 0000 : 11 1111	Page 1: 00 0040–00 007F
00 0000 0000 0000 10 : 00 0000 0000 0000 10	00 0000 : 11 1111	Page 2: 00 0080–00 00BF
. . . . . .	. . . . . .	. . . .
11 1111 1111 1111 11 : 11 1111 1111 1111 11	00 0000 : 11 1111	Page 65 535: 3F FFC0–3F FFFF

2.2.5 Stack Pointer (SP)

The stack pointer (SP) enables the use of a software stack in data memory. The stack pointer has only 16 bits and can only address the low 64K of data space (see Figure 2–5). When the SP is used, the upper six bits of the 22-bit address are forced to 0. (For information about addressing modes that use the SP, see section 5.4 on page 5-7.). After reset, SP points to address 00 0000_{16}.

Figure 2–5. Address Reach of the Stack Pointer



The operation of the stack is as follows:

- The stack grows from low memory to high memory.
- The SP always points to the next empty location in the stack.
- At reset, the SP is cleared, so that it points to address 00 0000_{16}.
- When 32-bit values are saved to the stack, the least significant 16 bits are saved first, and the most significant 16 bits are saved to the next higher address (little endian format).
- When 32-bit operations read or write a 32-bit value, the ’27xx core expects the memory wrapper or peripheral-interface logic to align that read or write to an even address. For example, if the SP contains the odd address 00 0083_{16}, a 32-bit read operation reads from addresses 00 0082_{16} and 00 0083_{16}.
- The SP will overflow if its value is increased beyond $FFFF_{16}$ or decreased below 0000_{16}. When the SP increases past $FFFF_{16}$, it counts forward from 0000_{16}. For example, if SP = $FFFE_{16}$ and an instruction adds 3 to the SP, the result is 0001_{16}. When the SP decreases past 0000_{16}, it counts backward from $FFFF_{16}$. For example, if SP = 0002_{16} and an instruction subtracts 4 from SP, the result is $FFFE_{16}$.
- When values are being saved to the stack, the SP is not forced to align with even or odd addresses. Alignment is forced by the memory wrapper or peripheral-interface logic.

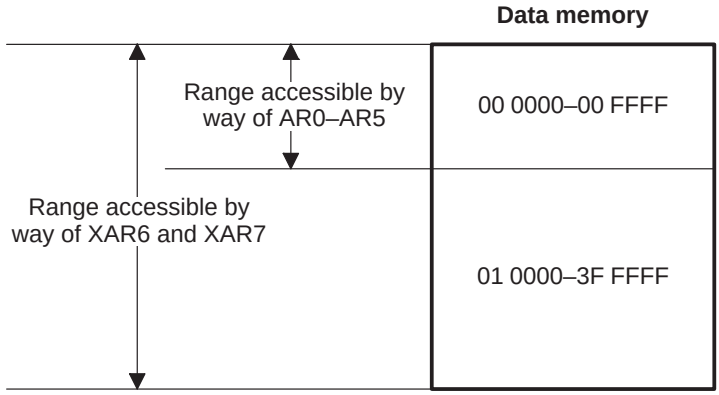
2.2.6 Auxiliary Registers (AR0–AR5, XAR6, XAR7)

The CPU provides eight auxiliary registers that can be used as pointers to memory (see section 5.5, *Indirect Addressing Modes That Use Auxiliary Registers*, on page 5-10). There are six 16-bit auxiliary registers—AR0, AR1, AR2, AR3, AR4, and AR5—and two 22-bit extended auxiliary registers—XAR6 and XAR7.

Figure 2–6 shows the ranges of data memory accessible to the 16-bit and 22-bit auxiliary registers. Because AR0–AR5 are 16-bits wide, their range is limited to addresses 00 0000_{16}–00 $FFFF_{16}$. When one of these six auxiliary registers is used, the content of the auxiliary register (or the content plus an offset) is concatenated with six leading zeros to form the full 22-bit address. XAR6 and XAR7, being 22 bits wide, can be used to access the full range of

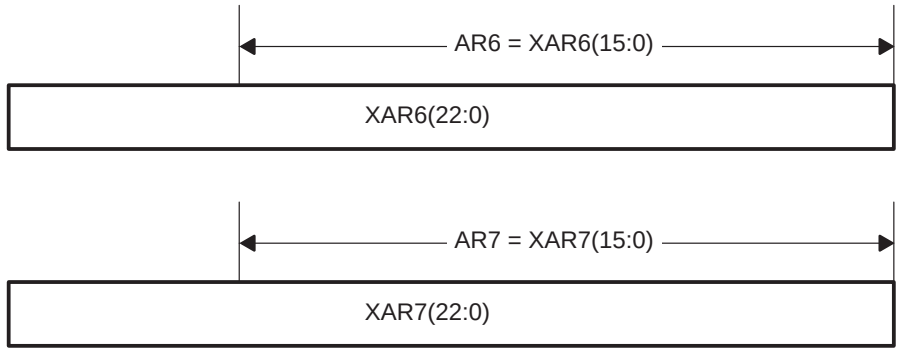
data memory. XAR7 can also be used by some instructions to point to any value in program memory; see section 5.5.3, *XAR7 Indirect Addressing Mode*.

Figure 2–6. Address Reach of the Auxiliary Registers



Many instructions allow you to access the 16 least significant bits (LSBs) of XAR6 and XAR7. As shown in Figure 2–7, the 16 LSBs of XAR6 are known as auxiliary register AR6, and the 16 LSBs of XAR7 are known as auxiliary register AR7. AR6 and AR7 are available as general-purpose registers, but they are not used as pointers.

Figure 2–7. AR6 and AR7



2.2.7 Program Counter (PC)

When the pipeline is full, the 22-bit program counter (PC) always points to the instruction that is currently being processed—the instruction that has just reached the decode 2 phase of the pipeline. Once an instruction reaches this phase of the pipeline, it cannot be flushed from the pipeline by an interrupt. It is executed before the interrupt is taken. The pipeline is discussed in Chapter 4.

2.2.8 Status Registers (ST0, ST1)

The ’27xx has two status registers, ST0 and ST1, which contain various flag bits and control bits. These registers can be stored into and loaded from data memory, enabling the status of the machine to be saved and restored for subroutines.

Figure 2–8 and Figure 2–9 show the organization of status registers ST0 and ST1, respectively. The status bits have been organized according to when the bit values are modified in the pipeline. Bits in ST0 are modified in the execute phase of the pipeline; bits in ST1 are modified in the decode 2 phase. (For details about the pipeline, see Chapter 4.) The status bits are described in detail in sections 2.3 (ST0) and 2.4 (ST1). Also, ST0 and ST1 are included in Appendix A, *Register Quick Reference*.

Figure 2–8. Status Register ST0

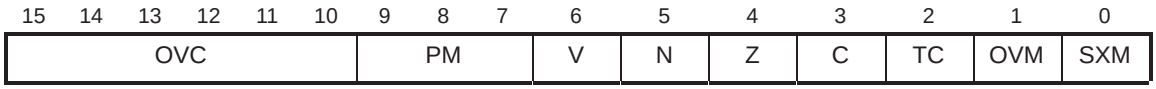
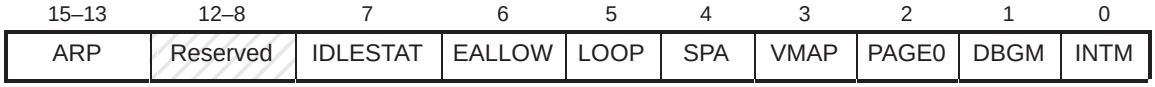


Figure 2–9. Status Register ST1



2.2.9 Interrupt-Control Registers (IFR, IER, DBGIER)

The ’27xx has three registers dedicated to the control of interrupts:

- ❏ Interrupt flag register (IFR)
- ❏ Interrupt enable register (IER)
- ❏ Debug interrupt enable register (DBGIER)

The IFR contains flag bits for maskable interrupts (those that can be enabled and disabled with software). When one of these flags is set, by hardware or software, the corresponding interrupt will be serviced if it is enabled. You enable or disable a maskable interrupt with its corresponding bit in the IER. The DBGIER indicates the time-critical interrupts that will be serviced (if enabled) while the DSP is in real-time emulation mode and the CPU is halted.

The ’27xx interrupts and the interrupt-control registers are described in detail in Chapter 3, *Interrupts*. Also, the IFR, IER, and DBGIER are included in Appendix A, *Register Quick Reference*.

2.3 Status Register ST0

The following figure shows the bit fields of status register ST0. All of these bit fields are modified in the execute phase of the pipeline. Detailed descriptions of these bits follow the figure.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OVC						PM			V	N	Z	C	TC	OVM	SXM
R/W–000000						R/W–000			R/W–0	R/W–0	R/W–0	R/W–0	R/W–0	R/W–0	R/W–0

Note: R = Read access; W = Write access; value following dash (–) is value after reset.

OVC
Bits15–10

Overflow counter. The overflow counter is a 6-bit signed counter with a range of –32 to 31. When overflow mode is off (OVM = 0), ACC overflows normally, and OVC keeps track of overflows. When overflow mode is on (OVM = 1) and an overflow occurs in ACC, the OVC is not affected. Instead, the CPU automatically fills ACC with a positive or negative saturation value (see the description for OVM on page 2-17).

When ACC overflows in the positive direction (from $7FFF\ FFFF_{16}$ to $8000\ 0000_{16}$), the OVC is incremented by 1. When ACC overflows in the negative direction (from $8000\ 0000_{16}$ to $7FFF\ FFFF_{16}$) the OVC is decremented by 1. The increment or decrement is performed as the overflow affects the V flag.

If OVC increments past its most positive value, 31, the counter wraps around to –32. If OVC decrements past its most negative value, –32, the counter wraps around to 31. At reset, OVC is cleared.

OVC is not affected by overflows in registers other than ACC and is not affected by compare instructions (CMP and CMPL). The table that follows explains how OVC may be affected by the saturate accumulator (SAT ACC) instruction.

Condition	Operation Performed by SAT ACC Instruction
OVC = 0	Leave ACC and OVC unchanged.
OVC > 0	Saturate ACC in the positive direction (fill ACC with 7FFF $FFFF_{16}$), and clear OVC.
OVC < 0	Saturate ACC in the negative direction (fill ACC with 8000 0000_{16}), and clear OVC.

PM
Bits 9–7

Product shift mode bits. This 3-bit value determines the shift mode for any output operation from the product (P) register. The shift modes are shown in the following table. The output can be to the ALU or to memory. All instructions that are affected by the product shift mode will sign extend the P register value during a right shift operation. At reset, PM is cleared (left shift by 1 bit is the default).

PM	Product Shift Mode
000	Left shift by 1. During the shift, the low-order bit is zero filled. At reset, this mode is selected.
001	No shift
010	Right shift by 1. During the shift, the lower bits are lost, and the shifted value is sign extended.
011	Right shift by 2. During the shift, the lower bits are lost, and the shifted value is sign extended.
100	Right shift by 3. During the shift, the lower bits are lost, and the shifted value is sign extended.
101	Right shift by 4. During the shift, the lower bits are lost, and the shifted value is sign extended.
110	Right shift by 5. During the shift, the lower bits are lost, and the shifted value is sign extended.
111	Right shift by 6. During the shift, the lower bits are lost, and the shifted value is sign extended.

Note: For performing unsigned arithmetic, you must use a product shift of 0 (PM = 001) to avoid sign extension and generation of incorrect results.

V
Bit 6

Overflow flag. If the result of an operation causes an overflow in the register holding the result, V is set and latched. If no overflow occurs, V is not modified. Once V is latched, it remains set until it is cleared by reset or by a conditional branch instruction that tests V. Such a conditional branch clears V regardless of whether the tested condition (V = 0 or V = 1) is true.

An overflow occurs in ACC (and V is set) if the result of an addition or subtraction does not fit within the signed numerical range -2^{31} to $(+2^{31} - 1)$, or $8000\ 0000_{16}$ to $7FFF\ FFFF_{16}$.

An overflow occurs in AH, AL, or another 16-bit register or data-memory location if the result of an addition or subtraction does not fit within the signed numerical range -2^{15} to $(+2^{15} - 1)$, or 8000_{16} to $7FFF_{16}$.

Compare instructions (CMP, CMPB and CMPL) do not affect the state of the V flag.

V can be summarized as follows:

V	Condition
0	V has been cleared.
1	An overflow has been detected, or V has been set.

N
Bit 5

Negative flag. During certain operations, N is set if the result of the operation is a negative number or cleared if the result is a positive number. At reset, N is cleared.

Results in ACC are tested for the negative condition. Bit 31 of ACC is the sign bit. If bit 31 is a 0, ACC is positive; if bit 31 is a1, ACC is negative. N is set if a result in ACC is negative or cleared if a result is positive.

Results in AH, AL, and other 16-bit registers or data-memory locations are also tested for the negative condition. In these cases bit 15 of the value is the sign bit (1 indicates negative, 0 indicates positive). N is set if the value is negative or cleared if the value is positive.

The TEST ACC instruction sets N if the value in ACC is negative. Otherwise the instruction clears N.

As shown in the following table, under overflow conditions, the way the N flag is set for compare operations is different from the way it is set for addition or subtraction operations. For addition or subtraction operations, the N flag is set to match the most significant bit of the truncated result. For compare operations, the N flag assumes infinite precision. This applies to operations whose result is loaded to ACC, AH, AL, another register, or a data-memory location.

A†	B†	(A – B)	Subtraction	Compare‡
Pos	Neg	Neg (due to overflow in positive direction)	N = 1	N = 0
Neg	Pos	Pos (due to overflow in negative direction)	N = 0	N = 1

† For 32-bit data: Pos = Positive number from 0000 0000_{16} to 7FFF $FFFF_{16}$
Neg = Negative number from 8000 0000_{16} to FFFF $FFFF_{16}$
For 16-bit data: Pos = Positive number from 0000_{16} to $7FFF_{16}$
Neg = Negative number from 8000_{16} to $FFFF_{16}$
‡ The compare instructions are CMP, CMPB, and CMPL.

N can be summarized as follows:

N	Condition
0	The tested number is positive, or N has been cleared.
1	The tested number is negative, or N has been set.

Z
Bit 4

Zero flag. Z is set if the result of certain operations is 0 or is cleared if the result is nonzero. This applies to results that are loaded into ACC, AH, AL, another register, or a data-memory location. At reset, Z is cleared.

The TEST ACC instruction sets Z if the value in ACC is 0. Otherwise, it clears Z.

Z can be summarized as follows:

Z	Condition
0	The tested number is nonzero, or Z has been cleared.
1	The tested number is 0, or Z has been set.

C
Bit 3

Carry bit. This bit indicates when an addition or increment generates a carry or when a subtraction, compare, or decrement generates a borrow. It is also affected by rotate operations on ACC and barrel shifts on ACC, AH, and AL.

During additions/increments, C is set if the addition generates a carry; otherwise C is cleared. There is one exception: If you are using the ADD instruction with a shift of 16, the ADD instruction can set C but cannot clear C.

During subtractions/decrements/compares, C is cleared if the subtraction generates a carry; otherwise C is set. There is one exception: if you are using the SUB instruction with a shift of 16, the SUB instruction can clear C but cannot set C.

This bit can be individually set and cleared by the SETC C instruction and CLRC C instruction, respectively. At reset, C is cleared.

C can be summarized as follows:

C	Condition
0	A subtraction generated a borrow, an addition did not generate a carry, or C has been cleared. *Exception:* An ADD instruction with a shift of 16 cannot clear C.
1	An addition generated a carry, a subtraction did not generate a borrow, or C has been set. *Exception:* A SUB instruction with a shift of 16 cannot set C.

TC
Bit 2

Test/control flag. This bit shows the result of a test performed by either the TBIT (test bit) instruction or the NORM (normalize) instruction.

The TBIT instruction tests a specified bit. When TBIT is executed, the TC bit is set if the tested bit is 1 or cleared if the tested bit is 0.

When a NORM instruction is executed, TC is modified as follows: If ACC holds 0, TC is set. If ACC does not hold 0, the CPU calculates the exclusive-OR of ACC bits 31 and 30, and then loads TC with the result.

This bit can be individually set and cleared by the SETC TC instruction and CLRC TC instruction, respectively. At reset, TC is cleared.

OVM
Bit 1

Overflow mode bit. When ACC accepts the result of an addition or subtraction and the result causes an overflow, OVM determines how the CPU handles the overflow:

OVM	How ACC Overflow is Handled
0	Results overflow normally in ACC. The OVC reflects the overflow (see the description for the OVC on page 2-13)
1	ACC is filled with either its most positive or most negative value as follows: If ACC overflows in the positive direction (from $7FFF\ FFFF_{16}$ to $8000\ 0000_{16}$), ACC is then filled with $7FFF\ FFFF_{16}$. If ACC overflows in the negative direction (from $8000\ 0000_{16}$ to $7FFF\ FFFF_{16}$), ACC is then filled with $8000\ 0000_{16}$.

This bit can be individually set and cleared by the SETC OVM instruction and CLRC OVM instruction, respectively. At reset, OVM is cleared.

SXM
Bit 0

Sign-extension mode bit. SXM affects the MOV, ADD, and SUB instructions that use a 16-bit value in an operation on the 32-bit accumulator. When the 16-bit value is loaded into (MOV), added to (ADD), or subtracted from (SUB) the accumulator, SXM determines whether the value is sign extended during the operation:

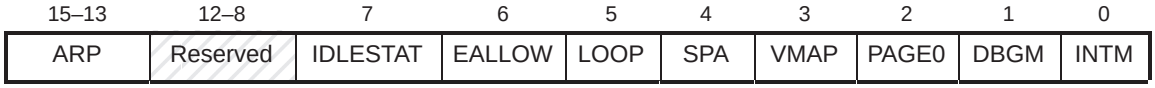
SXM	Sign Extension Mode
0	Sign extension is suppressed. (The value is treated as unsigned.)
1	Sign extension is enabled. (The value is treated as signed.)

SXM also determines whether the accumulator is sign extended when it is shifted right by the SFR instruction. SXM does not affect instructions that shift the product register value; all right shifts of the product register value use sign extension.

This bit can be individually set and cleared by the SETC SXM instruction and CLRC SXM instruction, respectively. At reset, SXM is cleared.

2.4 Status Register ST1

The following figure shows the bit fields of status register ST1. All of these bit fields are modified in the decode 2 phase of the pipeline. Detailed descriptions of these bits follow the figure.



Notes: 1) R = Read access; W = Write access; value following dash (–) is value after reset; reserved bits are always 0s and are not affected by writes.

2) The value of the VMAP bit after reset depends on the level of the VMAP input signal at reset. If the VMAP signal is low, the VMAP bit is 0 after reset; if the VMAP signal is high, the VMAP bit is 1 after reset.

ARP
Bits 15–13

Auxiliary register pointer. This 3-bit field points to the current auxiliary register. This could be one of the 16-bit auxiliary registers AR0 to AR5 or one of the 22-bit extended auxiliary registers (XAR6 or XAR7). AR6 and AR7 (the lower portions of XAR6 and XAR7, respectively) are available as general-purpose registers but are not used as pointers. The mapping of ARP values to auxiliary registers is as follows:

ARP	Selected Auxiliary Register
000	AR0 (selected at reset)
001	AR1
010	AR2
011	AR3
100	AR4
101	AR5
110	XAR6
111	XAR7

Reserved
Bits 12–8

These bits are reserved. Writes to these bits have no effect.

IDLESTAT
Bit 7

IDLE status bit. This ready-only bit is set when the IDLE instruction is executed. It is cleared by any one of the following events:

- ❏ An interrupt is serviced.
- ❏ An interrupt is not serviced but takes the core out of the IDLE state.
- ❏ A valid instruction enters the instruction register (the register that holds the instruction currently being decoded).
- ❏ A device reset occurs.

When the CPU services an interrupt, the current value of IDLESTAT is saved on the stack (when ST1 is saved on the stack), and then IDLESTAT is cleared. Upon return from the interrupt, IDLESTAT is not restored from the stack.

EALLOW
Bit 6

Emulation access enable bit. This bit, when set, enables access to emulation registers. EALLOW is set by the EALLOW instruction and cleared by the EDIS instruction. You can write to EALLOW by using the POP ST1 instruction or the POP DP:ST1 instruction.

When the CPU services an interrupt, the current value of EALLOW is saved on the stack (when ST1 is saved on the stack), and then EALLOW is cleared. Therefore, at the start of an interrupt service routine (ISR), access to emulation registers is disabled. If the ISR must access emulation registers, it must include an EALLOW instruction. At the end of the ISR, EALLOW can be restored by the IRET instruction.

LOOP
Bit 5

Loop instruction status bit. LOOP is set when a loop instruction (LOOPNZ or LOOPZ) reaches the decode 2 phase of the pipeline. The loop instruction does not end until a specified condition is met. When the condition is met, LOOP is cleared. LOOP is a read-only bit; it is not affected by any instruction except a loop instruction.

When the CPU services an interrupt, the current value of LOOP is saved on the stack (when ST1 is saved on the stack), and then LOOP is cleared. Upon return from the interrupt, LOOP is not restored from the stack.

SPA
Bits 4

Stack pointer alignment bit. SPA indicates whether the CPU has previously aligned the stack pointer to an even address by the ASP instruction:

SPA	Condition
0	The stack pointer has not been aligned to an even address.
1	The stack pointer has been aligned to an even address.

When the ASP (align stack pointer) instruction is executed, if the stack pointer (SP) points to an odd address, SP is incremented by 1 so that it points to an even address, and SPA is set. If SP already points to an even address, SP is not changed, but SPA is cleared. When the NASP (unalign stack pointer) instruction is executed, if SPA is 1, SP is decremented by 1 and SPA is cleared. If SPA is 0, SP is not changed.

At reset, SPA is cleared.

VMAP
Bit 3

Vector map bit. VMAP determines whether the interrupt vectors (including the reset vector) are mapped to the lowest or highest addresses in program memory:

VMAP	Condition
0	Interrupt vectors are mapped to the bottom of program memory, addresses 00 0000_{16}–00 $003F_{16}$.
1	Interrupt vectors are mapped to the top of program memory, addresses 3F $FFC0_{16}$–3F $FFFF_{16}$.

The core input signal VMAP, is sampled on reset. If the signal is high, the VMAP bit is set. If the signal is low, the VMAP bit is cleared. The output core signal VMAPS reflects the status of the VMAP bit.

This bit can be individually set and cleared by the SETC VMAP instruction and CLRC VMAP instruction, respectively.

PAGE0
Bit 2

PAGE0 addressing mode configuration bit. PAGE0 selects between two mutually-exclusive addressing modes: PAGE0 direct addressing mode and PAGE0 stack addressing mode. Selection of the modes is as follows:

PAGE0	Addressing Mode Enabled
0	PAGE0 stack addressing mode
1	PAGE0 direct addressing mode

This bit can be individually set and cleared by the SETC PAGE0 instruction and CLRC PAGE0 instruction, respectively. At reset, the PAGE0 bit is cleared (PAGE0 stack addressing mode is selected).

For details about the above addressing modes, see Chapter 5, *Addressing Modes*.

DBGM
Bit 1

Debug enable mask bit. When DBGM is set, the emulator cannot accesss memory or registers in real time. The debugger cannot update its windows.

In the real-time emulation mode, if DBGM = 1, the CPU ignores halt requests or hardware breakpoints until DBGM is cleared. DBGM does not prevent the CPU from halting at a software breakpoint. One effect of this may be seen in real-time emulation mode. If you single-step an instruction in real time emulation mode and that instruction sets DBGM, the CPU continues to execute instructions until DBGM is cleared.

When you give the TI debugger the REALTIME command (to enter real-time mode), DBGM is forced to 0. Having DBGM = 0 ensures that DT-DMAs are allowed; memory and register values can be passed to the host processor for updating debugger windows.

Before the CPU executes an interrupt service routine (ISR), it sets DBGM. When DBGM = 1, halt requests from the host processor and hardware breakpoints are ignored. If you want to single-step through or set breakpoints in a non-time-critical ISR, you must add a CLRC DBGM instruction at the beginning of the ISR.

DBGM is primarily used in emulation to block debug events in time-critical portions of program code. DBGM enables or disables debug events as follows:

DBGM	Condition
0	Debug events are enabled.
1	Debug events are disabled.

When the CPU services an interrupt, the current value of DBGM is saved on the stack (when ST1 is saved on the stack), and then DBGM is set. Upon return from the interrupt, DBGM is restored from the stack.

This bit can be individually set and cleared by the SETC DBGM instruction and CLRC DBGM instruction, respectively. DBGM is also set automatically during interrupt operations. At reset, DBGM is set. Executing the ABORTI (abort interrupt) instruction also sets DBGM.

INTM
Bit 0

Interrupt global mask bit. This bit globally enables or disables all maskable interrupts (those that can be blocked by software):

INTM	Condition
0	Maskable interrupts are globally enabled. In order to be acknowledged by the CPU, a maskable interrupt must also be locally enabled by the interrupt enable register (IER).
1	Maskable interrupts are globally disabled. Even if a maskable interrupt is locally enabled by the interrupt enable register (IER), it is not acknowledged by the CPU.

INTM has no effect on the nonmaskable interrupts, including a hardware reset or the hardware interrupt $\overline{\text{NMI}}$. In addition, when the CPU is halted in real-time emulation mode, an interrupt enabled by the IER and the DBGIER will be serviced even if INTM is set to disable maskable interrupts.

When the CPU services an interrupt, the current value of INTM is saved on the stack (when ST1 is saved on the stack), and then INTM is set. Upon return from the interrupt, INTM is restored from the stack.

This bit can be individually set and cleared by the SETC INTM instruction and CLRC INTM instruction, respectively. At reset, INTM is set. The value in INTM does not cause modification to the interrupt flag register (IFR), the interrupt enable register (IER), or the debug interrupt enable register (DBGIER).

2.5 Program Flow

The program control logic and program-address generation logic work together to provide proper program flow. Normally, the flow of a program is sequential: the CPU executes instructions at consecutive program-memory addresses. At times, a discontinuity is required; that is, a program must branch to a nonsequential address and then execute instructions sequentially at that new location. For this purpose, the ’27x supports interrupts, branches, calls, returns, and repeats.

Proper program flow also requires smooth flow at the instruction level. To meet this need, the ’27x has a protected pipeline and an instruction-fetch mechanism that attempts to keep the pipeline full.

2.5.1 Interrupts

Interrupts are hardware- or software-driven events that cause the CPU to suspend its current program sequence and execute a subroutine called an interrupt service routine. Interrupts are described in detail in Chapter 3 and tips for managing interrupts are given in section 8.3 on page 8-9.

2.5.2 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). Called subroutines or interrupt service routines are each concluded with a return instruction, which takes the return address from the stack or from XAR7 and places it into the program counter (PC).

The following branch instructions are conditional: B, SB, and BANZ. They are executed only if a certain specified or predefined condition is met. For detailed descriptions of these instructions, see Chapter 6, *Assembly Language Instructions*. For examples of these instructions, section 8.2, *Performing Special Branch Operations*.

2.5.3 Repeating a Single Instruction

The repeat (RPT) instruction allows the execution of a single instruction (N + 1) times, where N is specified as an operand of the RPT instruction. The instruction is executed once and then repeated N times. 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. For a description of RPT, see Chapter 6, *Assembly Language Instructions*.

2.5.4 Instruction Pipeline

Each instruction passes through eight independent phases that form an instruction pipeline. At any given time, up to eight instructions may be active, each in a different phase of completion. Not all reads and writes happen in the same phases, but a pipeline-protection mechanism stalls instructions as needed to ensure that reads and writes to the same location happen in the order in which they are programmed.

To maximize pipeline efficiency, an instruction-fetch mechanism attempts to keep the pipeline full. Its role is to fill an instruction-fetch queue, which holds instructions in preparation for decoding and execution. The instruction-fetch mechanism fetches 32-bits at a time from program memory; it fetches one 32-bit instruction or two 16-bit instructions.

The instruction-fetch mechanism uses three program-address counters: the program counter (PC), the instruction counter (IC), and the fetch counter (FC). When the pipeline is full the PC will always point to the instruction in its decode 2 pipeline phase. The IC points to the next instruction to be processed. When the PC points to a 1-word instruction, IC = (PC+1); when the PC points to a 2-word instruction, IC = (PC+2). The value in the FC is the address from which the next fetch is to be made.

The pipeline and the instruction-fetch mechanism are described in more detail in Chapter 4, *Pipeline*.

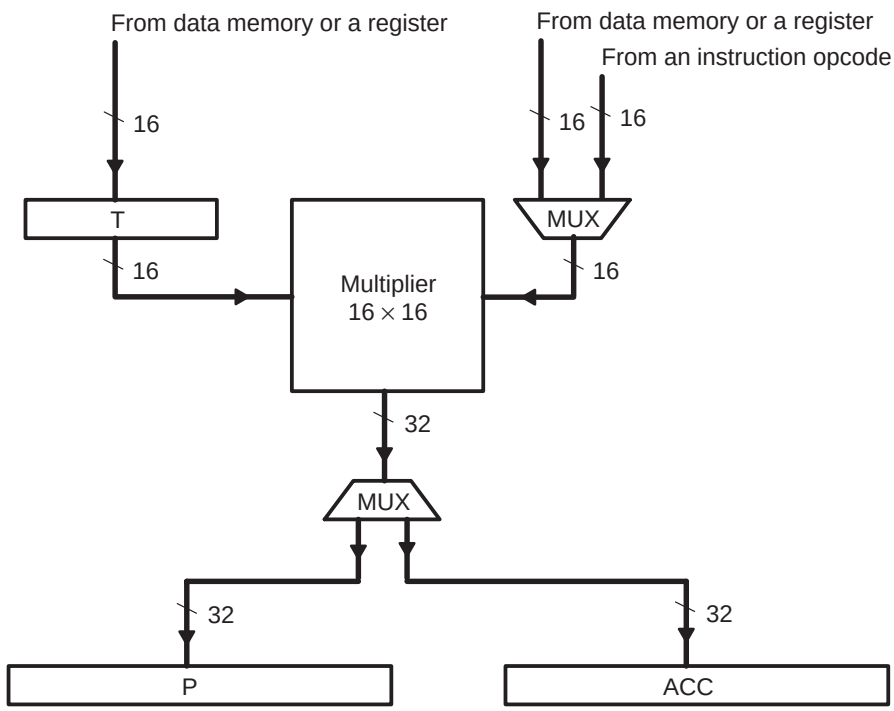
2.6 Multiply Operations

The ’27x has a 16-bit × 16-bit hardware multiplier that can produce a 32-bit signed or unsigned 32-bit product. Figure 2–10 shows the CPU components involved in multiplication. The multiplier accepts two 16-bit inputs:

- One input is from the 16-bit multiplicand register (T). Most ’27x multiply instructions require that you load T from a data-memory location or a register before you execute the instruction. However, the MAC instruction, one syntax of the MPY instruction, and one syntax of the MPYA instruction also load T before the multiplication.
- The other input is from one of the following:
  - A data-memory location or a register (depending on which you specify in the multiply instruction).
  - An instruction opcode. Some ’27x multiply instructions allow you to include a constant as an operand.

After the T register value has been multiplied by the second value, the 32-bit result is stored in one of two places, depending on the particular multiply instruction: the 32-bit product register (P) or the 32-bit accumulator (ACC).

Figure 2–10. Conceptual Diagram of Components Involved in Multiplication



2.7 Shift Operations

The shifter holds 32 bits and accepts either a 16-bit or 32-bit input value. When the input value has 16 bits, the value is loaded into the 16 least significant bits (LSBs) of the shifter. Depending on the instruction that uses the shifter, the output of the shifter may be all of its 32 bits or just its 16 LSBs.

When a value is shifted *right* by an amount N, the N LSBs of the value are lost and the bits to the left of the value are filled with all 0s or all 1s. If sign extension is specified, the bits to the left are filled with copies of the sign bit. If sign extension is not specified, the bits to the left are filled with 0s, or zero filled.

When a value is shifted *left* by an amount N, the bits to the right of the shifted value are zero filled. If the value has 16 bits and sign extension is specified, the bits to the left are filled with copies of the sign bit. If the value has 16 bits and sign extension is not specified, the bits to the left are zero filled. If the value has 32 bits, the N most significant bits (MSBs) of the value are lost, and sign extension is irrelevant.

Table 2–4 lists the instructions that use the shifter and provides an illustration of the corresponding shifter operation. The table uses the following graphical symbols:

This symbol represents the 32-bit shifter. The text inside the box indicates the direction of the shift.

This symbol indicates zero filling.

This symbol indicates sign extending.

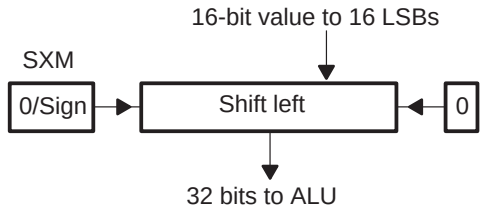
This symbol indicates that the MSBs of the shifter depend on the sign-extension mode bit (SXM). If SXM = 0, the MSBs are zero filled after the shift. If SXM = 1, the MSBs are filled with the sign of the shifted value.

This symbol indicates the carry bit (C).

For explanations of the instruction syntaxes listed in Table 2–4, see Chapter 6, *Assembly Language Instructions*.

Table 2–4. Shift Operations

Operation Type	Illustration

Left shift of 16-bit value for ACC operation. Syntaxes:
ADD ACC, *loc*<< *shift3*
ADD ACC, *#16BitSU* << *shift2*
SUB ACC, *loc*<< *shift3*
SUB ACC, *#16BitSU* << *shift2*
MOV ACC, *loc*<< *shift3*
MOV ACC, *#16BitSU* << *shift2*
	

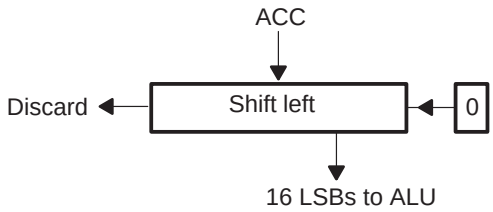
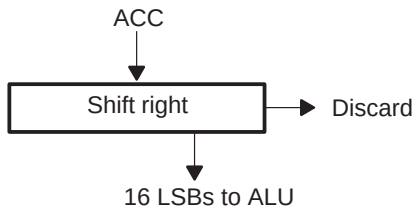
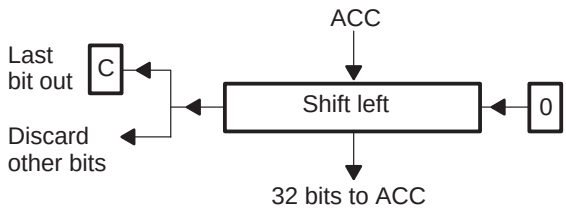
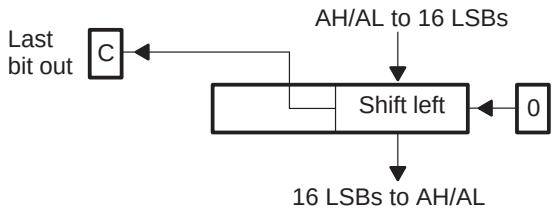
Store 16 LSBs of left-shifted ACC. Syntax: MOV *loc*, ACC<< *shift1*	
Store 16 MSBs of left-shifted ACC. Syntax: MOVH *loc*, ACC<<*shift1* Note: This instruction performs a single right shift by (16–*shift1*), where *shift1* is a value from 0 to 8.	
Logical left shift of ACC. The last bit to be shifted out fills the carry bit (C). Syntaxes: LSL ACC, *shift* LSL ACC, T Note: If T(3:0) = 0, indicating a shift of 0, C is cleared.	
Logical left shift of AH or AL. The last bit to be shifted out fills the carry bit (C). Syntaxes: LSL A*X*, *shift* LSL A*X*, T Note: If T(3:0) = 0, indicating a shift of 0, C is cleared.	

Table 2–4. Shift Operations (Continued)

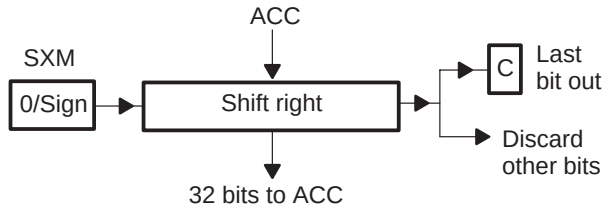
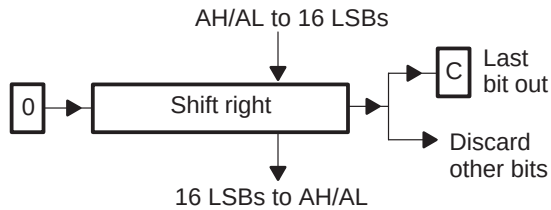
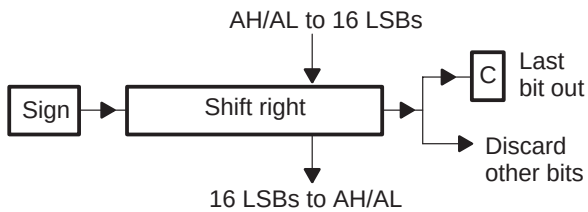
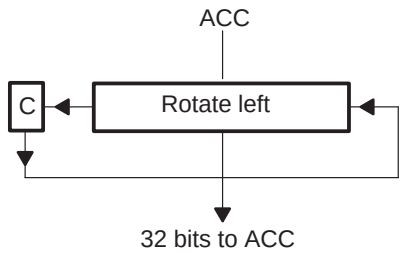
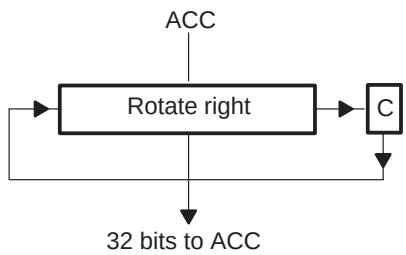
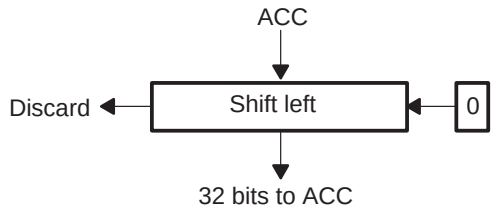
Operation Type	Illustration
Right shift of ACC. If SXM = 0, a logical shift is performed. If SXM = 1, an arithmetic shift is performed. The last bit to be shifted out fills the carry bit (C). Syntaxes: SFR ACC, *shift* SFR ACC, T Note: If T(3:0) = 0, indicating a shift of 0, C is cleared.	
Logical right shift of AH or AL. The last bit to be shifted out fills the carry bit (C). Syntaxes: LSR A*X*, *shift* LSR A*X*, T Note: If T(3:0) = 0, indicating a shift of 0, C is cleared.	
Arithmetic right shift of AH or AL. The last bit to be shifted out fills the carry bit (C). Syntaxes: ASR A*X*, *shift* ASR A*X*, T Note: If T(3:0) = 0, indicating a shift of 0, C is cleared.	
Rotate ACC left by 1 bit. Bit 31 of ACC fills the carry bit (C). C fills bit 0 of ACC. Syntax: ROL ACC	
Rotate ACC right by 1 bit. Bit 0 of ACC fills the carry bit (C). C fills bit 31 of ACC. Syntax: ROR ACC	

Table 2–4. Shift Operations (Continued)

Operation Type	Illustration
Conditional shift of ACC by 1 bit. Syntaxes: NORM ACC, *aux*++ NORM ACC, *aux*–– SUBCU ACC, *loc*	

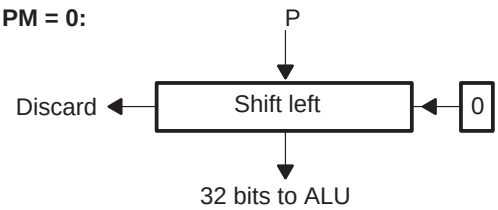
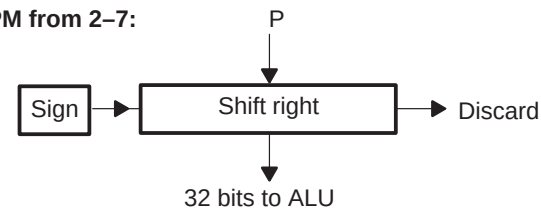
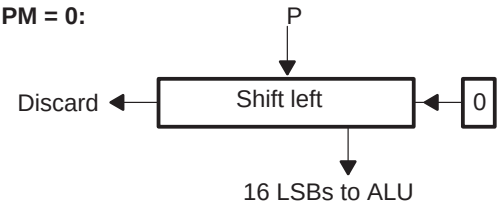
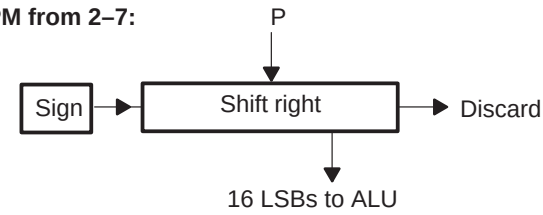
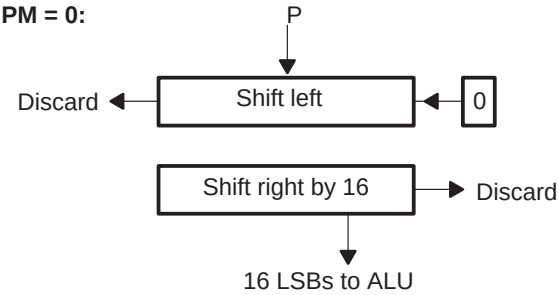
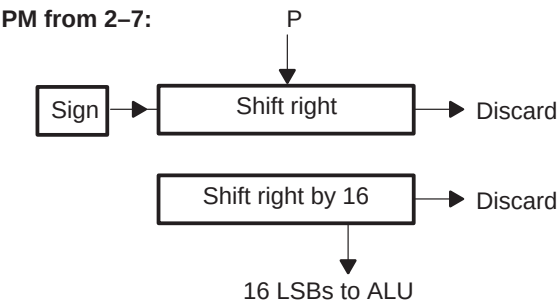
Shift of P as per PM bits. Syntaxes:
ADD ACC, P
SUB ACC, P
CMP ACC, P
MAC P, *loc*, 0:*pmem*
MOV ACC, P
MOVA T, *loc*
MOVP T, *loc*
MOVS T, *loc*
MPYA P, *loc*, *#16BitSigned*
MPYA P, T, *loc*
MPYS P, T, *loc*
	For PM = 0:
	 
	For PM = 1: No shift
	For PM from 2–7:
	 

Table 2–4. Shift Operations (Continued)

Operation Type	Illustration
Store 16 LSBs of shifted P. P is shifted as per the PM bits. The 16 LSBs of shifter are stored. Syntax: MOV *loc*, P	**For PM = 0:**  **For PM = 1:** No shift **For PM from 2–7:** 
Store 16 MSBs of shifted P. P is shifted as per the PM bits. The result is shifted right by 16 so that its 16 MSBs are in the 16 LSBs of the shifter. 16 LSBs of shifter are stored. Syntax: MOVH *loc*, P	**For PM = 0:**  **For PM = 1:** No shift **For PM from 2–7:** 

Chapter 3

Interrupts and Reset

This chapter describes the available interrupts and how they are handled by the CPU. It also explains how to control those interrupts that can be controlled through software. Finally, it describes how a hardware reset affects the CPU.

Topic **Page**

3.1	Interrupts Overview	3-2
3.2	Interrupt Vectors and Priorities	3-3
3.3	Maskable Interrupts: $\overline{\text{INT1}}$–$\overline{\text{INT14}}$, DLOGINT, and RTOSINT	3-5
3.4	Standard Operation for Maskable Interrupts	3-10
3.5	Nonmaskable Interrupts	3-16
3.6	Illegal-Instruction Trap	3-21
3.7	Hardware Reset ($\overline{\text{RS}}$)	3-22

3.1 Interrupts Overview

Interrupts are hardware- or software-driven signals that cause the ’27x to suspend its current program sequence and execute a subroutine. Typically, interrupts are generated by hardware devices that need to give data to or take data from the ’27x (for example, A/D and D/A converters and other processors). Interrupts can also signal that a particular event has taken place (for example, a timer has finished counting).

On the ’27x, interrupts can be triggered by software (the INTR, OR IFR, or TRAP instruction) or by hardware (a pin, a peripheral, or on-chip logic). If hardware interrupts are triggered at the same time, the ’27x services them according to a set priority ranking. Each of the ’27x interrupts, whether hardware or software, can be placed in one of the following two categories:

- **Maskable interrupts.**These are interrupts that can be blocked (masked) or enabled (unmasked) through software.
- **Nonmaskable interrupts.** These interrupts cannot be blocked. The ’27x will immediately approve this type of interrupt and branch to the corresponding subroutine. All software-initiated interrupts are in this category.

The ’27x handles interrupts in four main phases:

1) **Receive the interrupt request.** Suspension of the current program sequence must be requested by a software interrupt (from program code) or a hardware interrupt (from a pin or an on-chip device).
2) **Approve the interrupt.** The ’27x must approve the interrupt request. If the interrupt is maskable, certain conditions must be met in order for the ’27x to approve it. For nonmaskable hardware interrupts and for software interrupts, approval is immediate.
3) **Prepare for the interrupt service routine and save register values.** The main tasks performed in this phase are:
   - Complete execution of the current instruction and flush from the pipeline any instructions that have not reached the decode 2 phase.
   - Automatically save most of the current program context by saving the following registers to the stack: ST0, T, AL, AH, PL, PH, AR0, AR1, DP, ST1, DBGSTAT, and IER.
   - Fetch the interrupt vector and load it into the program counter (PC).
4) **Execute the interrupt service routine.** The ’27x branches to its corresponding subroutine called an interrupt service routine (ISR). The ’27x branches to the address (vector) you store at a predetermined vector location and executes the ISR you have written.

3.2 Interrupt Vectors and Priorities

The ’27x supports 32 interrupt vectors, including the reset vector. Each vector is a 22-bit address that is the start address for the corresponding interrupt service routine (ISR). Each vector is stored in 32 bits at two consecutive addresses. The location at the lower address holds the 16 least significant bits (LSBs) of the vector. The location at the higher address holds the 6 most significant bits (MSBs) right-justified. When an interrupt is approved, the 22-bit vector is fetched, and the 10 MSBs at the higher address are ignored.

Table 3–1 lists the available interrupt vectors and their locations. The addresses are shown in hexadecimal form. The table also shows the priority of each of the hardware interrupts.

Table 3–1. Interrupt Vectors and Priorities

	Absolute Address (hexadecimal)			
Vector	**VMAP = 0**	**VMAP = 1**	**Hardware Priority**	**Description**
RESET	00 0000	3F FFC0	1 (highest)	Reset
INT1	00 0002	3F FFC2	5	Maskable interrupt 1
INT2	00 0004	3F FFC4	6	Maskable interrupt 2
INT3	00 0006	3F FFC6	7	Maskable interrupt 3
INT4	00 0008	3F FFC8	8	Maskable interrupt 4
INT5	00 000A	3F FFCA	9	Maskable interrupt 5
INT6	00 000C	3F FFCC	10	Maskable interrupt 6
INT7	00 000E	3F FFCE	11	Maskable interrupt 7
INT8	00 0010	3F FFD0	12	Maskable interrupt 8
INT9	00 0012	3F FFD2	13	Maskable interrupt 9
INT10	00 0014	3F FFD4	14	Maskable interrupt 10
INT11	00 0016	3F FFD6	15	Maskable interrupt 11
INT12	00 0018	3F FFD8	16	Maskable interrupt 12
INT13	00 001A	3F FFDA	17	Maskable interrupt 13
INT14	00 001C	3F FFDC	18	Maskable interrupt 14
DLOGINT†	00 001E	3F FFDE	19 (lowest)	Maskable data log interrupt

† Interrupts DLOGINT and RTOSINT are generated by the emulation logic internal to the core.

Table 3–1. Interrupt Vectors and Priorities (Continued)

Vector	Absolute Address (hexadecimal) VMAP = 0	Absolute Address (hexadecimal) VMAP = 1	Hardware Priority	Description
RTOSINT†	00 0020	3F FFE0	4	Maskable real-time operating system interrupt
Reserved	00 0022	3F FFE2	2	Reserved
NMI	00 0024	3F FFE4	3	Nonmaskable interrupt
ILLEGAL	00 0026	3F FFE6	–	Illegal-instruction trap
USER1	00 0028	3F FFE8	–	User-defined software interrupt
USER2	00 002A	3F FFEA	–	User-defined software interrupt
USER3	00 002C	3F FFEC	–	User-defined software interrupt
USER4	00 002E	3F FFEE	–	User-defined software interrupt
USER5	00 0030	3F FFF0	–	User-defined software interrupt
USER6	00 0032	3F FFF2	–	User-defined software interrupt
USER7	00 0034	3F FFF4	–	User-defined software interrupt
USER8	00 0036	3F FFF6	–	User-defined software interrupt
USER9	00 0038	3F FFF8	–	User-defined software interrupt
USER10	00 003A	3F FFFA	–	User-defined software interrupt
USER11	00 003C	3F FFFC	–	User-defined software interrupt
USER12	00 003E	3F FFFE	–	User-defined software interrupt

† Interrupts DLOGINT and RTOSINT are generated by the emulation logic internal to the core.

The vector table can be mapped to the top or bottom of program space, depending on the value of the vector map bit (VMAP) in status register ST1. (ST1 is described in section 2.4 on page 2-19.) If the VMAP bit is 0, the vectors are mapped beginning at address $00\ 0000_{16}$. If the VMAP bit is 1, the vectors are mapped beginning at address $3F\ FFC0_{16}$. Table 3–1 lists the absolute addresses for VMAP = 0 and VMAP = 1.

The VMAP bit can be set by the SETC VMAP instruction and cleared by the CLRC VMAP instruction. The input core signal VMAP, sampled only at reset, determines the reset value of the VMAP bit. The state of the VMAP bit is also reflected at the output core signal VMAPS.

3.3 Maskable Interrupts: $\overline{INT1}$–$\overline{INT14}$, DLOGINT, and RTOSINT

$\overline{INT1}$–$\overline{INT14}$ are 14 general-purpose interrupts. DLOGINT (the data log interrupt) and RTOSINT (the real-time operating system interrupt) are available for emulation purposes. These interrupts are supported by three dedicated registers: the interrupt flag register (IFR), the interrupt enable register (IER), and the debug interrupt enable register (DBGIER).

The 16-bit IFR contains flag bits that indicate which of the corresponding interrupts are pending (waiting for approval from the CPU). The external input lines $\overline{INT1}$–$\overline{INT14}$ are sampled at every CPU clock cycle. If an interrupt signal is recognized, the corresponding bit in the IFR is set and latched. For DLOGINT or RTOSINT, a signal sent by the core's on-chip analysis logic causes the corresponding flag bit to be set and latched. You can set one or more of the IFR bits at the same time by using the OR IFR instruction. More details about the IFR are given in section 3.3.1. The on-chip analysis resources are introduced in Chapter 7.

The interrupt enable register (IER) and the debug interrupt enable register (DBGIER) each contain bits for individually enabling or disabling the maskable interrupts. To enable one of the interrupts in the IER, you set the corresponding bit in the IER; to enable the same interrupt in the DBGIER, you set the corresponding bit in the DBGIER. The DBGIER indicates which interrupts can be serviced when the core is in the real-time emulation mode. The IER and the DBGIER are discussed more in section 3.3.2. Real-time mode is discussed in section 7.4.2 on page 7-9.

The maskable interrupts also share bit 0 in status register ST1. This bit, the interrupt global mask bit (INTM), is used to globally enable or globally disable these interrupts. When INTM = 0, these interrupts are globally enabled. When INTM = 1, these interrupts are globally disabled. You can set and clear INTM with the SETC INTM and CLRC INTM instructions, respectively. ST1 is described in section 2.4 on page 2-19.

After a flag has been latched in the IFR, the corresponding interrupt is not serviced until it is appropriately enabled by two of the following: the IER, the DBGIER, and the INTM bit. As shown in Table 3–2, the requirements for enabling the maskable interrupts depend on the interrupt-handling process used. In the standard process, which occurs in most circumstances, the DBGIER is ignored. When the '27x is in real-time emulation mode and the CPU is halted, a different process is used. In this special case, the DBGIER is used and the INTM bit is ignored. (If the DSP is in real-time mode and the CPU is running, the standard interrupt-handling process applies.)

Once an interrupt has been requested and properly enabled, the CPU prepares for and then executes the corresponding interrupt service routine. For a detailed description of this process, see section 3.4.

Table 3–2. Requirements for Enabling a Maskable Interrupt

Interrupt-Handling Process	Interrupt Enabled If ...
Standard	INTM = 0 and bit in IER is 1
DSP in real-time mode and CPU halted	Bit in IER is 1 and bit in DBGIER is 1

As an example of varying interrupt-enable requirements, suppose you want interrupt $\overline{INT5}$ enabled. This corresponds to bit 4 in the IER and bit 4 in the DBGIER. Usually, $\overline{INT5}$ is enabled if INTM = 0 and IER(4) = 1. In real-time emulation mode with the CPU halted, $\overline{INT5}$ is enabled if IER(4) = 1 and DBGIER(4) = 1.

3.3.1 Interrupt Flag Register (IFR)

Figure 3–1 shows the IFR. If a maskable interrupt is pending (waiting for approval from the CPU), the corresponding IFR bit is 1; otherwise, the IFR bit is 0. To identify pending interrupts, use the PUSH IFR instruction and then test the value on the stack. Use the OR IFR instruction to set IFR bits, and use the AND IFR instruction to clear pending interrupts. When a hardware interrupt is serviced, or when an INTR instruction is executed, the corresponding IFR bit is cleared. All pending interrupts are cleared by the AND IFR, #0 instruction or by a hardware reset.

Notes:

When an interrupt is requested by the TRAP instruction, if the corresponding IFR bit is set, the CPU does not clear it automatically. If an application requires that the IFR bit be cleared, the bit must be cleared in the interrupt service routine.

Figure 3–1. Interrupt Flag Register (IFR)

15	14	13	12	11	10	9	8
RTOSINT	DLOGINT	INT14	INT13	INT12	INT11	INT10	INT9
R/W–0	R/W–0	R/W–0	R/W–0	R/W–0	R/W–0	R/W–0	R/W–0
7	6	5	4	3	2	1	0
INT8	INT7	INT6	INT5	INT4	INT3	INT2	INT1
R/W–0	R/W–0	R/W–0	R/W–0	R/W–0	R/W–0	R/W–0	R/W–0

Note: R = Read access; W = Write access; value following dash (–) is value after reset.

Bits 15 and 14 of the IFR correspond to the interrupts RTOSINT and DLOGINT:

RTOSINT	**Real-time operating system interrupt flag**	
Bit 15	RTOSINT = 0	RTOSINT is not pending.
	RTOSINT = 1	RTOSINT is pending.
DLOGINT	**Data log interrupt flag**	
Bit 14	DLOGINT = 0	DLOGINT is not pending.
	DLOGINT = 1	DLOGINT is pending.

For bits INT1–INT14, the following general description applies:

INTx	**Interrupt x flag (x = 1, 2, 3, ..., or 14)**	
Bit (x–1)	INTx = 0	$\overline{INTx}$ is not pending.
	INTx = 1	$\overline{INTx}$ is pending.

3.3.2 Interrupt Enable Register (IER) and Debug Interrupt Enable Register (DBGIER)

Figure 3–2 shows the IER. To enable an interrupt, set its corresponding bit to 1. To disable an interrupt, clear its corresponding bit to 0. Two syntaxes of the MOV instruction allow you to read from the IER and write to the IER. In addition, the OR IER instruction enables you to set IER bits, and the AND IER instruction enables you to clear IER bits. When a hardware interrupt is serviced, or when an INTR instruction is executed, the corresponding IER bit is cleared. At reset, all the IER bits are cleared to 0, disabling all the corresponding interrupts.

Note:

When an interrupt is requested by the TRAP instruction, if the corresponding IER bit is set, the CPU does not clear it automatically. If an application requires that the IER bit be cleared, the bit must be cleared in the interrupt service routine.

Figure 3–2. Interrupt Enable Register (IER)

15	14	13	12	11	10	9	8
RTOSINT	DLOGINT	INT14	INT13	INT12	INT11	INT10	INT9
R/W–0	R/W–0	R/W–0	R/W–0	R/W–0	R/W–0	R/W–0	R/W–0
7	6	5	4	3	2	1	0
INT8	INT7	INT6	INT5	INT4	INT3	INT2	INT1
R/W–0	R/W–0	R/W–0	R/W–0	R/W–0	R/W–0	R/W–0	R/W–0

Note: R = Read access; W = Write access; value following dash (–) is value after reset.

Note:

When using the AND IER and OR IER instructions, make sure that they do not modify the state of bit 15 (RTOSINT) unless a real-time operating system is present.

Bits 15 and 14 of the IER enable or disable the interrupts RTOSINT and DLOGINT:

RTOSINT	**Real-time operating system interrupt enable bit**	
Bit 15	RTOSINT = 0	RTOSINT is disabled.
	RTOSINT = 1	RTOSINT is enabled.
DLOGINT	**Data log interrupt enable bit**	
Bit 14	DLOGINT = 0	DLOGINT is disabled.
	DLOGINT = 1	DLOGINT is enabled.

For bits INT1–INT14, the following general description applies:

INTx	**Interrupt x enable bit (x = 1, 2, 3, ..., or 14)**	
Bit (x–1)	INTx = 0	$\overline{INTx}$ is disabled.
	INTx = 1	$\overline{INTx}$ is enabled.

Figure 3–3 shows the DBGIER, which is used only when the CPU is halted in real-time emulation mode. An interrupt enabled in the DBGIER is defined as a *time-critical interrupt*. When the CPU is halted in real-time mode, the only interrupts that are serviced are time-critical interrupts that are also enabled in the IER. If the CPU is running in real-time emulation mode, the standard interrupt-handling process is used and the DBGIER is ignored.

As with the IER, you can read the DBGIER to identify enabled or disabled interrupts and write to the DBGIER to enable or disable interrupts. To enable an interrupt, set its corresponding bit to 1. To disable an interrupt, set its corresponding bit to 0. Use the PUSH DBGIER instruction to read from the DBGIER and the POP DBGIER instruction to write to the DBGIER. At reset, all the DBGIER bits are set to 0.

Figure 3–3. Debug Interrupt Enable Register (DBGIER)

15	14	13	12	11	10	9	8
RTOSINT	DLOGINT	INT14	INT13	INT12	INT11	INT10	INT9
R/W–0	R/W–0	R/W–0	R/W–0	R/W–0	R/W–0	R/W–0	R/W–0
7	6	5	4	3	2	1	0
INT8	INT7	INT6	INT5	INT4	INT3	INT2	INT1
R/W–0	R/W–0	R/W–0	R/W–0	R/W–0	R/W–0	R/W–0	R/W–0

Note: R = Read access; W = Write access; value following dash (–) is value after reset.

Bits 15 and 14 of the DBGIER enable or disable the interrupts RTOSINT and DLOGINT:

RTOSINT	**Real-time operating system interrupt debug enable bit**	
Bit 15	RTOSINT = 0	RTOSINT is disabled.
	RTOSINT = 1	RTOSINT is enabled.
DLOGINT	**Data log interrupt debug enable bit**	
Bit 14	DLOGINT = 0	DLOGINT is disabled.
	DLOGINT = 1	DLOGINT is enabled.

For bits INT1–INT14, the following general description applies:

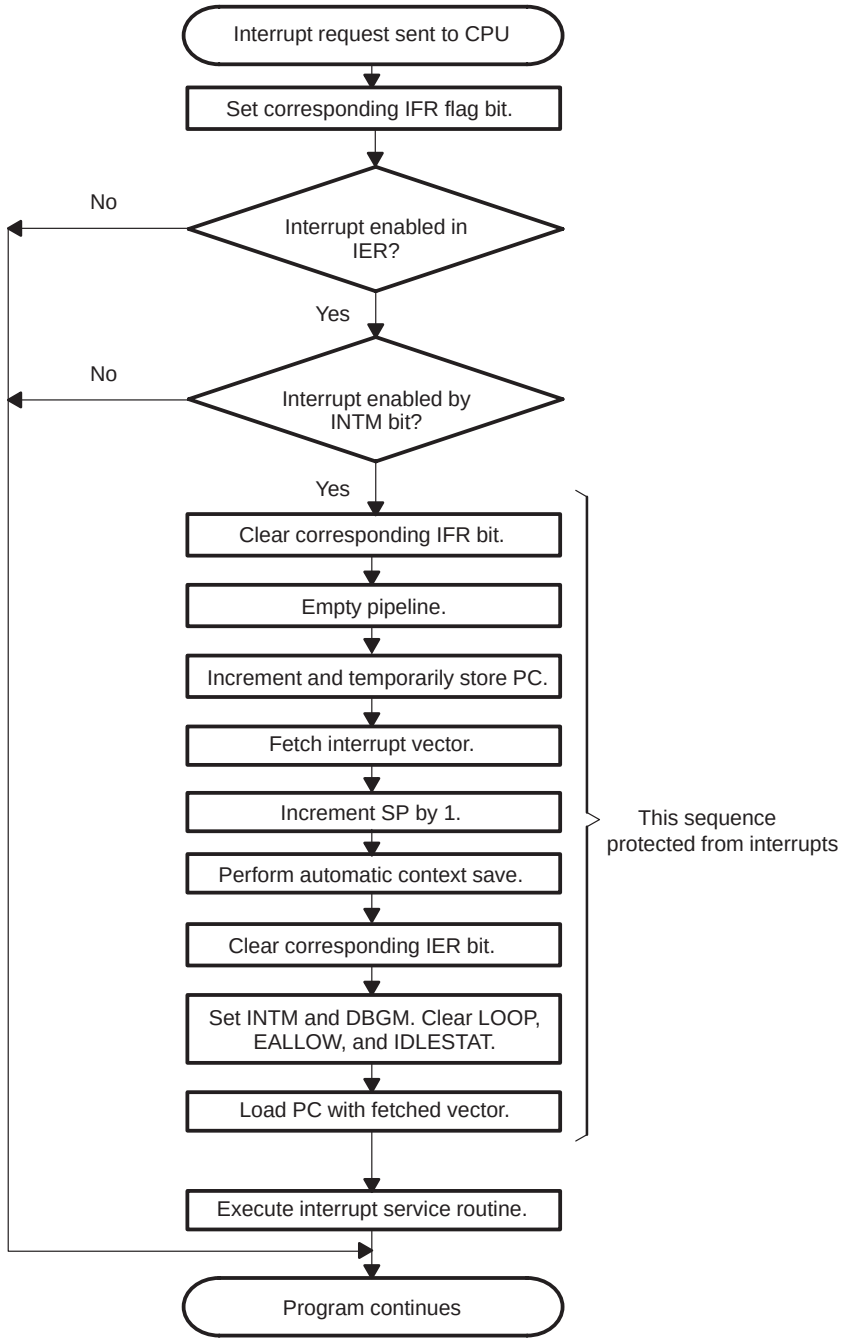
INTx	**Interrupt x debug enable bit (x = 1, 2, 3, ..., or 14)**	
Bit (x–1)	INTx = 0	$\overline{INTx}$ is disabled.
	INTx = 1	$\overline{INTx}$ is enabled.

3.4 Standard Operation for Maskable Interrupts

The flow chart in Figure 3–4 shows the standard process for handling interrupts. Section 7.4.2 on page 7-9 contains information on handling interrupts when the DSP is in real-time mode and the CPU is halted. When more than one interrupt is requested at the same time, the ’27x services them one after another according to their set priority ranking. See the priorities in Table 3–1 on page 3-3.

Figure 3–4 is not meant to be an exact representation of how an interrupt is handled. It is a conceptual model of the important events.

Figure 3–4. Standard Operation for Maskable Interrupts



What following list explains the steps shown in Figure 3–4:

1) **Interrupt request sent to CPU.** One of the following events occurs:
   - ❑ One of the pins $\overline{\text{INT1}}$–$\overline{\text{INT14}}$ is driven low.
   - ❑ The core emulation logic sends to the CPU a signal for DLOGINT or RTOSINT.
   - ❑ One of the interrupts $\overline{\text{INT1}}$–$\overline{\text{INT14}}$, DLOGINT, and RTOSINT is initiated by way of the OR IFR instruction.
2) **Set corresponding IFR flag bit.** When the CPU detects a valid interrupt in step 1, it sets and latches the corresponding flag in the interrupt flag register (IFR). This flag stays latched even if the interrupt is not approved by the CPU in step 3. The IFR is explained in detail in section 3.3.1.
3) **Interrupt enabled in IER? Interrupt enabled by INTM bit?** The CPU approves the interrupt only if the following conditions are true:
   - ❑ The corresponding bit in the IER is 1.
   - ❑ The INTM bit in ST1 is 0.

   Once an interrupt has been enabled and then approved by the CPU, no other interrupts can be serviced until the CPU has begun executing the interrupt service routine for the approved interrupt (step 13). The IER is described in section 3.3.2. ST1 is described in section 2.4 on page 2-19.
4) **Clear corresponding IFR bit.** Immediately after the interrupt is approved, its IFR bit is cleared. If the interrupt signal is kept low, the IFR register bit will be set again. However, the interrupt is not immediately serviced again. The CPU blocks new hardware interrupts until the interrupt service routine (ISR) begins. In addition, the IER bit is cleared (in step 10) before the ISR begins; therefore, an interrupt from the same source cannot disturb the ISR until the IER bit is set again by the ISR.
5) **Empty the pipeline.** The CPU completes any instructions that have reached or passed their decode 2 phase in the instruction pipeline. Any instructions that have not reached this phase are flushed from the pipeline.
6) **Increment and temporarily store PC.** The PC is incremented by 1 or 2, depending on the size of the current instruction. The result is the *return address*, which is temporarily saved in an internal hold register. During the automatic context save (step 9), the return address is pushed onto the stack.
7) **Fetch interrupt vector.** The PC is filled with the address of the appropriate interrupt vector, and the vector is fetched from that location. (To determine which vector address has been assigned to each of the interrupts, see section 3.2, *Interrupt Vectors*, on page 3-3.)

8) **Increment SP by 1.** The stack pointer (SP) is incremented by 1 in preparation for the automatic context save (step 9). During the automatic context save, the CPU performs 32-bit accesses, and the core expects 32-bit accesses to be aligned to even addresses by the memory wrapper. Incrementing SP by 1 ensures that the first 32-bit access does not overwrite the previous stack value. For the details about alignment of 32-bit accesses, see section 6.2 on page 6-31.

9) **Perform automatic context save.** A number of register values are saved automatically to the stack. These registers are saved in pairs; each pair is saved in a single 32-bit operation. At the end of each 32-bit save operation, the SP is incremented by 2. Table 3–3 shows the register pairs and the order in which they are saved. The core expects all 32-bit saves to be even-word aligned by the memory wrapper. As shown in the table, the SP is not affected by this alignment.

Table 3–3. Register Pairs Saved and SP Positions for Context Saves

Save Operation†	Register Pairs	Bit 0 of Storage Address	
		SP Starts at Odd Address	**SP Starts at Even Address**
		1 ← SP position before step 8	1
1st	ST0	0	0 ← SP position before step 8
	T	1	1
2nd	AL	0	0
	AH	1	1
3rd	PL‡	0	0
	PH	1	1
4th	AR0	0	0
	AR1	1	1
5th	ST1	0	0
	DP	1	1

Table 3–3. Register Pairs Saved and SP Positions for Context Saves (Continued)

Save Operation†	Register Pairs	Bit 0 of Storage Address	
		SP Starts at Odd Address	SP Starts at Even Address
6th	IER	0	0
	DBGSTAT§	1	1
7th	Return address (low half)	0	0
	Return address (high half)	1	1
		0 ← SP position after save	0
		1	1 ← SP position after save

† All registers are saved as pairs, as shown.
‡ The P register is saved with 0 shift (CPU ignores current state of the product shift mode bits, PM, in status register 0).
§ The DBGSTAT register contains special emulation information.

10) **Clear corresponding IER bit.** After the IER register is saved on the stack in step 9, the CPU clears the IER bit that corresponds to the interrupt being handled. This prevents reentry into the same interrupt. If you want to nest occurrences of the interrupt, have the ISR set that IER bit again.

11) **Set INTM and DBGM. Clear LOOP, EALLOW, and IDLESTAT.** All these bits are in status register ST1. By setting INTM to 1, the CPU prevents maskable interrupts from disturbing the ISR. If you wish to nest interrupts, have the ISR clear the INTM bit. By setting DBGM to 1, the CPU prevents debug events from disturbing time-critical code in the ISR. If you do not want debug events blocked, have the ISR clear DBGM.

    The CPU clears LOOP, EALLOW, and IDLESTAT so that the ISR operates within a new context.

12) **Load PC with fetched vector.** The PC is loaded with the interrupt vector that was fetched in step 7. The vector forces program control to the ISR.

13) **Execute interrupt service routine.** Here is where the CPU executes the program code you have prepared to handle the interrupt. A typical ISR is shown in Example 3–1.

    Although a number of register values are saved automatically in step 10, if the ISR uses other registers, you may need to save the contents of these registers at the beginning of the ISR. These values must then be restored before the return from the ISR. The ISR in Example 3–1 saves and restores auxiliary registers AR2–AR5, XAR6, and XAR7.

If you want the ISR to inform external hardware that the interrupt is being serviced, you can use the IACK instruction to send an interrupt acknowledge signal. The IACK instruction accepts a 16-bit constant as an operand and drives this 16-bit value on the 16 least significant lines of the data-write bus, DWDB(15:0). For a detailed description of the IACK instruction, see Chapter 6, *Assembly Language Instructions*.

14) **Program continues.** If the interrupt is not approved by the CPU, the interrupt is ignored, and the program continues uninterrupted. If the interrupt is approved, its interrupt service routine is executed and the program continues where it left off (at the return address).

Example 3–1. Typical ISR

```
INTX:  .long  INTXvector          ; Interrupt vector

INTXvector:                       ; Automatic context save already occurred.
       IACK   #INTX               ; Send out interrupt acknowledge (optional).

       PUSH   AR3:AR2             ; Save registers not saved during
       PUSH   AR5:AR4             ; automatic context save (optional).
       PUSH   XAR6                ;
       PUSH   XAR7                ;
       .                          ;
       .                          ;
       .                          ;

       ; Interrupt-specific code...
       .
       .
       .

       POP    XAR7                ; Restore registers saved at
       POP    XAR6                ; beginning of ISR.
       POP    AR5:AR4             ;
       POP    AR3:AR2             ;

       IRET                       ; Return and automatically restore
                                  ; register pairs
```

3.5 Nonmaskable Interrupts

Nonmaskable interrupts cannot be blocked by any of the enable bits (the INTM bit, the DBGM bit, and enable bits in the IFR, IER, or DBGIER). The ’27x immediately approves this type of interrupt and branches to the corresponding interrupt service routine. There is one exception to this rule: When the CPU is halted in stop mode (an emulation mode), no interrupts are serviced. Stop mode is described in section 7.4.1 on page 7-7.

The ’27x nonmaskable interrupts include:

- Software interrupts (the INTR and TRAP instructions).
- Hardware interrupt $\overline{\text{NMI}}$
- Illegal-instruction trap
- Hardware reset interrupt ($\overline{\text{RS}}$)

The software interrupt instructions and $\overline{\text{NMI}}$ are described in this section. The illegal-instruction trap and reset are described in sections 3.6 and 3.7, respectively.

3.5.1 INTR Instruction

You can use the INTR instruction to initiate one of the following interrupts by name: $\overline{\text{INT1}}$–$\overline{\text{INT14}}$, DLOGINT, RTOSINT and $\overline{\text{NMI}}$. For example, you can execute the interrupt service routine for $\overline{\text{INT1}}$ by using the following instruction:

```
INTR  INT1
```

Once an interrupt is initiated by the INTR instruction, how it is handled depends on which interrupt is specified:

- **$\overline{\text{INT1}}$–$\overline{\text{INT14}}$, DLOGINT, and RTOSINT.** These maskable interrupts have corresponding flag bits in the IFR. When a request for one of these interrupts is received at an external pin, the corresponding IFR bit is set and the interrupt must be enabled to be serviced. In contrast, when one of these interrupts is initiated by the INTR instruction, the IFR flag is not set, and the interrupt is serviced regardless of the value of any enable bits. However, in other respects, the INTR instruction and the hardware request are the same. For example, both clear the IFR bit that corresponds to the requested interrupt. For more details, see section 3.4 on page 3-10.

- **$\overline{\text{NMI}}$.** Because this interrupt is nonmaskable, a hardware request at a pin and a software request with the INTR instruction lead to the same events. These events are identical to those that take place during a TRAP instruction (see section 3.5.2).

Chapter 6, *Assembly Language Instructions*, contains a detailed description of the INTR instruction.

3.5.2 TRAP Instruction

You can use the TRAP instruction to initiate any interrupt, including one of the user-defined software interrupts (see USER1–USER12 in Table 3–1 on page 3-3). The TRAP instruction refers to one of the 32 interrupts by a number from 0 to 31. For example, you can execute the interrupt service routine for $\overline{\text{INT1}}$ by using the following instruction:

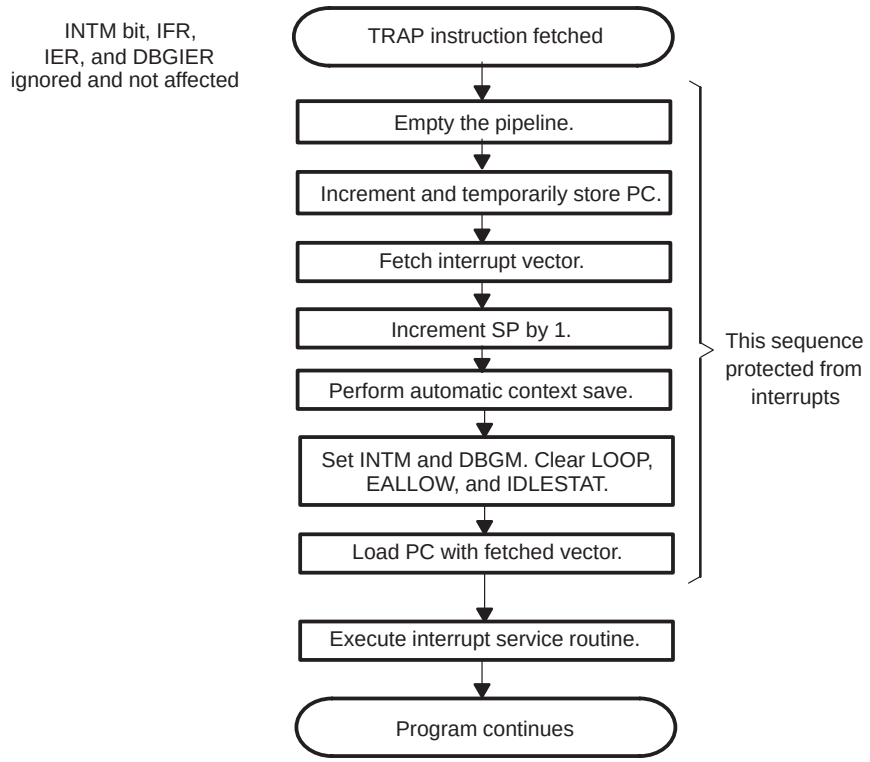
```
TRAP    #1
```

Regardless of whether the interrupt has bits set in the IFR and IER, neither the IFR nor the IER is affected by this instruction. Figure 3–5 shows a functional flow chart for an interrupt initiated by the TRAP instruction. For more details about the TRAP instruction, see Chapter 6, *Assembly Language Instructions*.

Note:

The TRAP #0 instruction does not initiate a full reset. It only forces execution of the interrupt service routine that corresponds to the RESET interrupt vector.

Figure 3–5. Functional Flow Chart for an Interrupt Initiated by the TRAP Instruction



The following lists explains the steps shown in Figure 3–5:

1) **TRAP instruction fetched.** The CPU fetches the TRAP instruction from program memory. The desired interrupt vector has been specified as an operand and is now encoded in the instruction word. At this stage, no other interrupts can be serviced until the CPU begins executing the interrupt service routine (step 9).

2) **Empty the pipeline.** The CPU completes any instructions that have reached or passed the decode 2 phase of the pipeline. Any instructions that have not reached this phase are flushed from the pipeline.

3) **Increment and temporarily store PC.** The PC is incremented by 1. This value is the *return address*, which is temporarily saved in an internal hold register. During the automatic context save (step 6), the return address is pushed onto the stack.

4) **Fetch interrupt vector.** The PC is set to point to the appropriate vector location (based on the VMAP bit and the interrupt), and the vector located at the PC address is loaded into the PC. (To determine which vector address has been assigned to each of the interrupts, see section 3.2, *Interrupt Vectors*, on page 3-3.)

5) **Increment SP by 1.** The stack pointer (SP) is incremented by 1 in preparation for the automatic context save (step 6). During the automatic context save, the CPU performs 32-bit accesses, which are aligned to even addresses. Incrementing SP by 1 ensures that the first 32-bit access will not overwrite the previous stack value. For the details about alignment of 32-bit accesses, see section 6.2 on page 6-31.

6) **Perform automatic context save.** A number of register values are saved automatically to the stack. These registers are saved in pairs; each pair is saved in a single 32-bit operation. At the end of each 32-bit operation, the SP is incremented by 2. Table 3–3 shows the register pairs and the order in which they are saved. All 32-bit saves are even-word aligned. As shown in the table, the SP is not affected by this alignment.

Table 3–4. Register Pairs Saved and SP Positions for Context Saves

Save Operation†	Register Pairs	Bit 0 of Storage Address	
		SP Starts at Odd Address	**SP Starts at Even Address**
		1 ← SP position before step 5	1
1st	ST0	0	0 ← SP position before step 5
	T	1	1
2nd	AL	0	0
	AH	1	1
3rd	PL‡	0	0
	PH	1	1
4th	AR0	0	0
	AR1	1	1
5th	ST1	0	0
	DP	1	1
6th	IER	0	0
	DBGSTAT§	1	1
7th	Return address (low half)	0	0
	Return address (high half)	1	1
		0 ← SP position after save	0
		1	1 ← SP position after save

† All registers are saved as pairs, as shown.
‡ The P register is saved with 0 shift (CPU ignores current state of the product shift mode bits, PM, in status register 0).
§ The DBGSTAT register contains special emulation information.

7) **Set INTM and DBGM. Clear LOOP, EALLOW, and IDLESTAT.** All these bits are in status register ST1 (described in section 2.4 on page 2-19). By setting INTM to 1, the CPU prevents maskable interrupts from disturbing the ISR. If you wish to nest interrupts, have the ISR clear the INTM bit. By setting DBGM to 1, the CPU prevents debug events from disturbing time-critical code in the ISR. If you do not want debug events blocked, have the ISR clear DBGM.

The CPU clears LOOP, EALLOW, and IDLESTAT so that the ISR operates within a new context.

8) **Load PC with fetched vector.** The PC is loaded with the interrupt vector that was fetched in step 4. The vector forces program control to the ISR.

9) **Execute interrupt service routine.** The CPU executes the program code you have prepared to handle the interrupt. You may wish to have the interrupt service routine (ISR) save register values in addition to those saved in step 6. A typical ISR is shown in Example 3–1 on page 3-15.

   If you want the ISR to inform external hardware that the interrupt is being serviced, you can use the IACK instruction to send an interrupt acknowledge signal. The IACK instruction accepts a 16-bit constant as an operand and drives this 16-bit value on the 16 least significant lines of the data-write bus, DWDB(15:0). For a detailed description of the IACK instruction, see Chapter 6, *Assembly Language Instructions*.

10) **Program continues.** After the interrupt service routine is completed, the program continues where it left off (at the return address).

3.5.3 Hardware Interrupt $\overline{\text{NMI}}$

An interrupt can be requested by way the $\overline{\text{NMI}}$ input pin, which must be driven low to initiate the interrupt. Although $\overline{\text{NMI}}$ cannot be masked, there are some debug execution states in which $\overline{\text{NMI}}$ is not serviced (see section 7.4, *Execution Control Modes*, on page 7-7). For more details on real-time mode, see section 7.4.2 on page 7-9. Once a valid request is detected on the $\overline{\text{NMI}}$ pin, the CPU handles the interrupt in the same manner as shown for the TRAP instruction (see section 3.5.2).

3.6 Illegal-Instruction Trap

Any one of the following three events causes an illegal-instruction trap:

- An invalid instruction is decoded (this includes invalid addressing modes).
- The opcode value 0000_{16} is decoded. This opcode corresponds to the ITRAP0 instruction.
- The opcode value $FFFF_{16}$ is decoded. This opcode corresponds to the ITRAP1 instruction.

An illegal-instruction trap cannot be blocked, not even during emulation. Once initiated, an illegal-instruction trap operates the same as a TRAP #19 instruction. The handling of an interrupt initiated by the TRAP instruction is described in section 3.5.2. As part of its operation, the illegal-instruction trap saves the return address on the stack. Thus, you can detect the offending address by examining this saved value. For more information about the TRAP instruction, see Chapter 6, *Assembly Language Instructions*.

3.7 Hardware Reset ($\overline{RS}$)

When asserted, the reset input signal ($\overline{RS}$) places the core into a known state. As part of a hardware reset, all current operations are aborted, the pipeline is flushed, and the CPU registers are reset as shown in Table 3–5. Then the RESET interrupt vector is fetched and the corresponding interrupt service routine is executed. For the reset condition of signals, see the data sheet for your particular ’27x DSP. Also see the your data sheet for specific information on the process for resetting your DSP. Although $\overline{RS}$ cannot be masked, there are some debug execution states in which $\overline{RS}$ is not serviced (see section 7.4, *Execution Control Modes*, on page 7-7).

Table 3–5. Registers After Reset

Register	Bit(s)	Value After Reset	Comments
ACC	all	$0000\ 0000_{16}$	
AR0	all	0000_{16}	
AR1	all	0000_{16}	
AR2	all	0000_{16}	
AR3	all	0000_{16}	
AR4	all	0000_{16}	
AR5	all	0000_{16}	
XAR6	all	$00\ 0000_{16}$	
XAR7	all	$00\ 0000_{16}$	
DP	all	0000_{16}	DP points to data page 0.
IFR	16 bits	0000_{16}	There are no pending interrupts. All interrupts pending at the time of reset have been cleared.
IER	16 bits	0000_{16}	Maskable interrupts are disabled in the IER.
DBGIER	all	0000_{16}	Maskable interrupts are disabled in the DBGIER.

Note: The registers listed in this table are introduced in section 2.2, *CPU Registers*, on page 2-4.

Table 3–5. Registers After Reset (Continued)

Register	Bit(s)	Value After Reset	Comments
P	all	$0000\ 0000_{16}$	
PC	all	If VMAP bit = 0 PC = [$00\ 0000_{16}$] If VMAP bit = 1 PC = [$3F\ FFC0_{16}$]	PC is loaded with the reset interrupt vector at program-space address $00\ 0000_{16}$ or $3F\ FFC0_{16}$.
SP	all	0000_{16}	SP points to address 0000_{16}.
ST0	0: SXM	0	Sign extension is suppressed.
	1: OVM	0	Overflow mode is off.
	2: TC	0	
	3: C	0	
	4: Z	0	
	5: N	0	
	6: V	0	
	7–9: PM	000_2	The product shift mode is set to left-shift-by-1.
	10–15: OVC	$00\ 0000_2$	
ST1‡	0: INTM	1	Maskable interrupts are globally disabled. They cannot be serviced unless the ’27x is in real-time mode with the CPU halted.
	1: DBGM	1	Emulation accesses and events are disabled.

Note: The registers listed in this table are introduced in section 2.2, *CPU Registers*, on page 2-4.

Table 3–5. Registers After Reset (Continued)

Register	Bit(s)	Value After Reset	Comments
	2: PAGE0	0	PAGE0 stack addressing mode is enabled. PAGE0 direct addressing mode is disabled.
	3: VMAP	If the VMAP signal was low at reset, the VMAP bit is 0. If the VMAP signal was high at reset, the VMAP bit is 1.	If the VMAP bit is 0, the interrupt vectors are mapped to program-memory addresses 00 0000_{16}–00 $003F_{16}$. If the VMAP bit is 1, the interrupt vectors are mapped to program-memory addresses 3F $FFC0_{16}$–3F $FFFF_{16}$.
	4: SPA	0	
	5: LOOP	0	
	6: EALLOW	0	Access to emulation registers is disabled.
	7: IDLESTAT	0	
	8–12: Reserved	00000_2	
	13–15: ARP	000_2	ARP points to AR0.
T	all	0000_{16}	

Note: The registers listed in this table are introduced in section 2.2, *CPU Registers*, on page 2-4.

Chapter 4

Pipeline

This chapter explains the operation of the ’27x instruction pipeline. The pipeline contains hardware that prevents reads and writes at the same register or data-memory location from happening out of order. However, you can increase the efficiency of your programs if you take into account the operation of the pipeline. In addition, you should be aware of two types of pipeline conflicts the pipeline does not protect against and how you can avoid them (see section 4.5).

For more information about the instructions shown in examples throughout this chapter, see Chapter 6, *Assembly Language Instructions*.

Topic **Page**

4.1 Pipelining of Instructions 4-2
4.2 Visualizing Pipeline Activity 4-7
4.3 Freezes in Pipeline Activity 4-10
4.4 Pipeline Protection 4-12
4.5 Avoiding Unprotected Operations 4-16

4.1 Pipelining of Instructions

When executing a program, the ’27x CPU performs these basic operations:

- ❏ Fetches instructions from program memory
- ❏ Decodes instructions
- ❏ Reads data values from memory or from CPU registers
- ❏ Executes instructions
- ❏ Writes results to memory or to CPU registers

For efficiency, the ’27x performs these operations in eight independent phases. Reads from memory are designed to be pipelined in two stages, which correspond to the two pipeline phases used by the CPU for each memory-read operation. At any time, there can be up to eight instructions being carried out, each in a different phase of completion. Following are descriptions of the eight phases in the order they occur. The address and data buses mentioned in these descriptions are introduced in section 1.4.1 on page 1-11.

Fetch 1 (F1) In the fetch 1 (F1) phase, the CPU drives a program-memory address on the 22-bit program address bus, PAB(21:0).

Fetch 2 (F2) In the fetch 2 (F2) phase, the CPU reads from program memory by way of the program-read data bus, PRDB (31:0), and loads the instruction(s) into an instruction-fetch queue.

Decode 1 (D1) The ’27x supports both 32-bit and 16-bit instructions and an instruction can be aligned to an even or odd address. The decode 1 (D1) hardware identifies instruction boundaries in the instruction-fetch queue and determines the size of the next instruction to be executed. It also determines whether the instruction is a legal instruction.

Decode 2 (D2) The decode 2 (D2) hardware requests an instruction from the instruction-fetch queue. The requested instruction is loaded into the instruction register, where decoding is completed. Once an instruction reaches the D2 phase, it runs to completion. In this pipeline phase, the following tasks are performed:

- ❑ If data is to be read from memory, the CPU generates the source address or addresses.
- ❑ If data is to be written to memory, the CPU generates the destination address.
- ❑ The address register arithmetic unit (ARAU) performs any required modifications to the stack pointer (SP) or to an auxiliary register and/or the auxiliary register pointer (ARP).
- ❑ If a program-flow discontinuity (such as a branch or an illegal-instruction trap) is required, it is taken.

Read 1 (R1) If data is to be read from memory, the read 1 (R1) hardware drives the address(es) on the appropriate address bus(es).

Read 2 (R2) If data was addressed in the R1 phase, the read 2 (R2) hardware fetches that data by way of the appropriate data bus(es).

Execute (E) In the execute (E) phase, the CPU performs all multiplier, shifter, and ALU operations. This includes all the prime arithmetic and logic operations involving the accumulator and product register. For operations that involve reading a value, modifying it, and writing it back to the original location, the modification (typically an arithmetic or a logical operation) is performed during the E phase of the pipeline. Any CPU register values used by the multiplier, shifter, and ALU are read from the registers at the beginning of the E phase. A result that is to be written to a CPU register is written to the register at the end of the E phase.

Write (W) If a transferred value or result is to be written to memory, the write occurs in the write (W) phase. The CPU drives the destination address, the appropriate write strobes, and the data to be written. The actual storing, which takes at least one more clock cycle, is handled by memory wrappers or peripheral interface logic and is not visible as a part of the CPU pipeline.

Although every instruction passes through the eight phases, not every phase is active for a given instruction. Some instructions complete their operations in the decode 2 phase, others in the execute phase, and still others in the write phase. For example, instructions that do not read from memory perform no operations in the read phases, and instructions that do not write to memory perform no operation in the write phase.

Because different instructions perform modifications to memory and registers during different phases of their completion, an unprotected pipeline could lead to reads and writes at the same location happening out of the intended order. The CPU automatically adds inactive cycles to ensure that these reads and writes happen as intended. For more details about pipeline protection, see section 4.4 on page 4-12.

4.1.1 Decoupled Pipeline Segments

The fetch 1 through decode 1 (F1–D1) hardware acts independently of the decode 2 through write (D2–W) hardware. This allows the CPU to continue fetching instructions when the D2–W phases are halted. It also allows fetched instructions to continue through their D2–W phases when fetching of new instructions is delayed. Events that cause portions of the pipeline to halt are described in section 4.3.

Instructions in their fetch 1, fetch 2, and decode 1 phases are discarded if an interrupt or other program-flow discontinuity occurs. An instruction that reaches its decode 2 phase always runs to completion before any program-flow discontinuity is taken.

4.1.2 Instruction-Fetch Mechanism

The instruction-fetch mechanism is the hardware for the F1 and F2 pipeline phases. During the F1 phase, the mechanism drives an address on the program address bus (PAB). During the F2 phase, it reads from the program-read data bus (PRDB). While an instruction is read from program memory in the F2 phase, the address for the next fetch is placed on the program address bus (during the next F1 phase).

The instruction-fetch mechanism contains an instruction-fetch queue of four 32-bit registers. During the F2 phase, the fetched instruction is added to the queue, which behaves like a first-in, first-out (FIFO) buffer. The first instruction in the queue is the first to be executed. The instruction-fetch mechanism performs 32-bit fetches until the queue is full. When a program-flow discontinuity (such as a branch or an interrupt) occurs, the queue is emptied. When the instruction at the bottom of the queue reaches its D2 phase, that instruction is passed to the instruction register for further decoding.

4.1.3 Address Counters FC, IC, and PC

Three program-address counters are involved in the fetching and execution of instructions:

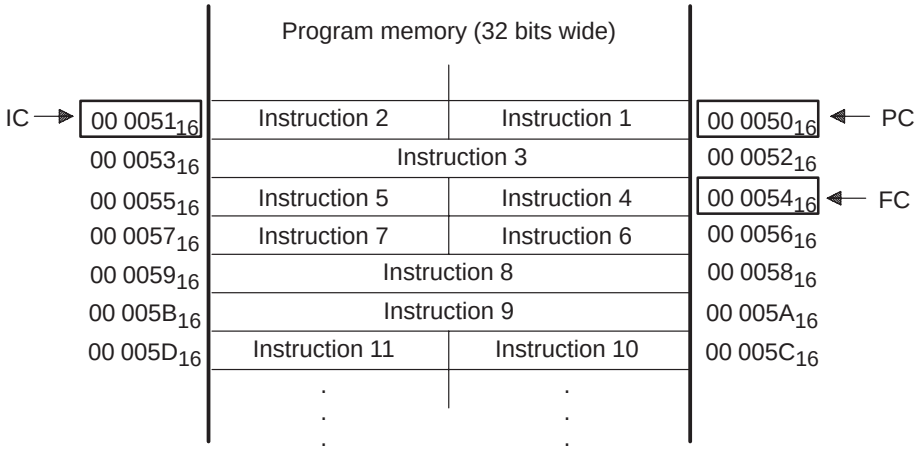
- **Fetch counter (FC).** The fetch counter contains the address that is driven on the program address bus (PAB) in the F1 pipeline phase. The CPU continually increments the FC until the queue is full or the queue is emptied by a program-flow discontinuity. Generally, the FC holds an even address and is incremented by 2, to accommodate 32-bit fetches. The only exception to this is when the code after a discontinuity begins at an odd address. In this case, the FC holds the odd address. After performing a16-bit fetch at the odd address, the CPU increments the FC by 1 and resumes 32-bit fetching at even addresses.

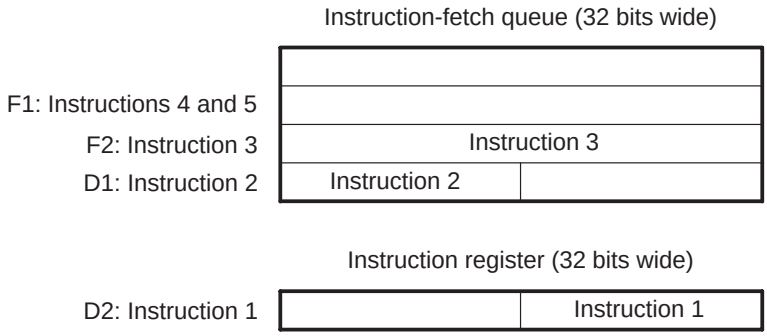
- **Instruction counter (IC).** After the D1 hardware determines the instruction size (16-bit or 32-bit), it fills the instruction counter (IC) with the address of the next instruction to undergo D2 decoding. On an interrupt or call operation, the IC value represents the return address, which is saved to the stack or to auxiliary register XAR7.

- **Program counter (PC).** When a new address is loaded into the IC, the previous IC value is loaded into the PC. The program counter (PC) always contains the address of the instruction that has reached its D2 phase.

Example 4–1 shows the relationship between the pipeline and the address counters. Instruction 1 has reached its D2 phase (it has been passed to the instruction register). The PC points to the address from which instruction 1 was taken (00 0050_{16}). Instruction 2 has reached its D1 phase and will be executed next (assuming no program-flow discontinuity flushes the instruction-fetch queue). The IC points to the address from which instruction 2 was taken (00 0051_{16}). Instruction 3 is in its F2 phase. It has been transferred to the instruction-fetch queue but has not been decoded. Instructions 4 and 5 are each in their F1 phase. The FC address (00 0054_{16}) is being driven on the PAB. During the next 32-bit fetch, Instructions 4 and 5 will be transferred from addresses 00 0054_{16} and 00 0055_{16} to the queue.

Example 4–1. Relationship Between Pipeline and Address Counters FC, IC, and PC





The remainder of this document refers almost exclusively to the PC. The FC and the IC are visible in only limited ways. For example, when a call is executed or an interrupt is initiated, the IC value is saved to the stack or to auxiliary register XAR7.

4.2 Visualizing Pipeline Activity

Consider Example 4–2, which lists eight instructions, I1–I8, and shows a diagram of the pipeline activity for those instructions. The F1 column shows addresses and the F2 column shows the instruction opcodes read at those addresses. During an instruction fetch, 32 bits are read, 16 bits from the specified address and 16 bits from the following address. The D1 column shows instructions being isolated in the instruction-fetch queue, and the D2 column indicates address generation and modification of address registers. The Instruction column shows the instructions that have reached the D2 phase. The R1 column shows addresses, and the R2 column shows the data values being read from those addresses. In the E column, the diagram shows results being written to the low half of the accumulator (AL). In the W column, address and a data values are driven simultaneously on the appropriate memory buses. For example, in the last active W phase of the diagram, the address 00 0205_{16} is driven on the data-write address bus (DWAB), and the data value 1234_{16} is driven on the data-write data bus (DWDB).

The highlighted blocks in Example 4–2 indicate the path taken by the instruction ADD AL,*AR0++. That path can be summarized as follows:

Phase	Activity Shown
F1	Drive address 00 0042_{16} on the program address bus (PAB).
F2	Read the opcodes F347 and F348 from addresses 00 0042_{16} and 00 0043_{16}, respectively.
D1	Isolate F348 in the instruction-fetch queue.
D2	Use AR0 = 0066_{16} to generate source address 00 0066_{16} and then increment AR0 to 0067_{16}.
R1	Drive address 00 0066_{16} on the data-read data bus (DRDB).
R2	Read the data value 1 from address 00 0066_{16}.
E	Add 1 to content of AL (1230_{16}) and store result (1231_{16}) to AL.
W	No activity

Example 4–2. Diagramming Pipeline Activity

```
Address  Opcode  Instruction                                    Initial Values
00 0040  F345    I1:  MOV  DP,#VarA  ; DP = page that has VarA.  VarA address = 00 0203
00 0041  F346    I2:  MOV  AL,@VarA  ; Move content of VarA to AL. VarA = 1230
00 0042  F347    I3:  MOVB AR0,#VarB ; AR0 points to VarB.       VarB address = 00 0066
00 0043  F348    I4:  ADD  AL,*AR0++ ; Add content of VarB to    VarB = 0001
                                     ; AL, and add 1 to AR0.    (VarB + 1) = 0003
00 0044  F349    I5:  MOV  @VarC,AL  ; Replace content of VarC   (VarB + 2) = 0005
                                     ; with content of AL.      VarC address = 00 0204
00 0045  F34A    I6:  ADD  AL,*AR0++ ; Add content of (VarB + 1) VarD address = 00 0205
                                     ; to AL, and add 1 to AR0.
00 0046  F34B    I7:  MOV  @VarD,AL  ; Replace content of VarD
                                     ; with content of AL.
00 0047  F34C    I8:  ADD  AL,*AR0   ; Add content of (VarB + 2)
                                     ; to AL.
```

F1	F2	D1	Instruction	D2	R1	R2	E	W
00 0040								
	F346:F345							
00 0042		F345						
	F348:F347	F346	I1: MOV DP,#VarA	DP = 8				
00 0044		F347	I2: MOV AL,@VarA	Generate VarA address	–			
	F34A:F349	F348	I3: MOVB AR0,#VarB	AR0 = 66	00 0203	–		
00 0046		F349	I4: ADD AL,*AR0++	AR0 = 67	–	1230	–	
	F34C:F34B	F34A	I5: MOV @VarC,AL	Generate VarC address	00 0066	–	AL=1230	–
		F34B	I6: ADD AL,*AR0++	AR0 = 68	–	0001	–	–
		F34C	I7: MOV @VarD,AL	Generate VarD address	00 0067	–	AL=1231	–
			I8: ADD AL,*AR0	AR0 = 68	–	0003	–	–
					00 0068	–	AL=1234	00 0204 1231
						0005	–	–
							AL=1239	00 0205 1234
								–

Note: The opcodes shown in the F2 and D1 columns were chosen for illustrative purposes; they are not the actual opcodes of the instructions shown.

The pipeline activity in Example 4–2 can also be represented by the simplified diagram in Example 4–3. This type of diagram is useful if your focus is on the path of each instruction rather than on specific pipeline events. In cycle 8, the pipeline is full: there is an instruction in every pipeline phase. Also, the effective execution time for each of these instructions is one cycle. Some instructions finish their activity at the D2 phase, some at the E phase, and some at the W phase. However, if you choose one phase as a reference, you can see that each instruction is in that phase for one cycle.

Example 4–3. Simplified Diagram of Pipeline Activity

F1	F2	D1	D2	R1	R2	E	W	Cycle
I1								1
I2	I1							2
I3	I2	I1						3
I4	I3	I2	I1					4
I5	I4	I3	I2	I1				5
I6	I5	I4	I3	I2	I1			6
I7	I6	I5	I4	I3	I2	I1		7
I8	I7	I6	I5	I4	I3	I2	I1	8
	I8	I7	I6	I5	I4	I3	I2	9
		I8	I7	I6	I5	I4	I3	10
			I8	I7	I6	I5	I4	11
				I8	I7	I6	I5	12
					I8	I7	I6	13
						I8	I7	14
							I8	15

4.3 Freezes in Pipeline Activity

This section describes the two causes for freezes in pipeline activity:

- ❏ Wait states
- ❏ An instruction-not-available condition

4.3.1 Wait States

When the core requests a read from or write to a memory device or peripheral device, that device may take more time to finish the data transfer than the core allots by default. Each device must use one of the core's *ready signals* to insert wait states into the data transfer when it needs more time. The core has three independent sets of ready signals: one set for reads from and writes to program space, a second set for reads from data space, and a third set for writes to data space. Wait-state requests freeze a portion of the pipeline if they are received during the F1, R1, or W phase of an instruction:

- ❏ **Wait states in the F1 phase.** The instruction-fetch mechanism halts until the wait states are completed. This halt effectively freezes activity for instructions in their F1, F2, and D1 phases. However, because the F1–D1 hardware and the D2–W hardware are decoupled, instructions that are in their D2–W phases continue to execute.

- ❏ **Wait states in the R1 phase.** All D2–W activities of the pipeline freeze. This is necessary because subsequent instructions can depend on the data-read taking place. Instruction fetching continues until the instruction-fetch queue is full or a wait-state request is received during an F1 phase.

- ❏ **Wait states in the W phase.** All D2–W activity in the pipeline freezes. This is necessary because subsequent instructions may depend on the write operation happening first. Instruction fetching continues until the instruction-fetch queue is full or a wait-state request is received during an F1 phase.

4.3.2 Instruction-Not-Available Condition

The D2 hardware requests an instruction from the instruction-fetch queue. If a new instruction has been fetched and has completed its D1 phase, the instruction is loaded into the instruction register for more decoding. However, if a new instruction is *not* waiting in the queue, an instruction-not-available condition exists. Activity in the F1–D1 hardware continues. However, the activity in the D2–W hardware ceases until a new instruction is available.

One time that an instruction-not-available condition will occur is when the first instruction after a discontinuity is at an odd address and has 32 bits. A *discontinuity* is a break in sequential program flow, generally caused by a branch, a call, a return, or an interrupt. When a discontinuity occurs, the instruction-fetch queue is emptied, and the CPU branches to a specified address. If the specified address is an odd address, a 16-bit fetch is performed at the odd address, followed by 32-bit fetches at subsequent even addresses. Thus, if the first instruction after a discontinuity is at an odd address and has 32 bits, two fetches are required to get the entire instruction. The D2–W hardware ceases until the instruction is ready to enter the D2 phase.

To avoid the delay where possible, you can begin each block of code with one or two (preferably two) 16-bit instructions:

```
FunctionA:
   16-bit instruction  ; First instruction
   16-bit instruction  ; Second instruction
   32-bit instruction  ; 32-bit instructions can start here
   .
   .
   .
```

If you choose to use a 32-bit instruction as the first instruction of a function or subroutine, you can prevent a pipeline delay only by making sure the instruction begins at an even address.

4.4 Pipeline Protection

Instructions are being executed in parallel in the pipeline, and different instructions perform modifications to memory and registers during different phases of completion. In an unprotected pipeline, this could lead to pipeline conflicts—reads and writes at the same location happening out of the intended order. However, the ’27xx pipeline has a mechanism that automatically protects against pipeline conflicts. There are two types of pipeline conflicts that can occur on the ’27xx:

- ❏ Conflicts during reads and writes to the same data-space location
- ❏ Register conflicts

The pipeline prevents these conflicts by adding inactive cycles between instructions that would cause the conflicts. Sections 4.4.1 and 4.4.2 explain the circumstances under which these pipeline-protection cycles are added and tells how to avoid them, so that you can reduce the number of inactive cycles in your programs.

4.4.1 Protection During Reads and Writes to the Same Data-Space Location

Consider two instructions, A and B. Instruction A writes a value to a memory location during its W phase. Instruction B must read that value from the same location during its R1 and R2 phases. Because the instructions are being executed in parallel, it is possible that the R1 phase of instruction B could occur before the W phase of instruction A. Without pipeline protection, instruction B could read too early and fetch the wrong value. The ’27xx pipeline prevents that read by holding instruction B in its D2 phase until instruction A is finished writing.

Example 4–4 shows a conflict between two instructions that are accessing the same data-memory location. The pipeline activity shown is for an *unprotected* pipeline. For convenience, the F1–D1 phases are not shown. I1 writes to VarA during cycle 5. Data memory completes the store in cycle 6. I2 should not read the data-memory location any sooner than cycle 7. However, I2 performs the read during cycle 4 (three cycles too early). To prevent this kind of conflict, the pipeline-protection mechanism would hold I2 in the D2 phase for 3 cycles. During these *pipeline-protection cycles*, no new operations occur.

Example 4–4. Conflict Between a Read From and a Write to Same Memory Location

```
I1:   MOV @VarA,AL ; Write AL to data–memory location
I2:   MOV AH,@VarA ; Read same location, store value in AH
```

D2	R1	R2	E	W	Cycle	Comments
I1					1	
I2	I1				2	
	I2	I1			3	I2 issues read request
		I2	I1		4	I2 reads in R2 phase
			I2	I1	5	I1 writes in W phase
				I2	6	I2 should issue read request here
					7	
					8	

You can reduce or eliminate these types of pipeline-protection cycles if you can take other instructions in your program and insert them between the instructions that conflict. Of course, the inserted instructions must not cause conflicts of their own or cause improper execution of the instructions that follow them. For example, the code in Example 4–4 could be improved by moving a CLRC instruction to the position between the MOV instructions (assume that the instructions following CLRC SXM operate correctly with SXM = 0):

```
I1:   MOV  @VarA,AL ; Write AL to data–memory location
      CLRC SXM      ; SXM = 0 (sign extension off)
I2:   MOV  AH,@VarA ; Read same location, store value in AH
```

Inserting the CLRC instruction between I1 and I2 reduces the number of pipeline-protection cycles to two. Inserting two more instructions would remove the need for pipeline protection. As a general rule, if a read operation occurs within three instructions from a write operation to the same memory location, the pipeline protection mechanism adds at least one inactive cycle.

4.4.2 Protection Against Register Conflicts

All reads from and writes to CPU registers occur in either the D2 phase or the E phase of an instruction. A register conflict arises when an instruction attempts to read and/or modify the content of a register (in the D2 phase) before a previous instruction has written to that register (in the E phase).

The pipeline-protection mechanism resolves register conflicts by holding the later instruction in its D2 phase for as many cycles as needed (one to three). You do not have to consider register conflicts unless you wish to achieve maximum pipeline efficiency. If you choose to reduce the number of pipeline-protection cycles, you can identify the pipeline phases in which registers are accessed and try to move conflicting instructions away from each other.

Generally, a register conflict involves one of the address registers:

- 16-bit auxiliary registers AR0–AR7
- 22-bit auxiliary registers XAR6 and XAR7
- 16-bit data page pointer (DP)
- 16-bit stack pointer (SP)

Example 4–5 shows a register conflict involving auxiliary register AR0. The pipeline activity shown is for an *unprotected* pipeline, and for convenience, the F1–D1 phases are not shown. I1 writes to AR0 at the end of cycle 4. I2 should not attempt to read AR0 until cycle 5. However, I2 reads AR0 (to generate an address) during cycle 2. To prevent this conflict, the pipeline-protection mechanism would hold I2 in the D2 phase for three cycles. During these cycles, no new operations occur.

Example 4–5. Register Conflict

```
I1: MOV AR0,@7    ; Load AR0 with the value addressed by
                  ; the operand @7.
I2: MOV AH,*AR0   ; Load AH with the value pointed to by
                  ; AR0.
```

D2	R1	R2	E	W	Cycle	Comments
I1					1	
I2	I1				2	I2 reads in D2 phase
	I2	I1			3	
		I2	I1		4	I1 writes in E phase
			I2	I1	5	I2 should read here
				I2	6	

You can reduce or eliminate pipeline-protection cycles due to a register conflict by inserting other instructions between the instructions that cause the conflict. For example, the code in Example 4–5 could be improved by moving two other instructions from elsewhere in the program (assume that the instructions following SETC SXM operate correctly with PM = 1 and SXM = 1):

```
I1: MOV AR0,@7    ; Load AR0 with the value addressed by
                  ; the operand @7.
    SPM 0         ; PM = 1 (no product shift)
    SETC SXM      ; SXM = 1 (sign extension on)
I2: MOV AH,*AR0   ; Load AH with the value pointed to by
                  ; AR0.
```

Inserting the SPM and SETC instructions reduces the number of pipeline-protection cycles to one. Inserting one more instruction would remove the need for pipeline protection. As a general rule, if a read operation occurs within three instructions from a write operation to the same register, the pipeline-protection mechanism adds at least one inactive cycle.

4.5 Avoiding Unprotected Operations

This section describes pipeline conflicts that the pipeline-protection mechanism does not protect against. These conflicts are avoidable, and this section offers suggestions for avoiding them.

4.5.1 Unprotected Program-Space Reads and Writes

The pipeline protects only register and data-space reads and writes. It does not protect the program-space reads done by the PREAD and MAC instructions or the program-space write done by the PWRITE instruction. Be careful with these instructions when using them to access a memory block that is shared by data space and program space.

As an example, suppose a memory location can be accessed at address 00 $0D50_{16}$ in program space and address 00 $0D50_{16}$ in data space. Consider the following lines of code:

```
; XAR7 = 000D50 in program space
; Data1 = 000D50 in data space
ADD   @Data1,AH    ; Store AH to data–memory location
                   ; Data1.
PREAD @AR1,*XAR7   ; Load AR1 from program–memory
                   ; location given by XAR7.
```

The operands @Data1 and *XAR7 are referencing the same location, but the pipeline cannot interpret this fact. The PREAD instruction reads from the memory location (in the R2 phase) before the ADD writes to the memory location (in the W phase).

However, the PREAD is not necessary in this program. Because the location can be accessed by an instruction that reads from data space, you can use another instruction, such as a MOV instruction:

```
ADD   @Data1,AH    ; Store AH to memory location Data1.
MOV   AR1,*XAR7    ; Load AR1 from memory location
                   ; given by XAR7.
```

4.5.2 An Access to One Location That Affects Another Location

If an access to one location affects another location, you may need to correct your program to prevent a pipeline conflict. You only need to be concerned about this kind of pipeline conflict if you are using the T3202700 core or you are using the T3202700B0 core and are addressing a location outside of a protected address range. (See section 4.5.3.). Consider the following example:

```
     MOV @DataA,#4  ; This write to DataA causes a
                    ; peripheral to clear bit 15 of DataB.
$10: TBIT@DataB,#15 ; Test bit 15 of DataB.
     SB  $10,NTC    ; Loop until bit 15 is set.
```

This program causes a misread. The TBIT instruction reads bit 15 (in the R2 phase) before the MOV instruction writes to bit 15 (in the W phase). If the TBIT instruction reads a 1, the code prematurely ends the loop. Because DataA and DataB reference different data-memory locations, the pipeline does not identify this conflict.

However, you can correct this type of error by inserting two or more NOP (no operation) instructions to allow for the delay between the write to DataA and the change to bit 15 of DataB. For example, if a 2-cycle delay is sufficient, you can fix the previous code as follows:

```
     MOV @DataA,#4  ; This write to DataA causes a
                    ; peripheral to clear bit 15 of DataB.
     NOP            ; Delay by 1 cycle.
     NOP            ; Delay by 1 cycle.
$10: TBIT@DataB,#15 ; Test bit 15 of DataB.
     SB  $10,NTC    ; Loop until bit 15 is set.
```

4.5.3 Write Followed By Read Protection Mode

The T320C27B0 core contains a write followed by read protection mode to ensure that any read operation that follows a write operation within a protected address range is executed as written by delaying the read operation until the write is initiated.

The PROTSTART(15:0) and PROTRANGE(15:0) input signals set the protection range. The PROTRANGE(15:0) value is a binary multiple with the smallest block size being 64 words, and the largest being 4M words (64 words, 128 words, 256 words ...1M words, 2M words, 4M words). The PROTSTART address must always be a multiple of the chosen range. For example, if a 4K block size is selected, then the start address must be a multiple of 4K.

The ENPROT signal enables this feature (when set high), iir disables this feature (when set low)

All of the above signals are latched on every cycle. Typically, you would hard code the above to a a given specified peripheral range and group all peripherals within this range. If programmability is required (such as on an emulation device), the above signals can be connected to a register and changed within the application program.

The above mechanism only works for reads that follow writes to the protected area. Reads and write sequences to unprotected areas are not affected, as shown in the following examples.

```
Example 1: write protected_area
           write protected_area
           write protected_area
                                    <– pipe protection
                                        (3 cycles)
           read protected_area


Example 2: write protected_area
           write protected_area
           write protected_area
                                    <– no pipe protection
                                        invoked
           read non_protected_area
                                    <– pipe protection
                                         (2 cycles)
           read protected_area
           read protected_area

Example 3: write non_protected_area
           write non_protected_area
           write non_protected_area
                                    <– no pipe protection
                                        invoked
           read protected_area
```

Chapter 5

Addressing Modes

This chapter explains the modes by which the ’27x instruction set accepts constants and addresses memory locations. There are four main modes:

- ❏ Immediate addressing
- ❏ Register addressing
- ❏ Direct addressing
- ❏ Indirect addressing (this includes circular addressing)

When you need to enter a constant as an operand, use immediate-constant addressing mode. When you need to access data memory, use direct, indirect, or immediate-pointer addressing mode. You can point to a program-memory address using XAR7 indirect addressing mode or immediate-pointer addressing mode. Register addressing mode enables you to directly access CPU registers.

When direct and indirect addressing modes are used, the address is generated by the address register arithmetic unit (ARAU).

For more information about the instructions shown in examples throughout this chapter, see Chapter 6, *Assembly Language Instructions*.

Topic **Page**

5.1 Immediate Addressing Modes 5-2
5.2 Direct Addressing Modes 5-4
5.3 Register Addressing Mode 5-6
5.4 Indirect Addressing Modes That Use the Stack Pointer 5-7
5.5 Indirect Addressing Modes That Use Auxiliary Registers 5-10
5.6 Summary of Indirect-Addressing Operands 5-18
5.7 Addressing-Mode Information in Opcodes 5-21

5.1 Immediate Addressing Modes

The ’27x supports two immediate addressing modes:

- ❑ Immediate-constant addressing
- ❑ Immediate-pointer addressing

5.1.1 Immediate-Constant Addressing Mode

As shown in the following examples, instructions that support immediate-constant addressing take a constant as an operand. The constant must be preceded by the symbol #.

```
ADD   ACC, #8   ; Add 8 to accumulator.
SUBB  SP, #10   ; Subtract 10 from stack pointer.
MOVW  DP, #200  ; Load data page pointer with 200.
```

The syntax determines the range of numbers that can be supplied in the operand. For example, this is the syntax for the MOV instruction shown above:

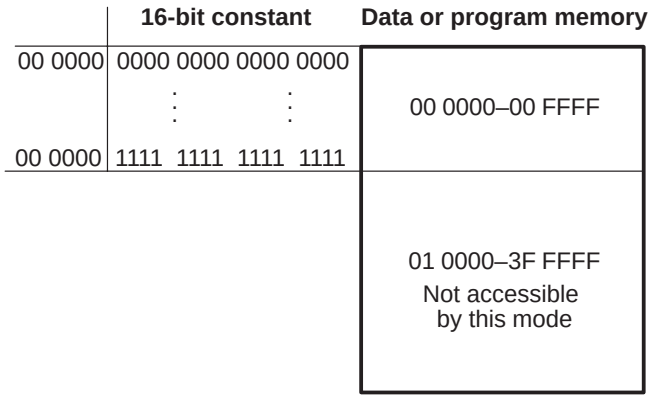
MOV DP, #*10bit*

The variable *10bit* represents a 10-bit unsigned constant (a value in the range 0 to 1023).

5.1.2 Immediate-Pointer Addressing Mode

Immediate-pointer addressing mode allows you to reference a 22-bit data-memory or program-memory address by supplying the 16 least significant bits of the address as a constant. The six most significant bits of the address are filled with zeros. Because these six bits are always zeros, the address reach of this mode is limited to the address range $00\ 0000_{16}$–$00\ FFFF_{16}$ (see Figure 5–1).

Figure 5–1. Accessing Memory in Immediate-Pointer Addressing Mode



This mode is only used by the following syntaxes of the MOV and MAC instructions:

- **MOV** *loc*, ***(0:***16bit***)**

  The 22-bit immediate address is the value 0:*16bit*. The value you supply for *16bit* is concatenated with six leading zeros to form the 22-bit address. This form of the MOV instruction loads the value at data-memory address (0:*16bit*) to the data-memory or register referenced by *loc*. The instruction takes two words, where the second word is the 16-bit constant.

- **MOV *(0:***16bit***),** *loc*

  Again, the immediate address is the value 0:*16bit*. This form of the MOV instruction loads the data-memory or register value referenced by *loc* to data-memory address 0:*16bit*. The instruction takes two words, where the second word is the 16-bit constant.

- **MAC P,** *loc*, **0:***pmem*

  Unlike the immediate address in the MOV instructions, this address is neither preceded by an asterisk (*) nor contained within parentheses. In addition, this pointer is to *program* memory, not data memory. As part of its operation, this MAC instruction multiplies the value at program-memory address 0:*pmem* by the value referenced by *loc*. The instruction takes two words, where the second word is the 16-bit constant *pmem*.

5.2 Direct Addressing Modes

In the direct addressing modes, data memory is seen as blocks of 64 words called data pages. As shown in Figure 5–2, the entire 4M words of data memory consists of 65 536 data pages labeled 0 through 65 535.

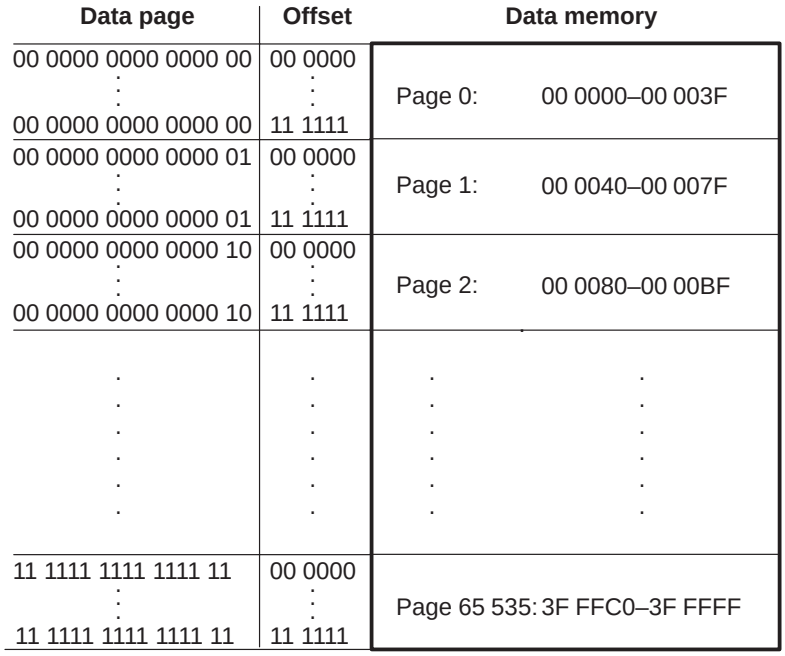
The data page is reflected in the 16 most significant bits (MSBs) of an address. For example, if the 16 MSBs are 0s, the address is in data page 0; if the 16 MSBs are all 1, the address is in data page 65 535.

The 6 least significant bits of an address are known as the offset. For example, an offset of 0 ($00\ 0000_2$) indicates the first word on a page; an offset of 63 ($11\ 1111_2$) indicates the last word on a page.

There are two direct addressing modes:

- ❑ DP direct addressing mode
- ❑ PAGE0 direct addressing mode

Figure 5–2. Pages of Data Memory



5.2.1 DP Direct Addressing Mode

When you use DP direct addressing, the data page is determined by the value in the 16-bit data page pointer (DP), an independent CPU register. For

example, if the DP value is 0000_{16}, the data page is 0. If the DP value is 0002_{16}, the data page is 2. To indicate which word on a page is accessed, you supply a 6-bit offset preceded by the symbol @. In the following example, the operand @1 indicates an offset of 1.

```
ADD    ACC, @1
```

To form a complete 22-bit address, the address register arithmetic unit (ARAU) concatenates the DP value and the 6-bit offset. Suppose DP is 1 ($0000\ 0000\ 0000\ 0001_2$) and the offset is 63 ($11\ 1111_2$). When concatenated, they form the address of the last word on data page 1 ($00\ 0000\ 0000\ 0000\ 0111\ 1111_2$).

Before you use DP direct addressing mode, make sure the DP contains the desired page number (from 0 to 65 535). For example:

```
MOVW   DP, #32   ; Set data page to 32
ADD    ACC, @1   ; Add second word on page 32 to ACC.
```

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.

5.2.2 PAGE0 Direct Addressing Mode

Instructions that use PAGE0 direct addressing mode always access data page 0 (addresses $00\ 0000_{16}$–$00\ 003F_{16}$), regardless of the DP value. You indicate a particular word by supplying an operand of the form @0:*6bit*, where *6bit* is the 6-bit offset. The following example shows an offset of 1. To form a complete 22-bit address, the processor concatenates 16 leading zeros and the six bits of the offset.

```
ADD    ACC, @0:1
```

PAGE0 direct addressing mode is only available when the PAGE0 bit in status register ST1 is 1. (ST1 is described in section 2.4 on page 2-19.) Before using this mode, make sure PAGE0 = 1. For example:

```
SETC   PAGE0     ; Turn PAGE0 direct addressing on.
ADD    ACC, @0:1 ; Add second word on page 0 to ACC.
```

Note:

There are two mutually-exclusive PAGE0 addressing modes:

- ❑ PAGE0 direct addressing
- ❑ PAGE0 stack addressing (described in section 5.4.2)

When PAGE0 = 1, you cannot use PAGE0 stack addressing. Likewise, when PAGE0 = 0, you cannot use PAGE0 direct addressing. You can switch freely between the modes using the SETC and CLRC instructions.

5.3 Register Addressing Mode

Register addressing mode allows you to access certain CPU registers by name instead of by address. In addition, this mode allows you to transfer data from one register to another without having to store the data to memory first. Table 5–1 explains the available operands for this mode. You can precede the register name with the symbol @, but the symbol is optional. This document uses @ to highlight register-addressing operands.

The operand @XARn is only supported by these instructions: ADDL, CMPL, MOVL, or SUBL. In addition, none of the other register-addressing operands can be used with ADDL, CMPL, MOVL, or SUBL.

Two examples follow. In the MOV instruction, @SP is a register-addressing operand, but AR2 is not. AR2 is a part of the particular syntax of the MOV instruction.

```
PUSH   @AL       ; Save low accumulator to stack.
MOV    AR2, @SP  ; Store stack pointer to auxiliary
                 ; register 2.
```

The registers shown in Table 5–1 are introduced in section 2.2, *CPU Registers*, on page 2-4.

Table 5–1. Operands for Register Addressing Mode

Operand†	Description	Example
@AH	High word of accumulator	**MOV @AH, #0** loads the high word of the accumulator with 0.
@AL	Low word of accumulator	**MOV AH, @AL** loads the low word of the accumulator into the high word of the accumulator.
@PH	High word of product register	**MOV AR4, @PH** loads AR4 with the high word of the P register.
@PL	Low word of product register	**MOV T, @PL** loads the low word of the P register into the T register.
@T	Multiplicand register	**ADD @T, #5** adds 5 to the T register value.
@SP	Stack pointer	**ADD @SP, AL** adds the content of AL to SP.
@ARx	Auxiliary register x x = 0, 1, 2, 3, 4, 5, 6, or 7	**ADD ACC, @AR0** adds the content of AR0 to ACC.
@XARn	Extended auxiliary register n n = 6 or 7	**SUBL ACC, @XAR6** subtracts the content of XAR6 from the accumulator.

† The symbol @ is optional for register addressing mode.

5.4 Indirect Addressing Modes That Use the Stack Pointer

The stack pointer (SP) always points to the next empty location in the software stack in data memory. Two indirect addressing modes use the stack pointer when referencing data memory:

- ❑ Stack-pointer indirect addressing mode
- ❑ PAGE0 stack addressing mode

Stack-pointer indirect addressing mode is used to access the data-memory location that is pointed to by the SP. PAGE0 stack addressing is used to access a data-memory location relative to the position of the SP. Because the stack pointer has only 16 bits, the reach of these modes is limited to the first 64K of data memory (addresses 00 0000_{16}–00 $FFFF_{16}$).

5.4.1 Stack-Pointer Indirect Addressing Mode

In stack-pointer indirect addressing mode, the stack pointer is used to point to the data-memory address where a read or write is to occur. As shown in Figure 5–3, the 22-bit address is the concatenation of six leading zeros and the 16 bits of the SP. Because the six most significant bits are always 0s in this mode, its reach is limited to the address range 00 0000_{16}–00 $FFFF_{16}$.

Figure 5–3. Accessing Data Memory in Stack-Pointer Indirect Addressing Mode

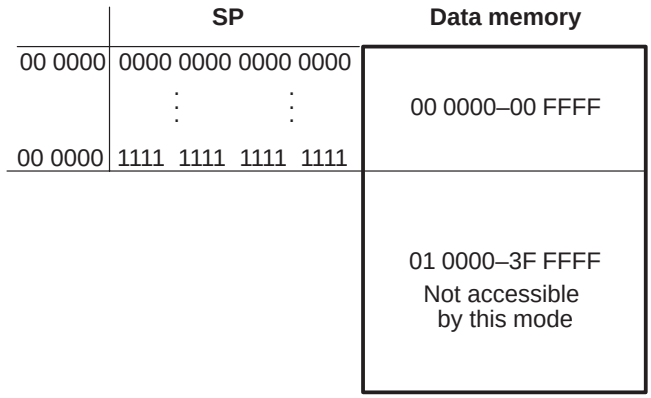


Table 5–2 explains the two operands available for stack-pointer indirect addressing mode. The increment or decrement of SP is performed by the address register arithmetic unit (ARAU) in the decode 2 phase of the pipeline.

Table 5–2. Operands for Stack-Pointer Indirect Addressing Mode

Operand	Description	Example

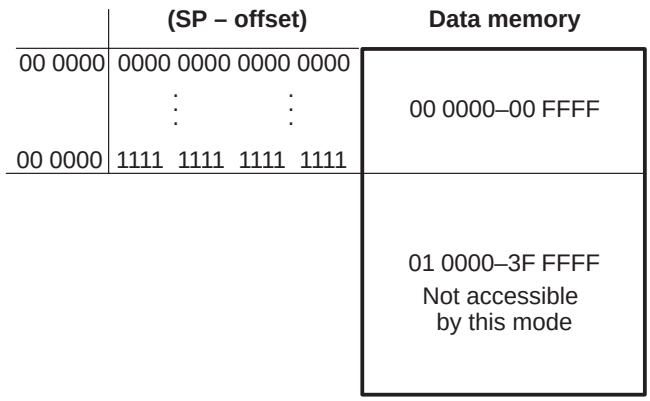
*SP++	Increment SP *after* the data-memory access.
	If (16-bit operation)
	SP = SP +1
	If (32-bit operation)
	SP = SP +2
		MOV *SP++, AH loads the high word of the accumulator to the 16-bit location pointed to by SP and increments the content of SP. Because the load is a 16-bit operation, SP is incremented by 1.

*––SP	Decrement SP *before* the data-memory access.
	If (16-bit operation)
	SP = SP – 1
	If (32-bit operation)
	SP = SP – 2
		MOVL ACC, *––SP decrements the content of SP and loads ACC with the 32-bit value referenced by SP. Because the load is a 32-bit operation, SP is decremented by 2.

5.4.2 PAGE0 Stack Addressing Mode

PAGE0 stack addressing mode allows an instruction to access a data-memory location relative to the position of the SP. In the instruction, you specify a positive 6-bit offset (from 0 to 63), and the address register arithmetic unit (ARAU) subtracts the offset from a copy of the SP value (SP is not modified). The result, (SP – offset), is concatenated with six leading zeros to form the full 22-bit address (see Figure 5–4). Because the six most significant bits are always 0s in this mode, the reach of the mode is limited to the address range 000000_{16}–$00FFFF_{16}$.

Figure 5–4. Accessing Data Memory in PAGE0 Stack Addressing Mode



PAGE0 stack addressing mode is only available when the PAGE0 bit in status register ST1 is 0. (ST1 is described in section 2.4 on page 2-19.) The general form for an operand in this mode is *–SP[*6bit*], where *6bit* is the 6-bit offset. Here is an example of using this mode:

```
CLRC   PAGE0        ; Select PAGE0 stack addressing mode.
SUB    ACC,*–SP[4]  ; Subtract data-memory value from ACC.
                    ; Value is 4 locations above the
                    ; address in SP. SP is not modified.
```

Notes:

There are two mutually-exclusive PAGE0 addressing modes:

- ❑ PAGE0 direct addressing (described in section 5.2)
- ❑ PAGE0 stack addressing

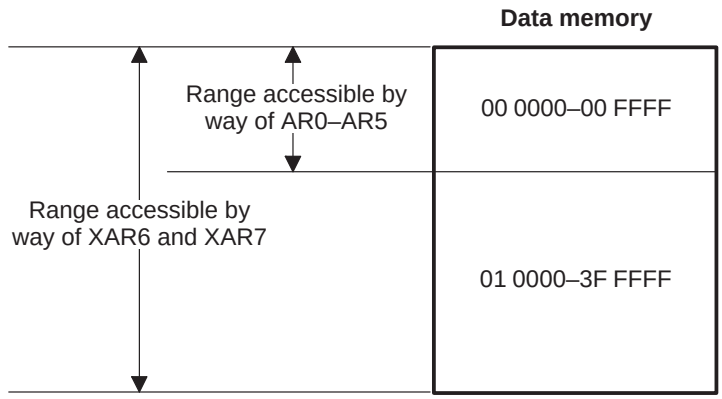
When PAGE0 = 1, you cannot use PAGE0 stack addressing. Likewise, when PAGE0 = 0, you cannot use PAGE0 direct addressing. You can switch freely between the modes using the SETC and CLRC instructions.

Do not use the operand *–SP[0] because that operation accesses the location pointed to by the stack pointer. This location is overwritten during interrupts, so storing information in this location is not advised.

5.5 Indirect Addressing Modes That Use Auxiliary Registers

For indirect addressing modes other than stack-pointer indirect addressing mode and PAGE0 stack addressing mode, the CPU provides eight auxiliary registers as pointers to memory. There are six 16-bit auxiliary registers—AR0, AR1, AR2, AR3, AR4, and AR5—and two 22-bit extended auxiliary registers—XAR6 and XAR7. Figure 5–5 shows the ranges of data memory accessible to the auxiliary registers. XAR7 can also be used by some instructions to point to any value in program memory; see section 5.5.3, *XAR7 Indirect Addressing Mode*.

Figure 5–5. Accessing Data Memory With the Auxiliary Registers



There are four indirect addressing modes that use auxiliary registers:

- ❑ Auxiliary-register indirect addressing mode (see section 5.5.1)
- ❑ ARP indirect addressing mode (see section 5.5.2)
- ❑ XAR7 indirect addressing mode (see section 5.5.3)
- ❑ Circular addressing mode (see section 5.5.4)

5.5.1 Auxiliary-Register Indirect Addressing Mode

Auxiliary-register indirect addressing mode enables you to point to addresses $00\ 0000_{16}$–$00\ FFFF_{16}$ using AR0–AR5 and point to addresses $00\ 0000_{16}$–$3F\ FFFF_{16}$ using XAR6 and XAR7. You can add an offset to an auxiliary register when you want to access a location relative to the current position of that auxiliary register. You can specify one of three sources for the offset:

- **Auxiliary register AR0.** After you load AR0 with a value from 0 to 65 535 (0000_{16} to $FFFF_{16}$), you specify AR0 as the offset in the instruction. For example:

```
ADD AL,*+AR3[AR0]  ; Concatenate sum (AR3 + AR0) with
                   ; six zeros to form address
                   ; 0:(AR3 + AR0). Add content of that
                   ; address to the low accumulator.
```

- **Auxiliary register AR1.** After you load AR1 with a value from 0 to 65 535, you specify AR1 as the offset in the instruction. For example:

```
ADD AL,*+XAR6[AR1] ; Calculate address (XAR6 + AR1).
                   ; Add content of that
                   ; address to the low accumulator.
```

- **A constant.** In an operand, you specify a value from 0 to 7. For example:

```
ADD AL,*+AR2[4]    ; Concatenate sum (AR2 + 4) with
                   ; six zeros to form address
                   ; 0:(AR2 + 4). Add content of that
                   ; address to the low accumulator.
```

When you use an operand that specifies an offset, the auxiliary register is *not* modified. The offset is only used to generate the address.

Note:

Some syntaxes of the MOVB instruction use the offset to indicate relative byte positions rather than relative word positions. For the details, see the description for MOVB, which begins on page 6-158.

Table 5–3 lists and explains the operands that are available for auxiliary-register indirect addressing mode. The increments and decrements are performed by the address register arithmetic unit (ARAU) in the decode 2 phase of the pipeline. The auxiliary register pointer (ARP) in status register ST1 is updated by all these operands. This update also happens in the decode 2 phase of the pipeline.

Table 5–3. Operands for Auxiliary-Register Indirect Addressing Mode

Operand	Description	Example

*ARx++
or
*XARn++
	Increment the specified auxiliary register *after* the data-memory access. Also, set ARP to x or n.
	If (16-bit operation)
	ARx/XARn = ARx/XARn + 1
	If (32-bit operation)
	ARx/XARn = ARx/XARn + 2
		MOV *AR2++, AH loads the high word of the accumulator to the address 0:AR2, increments the content of AR2, and sets ARP = 2.

*––ARx
or
*––XARn
	Decrement the specified auxiliary register *before* the data-memory access. Also, set ARP to x or n.
	If (16-bit operation)
	ARx/XARn = ARx/XARn – 1
	If (32-bit operation)
	ARx/XARn = ARx/XARn – 2
		MOV *––XAR6, AH decrements the content of XAR6, loads the high word of the accumulator to the address pointed to by XAR6, and sets ARP = 6.

*+ARx[AR0] or *+XARn[AR0]	Use the address 0:(ARx + AR0) or (XARn + AR0) for the data-memory access. Also, set ARP to x or n. Do not modify ARx or XARn.	**MOV *+AR4[AR0], AH** loads the high word of the accumulator to the 22-bit address 0:(AR4 + AR0) and sets ARP = 4. AR4 is not modified.
*+ARx[AR1] or *+XARn[AR1]	Use the address 0:(ARx + AR1) or (XARn + AR1) for the data-memory access. Also, set ARP to x or n. Do not modify ARx or XARn.	**MOV *+XAR7[AR1], AH** loads the high word of the accumulator to the 22-bit address (XAR7 + AR1) and sets ARP = 7. XAR7 is not modified.

Notes: 1) ARx = AR0, AR1, AR2, AR3, AR4, or AR5

2) XARn = XAR6 or XAR7

3) *3bit* is a 3-bit constant (a value from 0 to 7).

4) The notation 0:ARx or 0:(ARx + n) indicates that when a 16-bit auxiliary register is used, the six MSBs of the data-memory address are 0s.

Table 5–3. Operands for Auxiliary-Register Indirect Addressing Mode (Continued)

Operand	Description	Example
*+ARx[*3bit*] or *+XARn[*3bit*]	Use the address 0:(ARx + *3bit*) or (XARn + *3bit*) for the data-memory access. Also, set ARP to x or n. Do not modify ARx or XARn.	**MOV *+AR5[6], AH** loads the high word of the accumulator to the 16-bit location pointed to by 0:(AR5 + 6) and sets ARP = 5. AR5 is not modified.
*ARx or *XARn	Equivalent to *+ARx[0] or *+XARn[0]. Use the address in ARx or XARn for the data-memory access. Also, set ARP to x or n. Do not modify ARx or XARn.	**MOV *AR5, AH** loads the high word of the accumulator to the 16-bit location pointed to by AR5 and sets ARP = 5. AR5 is not modified.

Notes: 1) ARx = AR0, AR1, AR2, AR3, AR4, or AR5

2) XARn = XAR6 or XAR7

3) *3bit* is a 3-bit constant (a value from 0 to 7).

4) The notation 0:ARx or 0:(ARx + n) indicates that when a 16-bit auxiliary register is used, the six MSBs of the data-memory address are 0s.

5.5.2 ARP Indirect Addressing Mode

In ARP indirect addressing, the name of the auxiliary register is not specified in the instruction; rather, the instruction uses the auxiliary register pointed to by the 3-bit auxiliary register pointer (ARP) of status register ST1. (ST1 is described in section 2.4 on page 2-19.) Before using ARP indirect addressing mode, make sure ARP holds the appropriate value from 0 to 7 (0 is for AR0, 1 is for AR1, 2 is for AR2, and so on). The register pointed to by the ARP is referred to as the *current auxiliary register*.

You can add an offset when you want to access a location relative to the position of an auxiliary register. The source for the offset is AR0. After you load AR0 with a value from 0 to 65 535 (0000_{16} to $FFFF_{16}$), you use one of two operands: *0++ or *0––. The operand *0++ adds AR0 to the current auxiliary register; *0–– subtracts AR0 from the current auxiliary register. Here is an example:

```
ADD    AL,*0++  ; Add content of current auxiliary
                ; register to low half of
                ; accumulator. Then add content of
                ; AR0 to current auxiliary register.
```

Table 5–4 lists and explains the operands that are available for ARP indirect addressing mode. The increments and decrements are performed by the address register arithmetic unit (ARAU) in the decode 2 phase of the pipeline.

To load the ARP without performing any other action, you can use a NOP (no operation) instruction. For example, either of the following two instructions sets ARP to 2 and does nothing else.

```
NOP *AR2     ; Auxiliary–register indirect addressing mode
NOP *ARP2    ; ARP indirect addressing mode
```

Table 5–4. Operands for ARP Indirect Addressing Mode

Operand	Description	Example
*ARPx	Set ARP to x x Selects x Selects 0 AR0 4 AR4 1 AR1 5 AR5 2 AR2 6 XAR6 3 AR3 7 XAR7	**MOV *ARP2, AH** loads the high word of the accumulator to the 16-bit location pointed to by the current auxiliary register and then sets ARP to 2.
*	No increment or decrement of current auxiliary register.	**MOV *, AH** loads the high word of the accumulator to the 16-bit location pointed to by the current auxiliary register and does nothing to the content of the current AR.

*++	Increment current auxiliary register *after* the data-memory access.
	If (16-bit operation)
	ARx/XARn = ARx/XARn + 1
	If (32-bit operation)
	ARx/XARn = ARx/XARn + 2
		MOV *++, AH loads the high word of the accumulator to the 16-bit location pointed to by the current auxiliary register and increments the content of the current AR by 1.

*––	Decrement current auxiliary register *after* the data-memory access.
	If (16-bit operation)
	ARx/XARn = ARx/XARn – 1
	If (32-bit operation)
	ARx/XARn = ARx/XARn – 2
		MOV *––, AH loads the high word of the accumulator to the 16-bit location pointed to by the current auxiliary register and decrements the content of the current AR by 1.

Table 5–4. Operands for ARP Indirect Addressing Mode (Continued)

Operand	Description	Example
*0++	Increment current auxiliary register by index amount *after* the data-memory access. The index amount is the value in AR0.	**MOV *0++, AH** loads the high word of the accumulator to the 16-bit location pointed to by the current auxiliary register and adds the content of AR0 to the content of the current auxiliary register.
*0––	Decrement current auxiliary register by index amount *after* the data-memory access. The index amount is the value in AR0.	**MOV *0––, AH** loads the high word of the accumulator to the 16-bit location pointed to by the current auxiliary register and subtracts the content of AR0 from the content of the current auxiliary register.

5.5.3 XAR7 Indirect Addressing Mode

XAR7 indirect addressing mode allows you to use the extended auxiliary register XAR7 to point to an address in program memory. This mode has a single operand, *XAR7, and is only used for the following instructions. This operand does *not* affect the auxiliary register pointer (ARP).

- **PREAD** *operand1*, ***XAR7**

  where *operand1* represents a data-memory address. This instruction moves a 16-bit value from the program-memory location referenced by XAR7 to the data-memory location referenced by *operand1*.

- **PWRITE *XAR7,** *operand2*

  where *operand2* represents a data-memory address. This instruction moves a 16-bit value from the data-memory location referenced by *operand2* to the program-memory location referenced by XAR7.

- **LB *XAR7**

  This instruction forces a branch to the program-memory location referenced by XAR7.

- **CALL *XAR7**

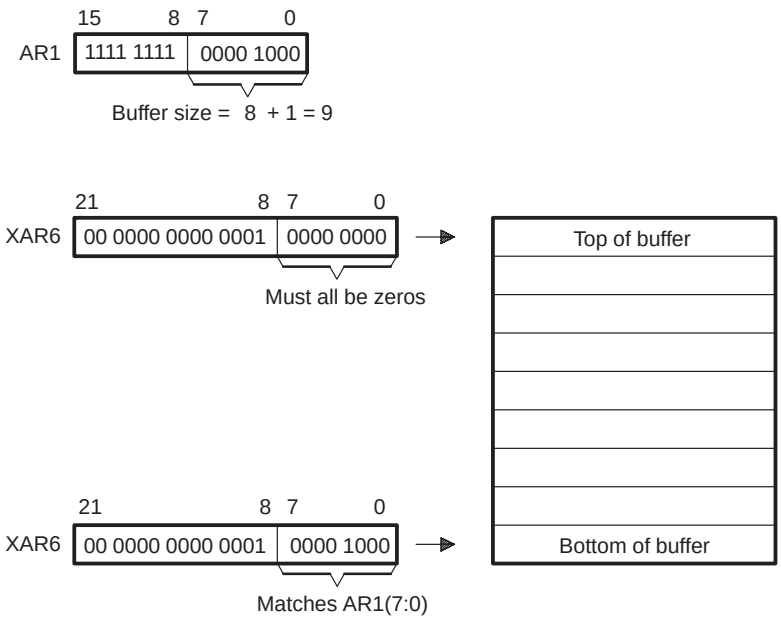
  This instruction calls a subroutine at the program-memory location referenced by XAR7.

5.5.4 Circular Addressing Mode

Circular addressing is used to access a user-defined buffer in data memory. Example 5–1 illustrates these characteristics of the buffer:

- The buffer size is determined by the eight least significant bits (LSBs) of auxiliary register AR1, or AR1(7:0). Specifically, the buffer size is AR1(7:0) plus 1. When AR1(7:0) is 255, the buffer is at its maximum size of 256.
- Auxiliary register XAR6 points to the current address in the buffer. The top of the buffer must be at an address whose eight LSBs are zeros. When the eight LSBs of XAR6 match the eight LSBs of AR1, XAR6 is pointing to the bottom of the buffer.

Example 5–1. Circular Addressing Mode



During a series of circular-addressing operations, the value in the 16 LSBs of XAR6 (AR6) is incremented until the eight LSBs of AR6 match the eight LSBs of AR1. When the values match, the eight LSBs of AR6 are cleared, so that XAR6 points to the top of the buffer. The value in the upper six bits of XAR6 is not modified during any of the above operations.

Circular addressing mode has a single operand, *AR6%++. This is the operation performed when you use this operand:

1) Set the auxiliary register pointer (ARP) to 6, and perform the read from or write to data memory. The address in XAR6 identifies the data-memory location where the access takes place.

2) If the eight LSBs of AR6 match the eight LSBs of AR1 (the bottom of the buffer has been reached), clear AR6(7:0). If the 8-bit values do not match, increment AR6 as follows:

   - ❏ If the instruction accessed one 16-bit location, increment AR6 by 1.
   - ❏ If the instruction accessed two 16-bit locations, increment AR6 by 2.

   The increment of AR6 is performed by the address register arithmetic unit (ARAU) in the decode 2 phase of the pipeline. The six MSBs of XAR6 are not affected.

Here is an example of using circular addressing mode:

```
; AR6(7:0) is not equal to AR1(7:0)
MOV   *AR6%++,AH   ; Load high word of the accumulator to
                   ; the address referenced by XAR6,
                   ; increment AR6 by 1, and set ARP = 6.
```

Because this MOV instruction writes to only one 16-bit data-memory location, AR6 is incremented by 1. In the following example, the ADDL instruction performs a 32-bit operation; that is, it accesses two data-memory locations. As a result, AR6 is incremented by 2.

```
; AR6(7:0) is not equal to AR1(7:0)
ADDL  ACC,*AR6%++ ; Read the 32-bit value at the addresses
                  ; referenced by XAR6. Add the 32-bit
                  ; value to ACC. Increment AR6 by 2.
                  ; Set ARP to 6.
```

A caution for those accessing the circular buffer with 32-bit operations

If at least one of the instructions accessing your circular buffer performs a 32-bit operation, *make sure XAR6 and AR1 are both even before the buffer is accessed.* This ensures that AR6(7:0) and AR1(7:0) will match at the bottom of the buffer every time XAR6 cycles through the buffer. As an example of the problems that could arise if XAR6 and AR1 are not both even, consider this scenario:

- ❏ XAR6 is loaded with the even address 01 $FF00_{16}$ and AR1 is loaded with the odd value 00 0007_{16}. The value in AR1 indicates that the bottom of the buffer is at address 01 $FF07_{16}$.
- ❏ The next six instructions perform 16-bit operations, and each causes AR6 to increment by 1. At this point, XAR6 points to 01 $FF06_{16}$.
- ❏ Then a seventh instruction performs a 32-bit operation and increments AR6 by 2. Now XAR6 points to 01 $FF08_{16}$, which is past the bottom of the buffer. Because AR6(7:0) and AR1(7:0) never match, XAR6 never gets returned to the top of the buffer.

5.6 Summary of Indirect-Addressing Operands

The following table summarizes the operands for all indirect addressing modes except XAR7 indirect addressing mode. XAR7 indirect addressing mode is only used in four instructions (see section 5.5.3).

Operand	Description	Example

*SP++	Increment SP *after* the data-memory access.
	If (16-bit operation)
	SP = SP +1
	If (32-bit operation)
	SP = SP +2
		MOV *SP++, AH loads the high word of the accumulator to the 16-bit location pointed to by SP and increments the content of SP. Because the load is a 16-bit operation, SP is incremented by 1.

*––SP	Decrement SP *before* the data-memory access.
	If (16-bit operation)
	SP = SP – 1
	If (32-bit operation)
	SP = SP – 2
		MOVL ACC, *––SP decrements the content of SP and loads ACC with the 32-bit value referenced by SP. Because the load is a 32-bit operation, SP is decremented by 2.

*–SP[*6bit*]	Use the address 0:(SP – *6bit*) for the data-memory access. Do not modify SP. This operand is only available when the PAGE0 bit in ST1 is 0.	**SUB ACC, *–SP[4]** subtracts the value at address 0:(SP – 4) from ACC. SP is not modified.

*ARx++
or
*XARn++
	Increment the specified auxiliary register *after* the data-memory access. Also, set ARP to x or n.
	If (16-bit operation)
	ARx/XARn = ARx/XARn + 1
	If (32-bit operation)
	ARx/XARn = ARx/XARn + 2
		MOV *AR2++, AH loads the high word of the accumulator to the address 0:AR2, increments the content of AR2 by 1, and sets ARP = 2.

*––ARx
or
*––XARn
	Decrement the specified auxiliary register *before* the data-memory access. Also, set ARP to x or n.
	If (16-bit operation)
	ARx/XARn = ARx/XARn – 1
	If (32-bit operation)
	ARx/XARn = ARx/XARn – 2
		MOV *––XAR6, AH decrements the content of XAR6, loads the high word of the accumulator to the address pointed to by XAR6, and sets ARP = 6.

Notes: 1) ARx = AR0, AR1, AR2, AR3, AR4, or AR5

2) XARn = XAR6 or XAR7

3) *3bit* is a 3-bit constant (a value from 0 to 7).

4) The notation 0:ARx or 0:(ARx + n) indicates that when a 16-bit auxiliary register is used, the six MSBs of the data-memory address are zeros.

Operand	Description	Example
*+ARx[AR0] or *+XARn[AR0]	Use the address 0:(ARx + AR0) or 0:(XARn + AR0) for the data-memory access. Also, set ARP to x or n. Do not modify ARx or XARn.	**MOV *+AR4[AR0], AH** loads the high word of the accumulator to the 22-bit address 0:(AR4 + AR0) and sets ARP = 4. AR4 is not modified.
*+ARx[AR1] or *+XARn[AR1]	Use the address 0:(ARx + AR1) or (XARn + AR1) for the data-memory access. Also, set ARP to x or n. Do not modify ARx or XARn.	**MOV *+XAR7[AR1], AH** loads the high word of the accumulator to the 22-bit address (XAR7 + AR1) and sets ARP = 7. XAR7 is not modified.
*+ARx[*3bit*] or *+XARn[*3bit*]	Use the address 0:(ARx + *3bit*) or (XARn + *3bit*) for the data-memory access. Also, set ARP to x or n. Do not modify ARx or XARn.	**MOV *+AR5[6], AH** loads the high word of the accumulator to the 16-bit location pointed to by 0:(AR5 + 6) and sets ARP = 5. AR5 is not modified.
*ARx or *XARn	Equivalent to *+ARx[0] or *+XARn[0]. Use the address in ARx or XARn for the data-memory access. Also, set ARP to x or n. Do not modify ARx or XARn.	**MOV *AR5, AH** loads the high word of the accumulator to the 16-bit location pointed to by AR5 and sets ARP = 5. AR5 is not modified.
*ARPx	Set ARP to x x Selects x Selects 0 AR0 4 AR4 1 AR1 5 AR5 2 AR2 6 XAR6 3 AR3 7 XAR7	**MOV *ARP2, AH** loads the high word of the accumulator to the 16-bit location pointed to by the current auxiliary register and then sets ARP to 2.
*	No increment or decrement of current auxiliary register.	**MOV *, AH** loads the high word of the accumulator to the 16-bit location pointed to by the current auxiliary register and does nothing to the content of the current AR.

Notes: 1) ARx = AR0, AR1, AR2, AR3, AR4, or AR5

2) XARn = XAR6 or XAR7

3) *3bit* is a 3-bit constant (a value from 0 to 7).

4) The notation 0:ARx or 0:(ARx + n) indicates that when a 16-bit auxiliary register is used, the six MSBs of the data-memory address are zeros.

Operand	Description	Example

*++	Increment current auxiliary register *after* the data-memory access.
	
	If (16-bit operation)
	ARx/XARn = ARx/XARn + 1
	If (32-bit operation)
	ARx/XARn = ARx/XARn + 2
		MOV *++, AH loads the high word of the accumulator to the 16-bit location pointed to by the current auxiliary register and increments the content of the current AR by 1.

*––	Decrement current auxiliary register *after* the data-memory access.
	
	If (16-bit operation)
	ARx/XARn = ARx/XARn – 1
	If (32-bit operation)
	ARx/XARn = ARx/XARn – 2
		MOV *––, AH loads the high word of the accumulator to the 16-bit location pointed to by the current auxiliary register and decrements the content of the current AR by 1.

*0++	Increment current auxiliary register by index amount *after* the data-memory access. The index amount is the value in AR0.	**MOV *0++, AH** loads the high word of the accumulator to the 16-bit location pointed to by the current auxiliary register and adds the content of AR0 to the content of the current auxiliary register.
*0––	Decrement current auxiliary register by index amount *after* the data-memory access. The index amount is the value in AR0.	**MOV *0––, AH** loads the high word of the accumulator to the 16-bit location pointed to by the current auxiliary register and subtracts the content of AR0 from the content of the current auxiliary register.

*AR6%++	*After* the data-memory access, if AR6(7:0) equals AR1(7:0), clear AR6(7:0). Otherwise increment AR6.
	
	If (16-bit operation)
	AR6 = AR6 +1
	If (32-bit operation)
	AR6 = AR6 +2
		ADDL ACC, *AR6%++ reads the 32-bit value at the addresses referenced by XAR6, adds that value to ACC, and sets ARP to 6. If AR6(7:0) equals AR1(7:0), AR6(7:0) is cleared. Otherwise, AR6 is incremented by 2.

Notes:
1) ARx = AR0, AR1, AR2, AR3, AR4, or AR5
2) XARn = XAR6 or XAR7
3) *3bit* is a 3-bit constant (a value from 0 to 7).
4) The notation 0:ARx or 0:(ARx + n) indicates that when a 16-bit auxiliary register is used, the six MSBs of the data-memory address are zeros.

5.7 Addressing-Mode Information in Opcodes

This section explains how each of the addressing modes is encoded in the instruction opcodes of instructions. It directly supports Chapter 6, *Assembly Language Instructions*. For example, one of the possible opcodes for the ADD instruction is shown in Chapter 6 as follows:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	See section 5.7 on page 5-21.							
16BitSigned															

The second word (*16BitSigned*) is an embedded constant from an immediate addressing mode. Bits 7 through 0 of the first word encode information about a register-, direct-, or indirect-addressing operand.

When a 2-word opcode is stored in program memory, the least significant word must be stored first.

5.7.1 Opcodes for Immediate Addressing Modes

When you use an immediate addressing mode, the constant you enter is embedded directly in the instruction opcode. If the instruction requires a constant with a range of fewer than 16 bits, only one word is needed for the opcode, and the constant is embedded in the lower portion of the opcode. Consider Example 5–2 and Example 5–3, which both use immediate-constant addressing mode. In Example 5–2 , an 8-bit constant is embedded in bits 7 through 0 of the opcode. Bits 15 through 8 contain the code for the chosen syntax of the ADDB instruction. In Example 5–3 , a 5-bit constant is embedded in bits 4 through 0, and bits 15 through 5 contain the code for the single syntax of the TRAP instruction.

Example 5–2. Opcode Format for Immediate Addressing With an 8-Bit Constant

Syntax: **ADDB ACC,** #*8bit*

Instruction:
```
ADDB ACC, #15     ; Add 15 to ACC.
```

Opcode:

15–8								7–0							
0	0	0	0	1	0	0	1	0	0	0	0	1	1	1	1
Code for this ADDB syntax								8-bit constant = 15							

Example 5–3. Opcode Format for Immediate Addressing With a 5-Bit Constant

Syntax: **TRAP #***VectorNumber*

Instruction:
```
TRAP #4     ; Initiate interrupt INT4.
```

Opcode:

	15–5										4–0				
	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0
	Code for TRAP										5-bit constant = 4				

When an instruction uses a constant of 16 or more bits, the instruction requires two words for its opcode. This is shown in Example 5–4 and Example 5–5. In Example 5–4, the first word of the opcode contains the opcode for the MOVW instruction, and the second word contains the 16-bit constant. Example 5–5 uses a 22-bit constant. The 6 most significant bits of the constant are embedded in bits 5 through 0 of the first word of the opcode. The 16 least significant bits of the constant form the second word.

Example 5–4. Opcode Format for Immediate Addressing With a 16-Bit Constant

Syntax: **MOVW DP,** #*16bit*

Instruction:
```
MOVW DP, #6   ; Load all 16 bits of DP with 6.
```

Opcode:

	15–0														
1st word	0	1	1	1	0	1	1	0	0	0	0	1	1	1	1	1
	Code for MOVW														

	15–0														
2nd word	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0
	16-bit constant = 6														

Example 5–5. Opcode Format for Immediate Addressing With a 22-Bit Constant

Syntax: **MOV XAR***n***,** #*22bit*

Instruction:
```
MOV XAR6, #0FFFFFh  ; Load all 22 bits of XAR6
                    ; with hexadecimal 0FFFFF.
```

Opcode:

	15–6										5–0					
1st word	0	1	1	1	0	1	1	0	1	0	0	0	1	1	1	1
	Code for this MOV syntax, using XAR6										Bits 21–16 of constant					

	15–0														
2nd word	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
	Bits 15–0 of constant														

5.7.2 Opcodes for Register, Direct, and Indirect Addressing Modes

Many instructions use register addressing mode or one of the direct or indirect addressing modes. For these instructions, the addressing-mode information is encoded in bits 7 through 0 of the first or only word of the opcode. Table 5–5 summarizes how these eight bits are filled for each of the modes.

Table 5–5. Encoding of Register-, Direct-, and Indirect-Addressing Operands

Addressing Mode	Bits 7–0 of Opcode	Operand	Comments
DP direct	0 0 CCC CCC	*@6bit*	The 6-bit constant given in the operand is placed in the opcode where the six Cs are shown.
PAGE0 direct	0 1 CCC CCC	@0:*6bit*	The 6-bit constant given in the operand is placed in the opcode where the six Cs are shown. PAGE0 direct addressing mode and PAGE0 stack addressing mode are encoded the same way because only one can be active at a time (selected by the PAGE0 bit).
PAGE0 stack	0 1 CCC CCC	*–SP[*6bit*]	The 6-bit offset given in the operand is placed in the opcode where the six Cs are shown. PAGE0 direct addressing mode and PAGE0 stack addressing mode are encoded the same way because only one can be active at a time (selected by the PAGE0 bit).
Auxiliary-register indirect	1 0 000 AAA	*AR*x*++ or XAR*n*++	In the operands, *x* is a number from 0 to 5, *n* is 6 or 7, and *3bit* is a 3-bit constant. In the opcode, three As indicate where the 3-bit auxiliary register number (from 0 to 7) is embedded. Three Cs indicate where the 3-bit constant is embedded.
	1 0 001 AAA	*––AR*x* or *––XAR*n*	
	1 0 010 AAA	*+AR*x*[AR0] or *+XAR*n*[AR0]	
	1 0 011 AAA	*+AR*x*[AR1] or *+XAR*n*[AR1]	
	1 1 CCC AAA	* +AR*x*[*3bit*] or *+XAR*n*[*3bit*]	

Table 5–5. Encoding of Register-, Direct-, and Indirect-Addressing Operands (Continued)

Addressing Mode	Bits 7–0 of Opcode	Operand	Comments
Register	1 0 100 AAA	@AR*x* or @XAR*n*	In the operands, *x* is a number from 0 to 7, and *n* is 6 or 7. The three As in the first register-addressing code indicate where the 3-bit auxiliary register number (from 0 to 7) is embedded.
	1 0 101 000	@AH	
	1 0 101 001	@AL	
	1 0 101 010	@PH	
	1 0 101 011	@PL	
	1 0 101 100	@T	
	1 0 101 101	@SP	
ARP indirect	1 0 110 AAA	*ARP*z*	The variable *z* is a 3-bit number from 0 to 7, indicating which auxiliary register is selected. This 3-bit number is embedded in the opcode where the three As are shown.
	1 0 111 000	*	
	1 0 111 001	*++	
	1 0 111 010	*––	
	1 0 111 011	*0++	
	1 0 111 100	*0––	
Stack-pointer indirect	1 0 111 101	*SP++	
	1 0 111 110	*––SP	
Circular	1 0 111 111	*AR6%++	

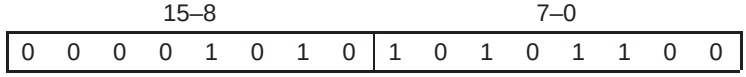
Example 5–6 through Example 5–9 show how bits 7 through 0 of opcodes are filled for particular cases of register, direct, and indirect addressing modes. The instruction in Example 5–6 uses register addressing to access the T register. This is indicated by the code in bits 7–0 of the opcode. You can verify this code by checking Table 5–5.

Example 5–6. Opcode Format for an Instruction Using Register Addressing Mode

Syntax: **INC** *loc*

Instruction:
```
INC @T              ; Increment content of
                    ; T register by 1.
```

Opcode:



Code for INC

Code for accessing T by way of register addressing mode

In Example 5–7 bits 7 and 6 of the opcode are both 0, indicating DP direct addressing mode. Bits 5 through 0 contain the offset of 5.

Example 5–7. Opcode Format for an Instruction Using DP Direct Addressing Mode

Syntax: **ADD ACC,** *loc {***<<** *shift3}*

Instruction:
```
ADD ACC, @5 << 4   ; Data page is 2 (DP = 2).
                   ; Copy content of hexadecimal
                   ; address 000085, shift it
                   ; left by 4, and add result
                   ; to ACC.
```

Opcode:

15–12				11–8				7–0							
1	0	1	0	0	1	0	0	0	0	0	0	0	1	0	1
Code for this ADD syntax				Shift of 4				Code for DP direct addressing mode, offset of 5							

Example 5–8 uses auxiliary-register indirect addressing to access data-memory. This is indicated by bits 7 through 0 of the first word of the opcode. Because XAR6 is used, bits 2 through 0 contain 6 (110_2). The example also uses immediate-pointer addressing to access program memory. The constant given in the third operand (5) indicates the 16 LSBs of address 00 0005_{16}. This 16-bit constant forms the second word of the opcode.

Example 5–8. Opcode Format for an Instruction Using Auxiliary-Register Indirect Addressing Mode

Syntax: **MAC P,** *loc,* **0:***pmem*

Instruction:
```
MAC P, *XAR6++, 0:5 ; XAR6 = 0F0000
                    ; Add content of P register
                    ; to accumulator. Multiply
                    ; content of data–memory
                    ; address 0F0000 by content
                    ; of program–memory address
                    ; 000005. Place result in
                    ; P register. Increment XAR6
                    ; by 1. Set ARP = 6.
```

Opcode:

	15–8								7–0							
1st word	0	0	0	1	0	1	0	0	1	0	0	0	0	1	1	0
	Code for MAC								Code for operand *XAR6++							

	15–0															
2nd word	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1

16 LSBs of program-memory address= 0005_{16}

The instruction in Example 5–9 uses stack-pointer indirect addressing to point to a location in data memory. In addition, it uses immediate-constant addressing mode to indicate a 16-bit signed constant. This constant forms the second word of the opcode.

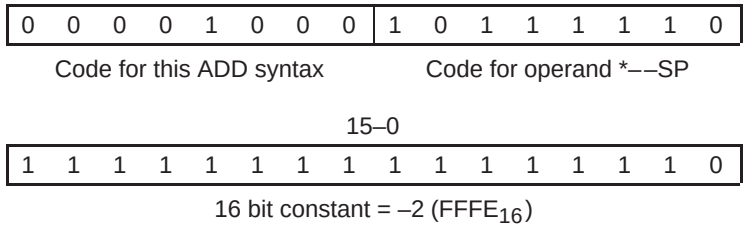
Example 5–9. Opcode Format for an Instruction Using Stack-Pointer Indirect Addressing Mode

Syntax: **ADD** *loc, #16BitSigned*

Instruction:

```
ADD *--SP, #-2      ; SP = 0801
                    ; Decrement SP by 1. Add -2
                    ; to content of data-memory
                    ; address 0800.
```

Opcode:



Chapter 6

Assembly Language Instructions

This chapter presents summaries of the instruction set, explains how some instructions are affected by alignment of 32-bit accesses to even addresses, defines special symbols and notations used, and describes each instruction in detail in alphabetical order. The number of cycles needed for each instruction is included in the tables in section 6.1.2 (page 6-9).

For more information about the registers mentioned throughout this chapter, see section 2.2, *CPU Registers*, on page 2-4.

Topic **Page**

6.1 Instruction Set Summaries 6-2
6.2 Alignment of 32-Bit Accesses to Even Addresses 6-31
6.3 How to Use the Instruction Descriptions 6-34
6.4 Instruction Descriptions 6-39

6.1 Instruction Set Summaries

The ’27x uses the following syntax formats. When a destination is specified, it is always the leftmost operand.

Format	Example
Mnemonic destination, source1, source2	`AND AH,@7,#16 ; Read 2 source values. ; Write logical AND of ; those values to destination.`
Mnemonic destination, source 1	`MOV ACC,@7 ; Read from source. Write ; value to destination.`
Mnemonic source1, source2	`CMPL ACC,@7 ; Compare two source values.`
Mnemonic source	`PUSH ST0 ; Read from source. Stack is ; destination.`
Mnemonic destination	`POP ST1 ; Source is stack. Write to ; destination.`

This section provides summaries of the instruction set as follows:

Summary Type	See ...	Comment
Alphabetical by mnemonic	Section 6.1.1 on page 6-2	This summary includes brief descriptions of what the instructions do.
By operation type	Section 6.1.2 on page 6-9	This summary includes information about instruction sizes and cycle times. It also includes status bit information.
By opcode	Section 6.1.3 on page 6-20	This summary can be used to match an opcode to its instruction syntax.

6.1.1 Alphabetical Summary by Mnemonic

Table 6–1 lists the instructions in alphabetical order by mnemonic, and gives a brief description of the instruction and comments about the instruction. The rightmost column of each table row lists the page where you can find more details about an instruction.

Table 6–1. Alphabetical Instruction Set Summary

Mnemonic	Description	Comments	Page
ABORTI	Abort interrupt	Used when an interrupt will not be returned from in the normal manner	6-40
ABS	Absolute value of accumulator	Finds the absolute value of the full 32-bit accumulator	6-41
ADD	Add value to specified location	Adds a 16-bit value to a data-memory location or a register. This value can be from data-memory or a register, or can be a constant.	6-43
ADDB	Add short value to specified register	Adds a 7-bit constant to an auxiliary register or the stack pointer, or adds an 8-bit constant to AH, AL, or ACC	6-51
ADDCU	Add unsigned value plus carry bit to accumulator	Unsigned value is a 16-bit data-memory or register value	6-55
ADDL	Add long value to accumulator	Adds a 32-bit data-memory value or a 22-bit auxiliary register value to ACC	6-58
ADDU	Add unsigned value to accumulator	Unsigned value is a 16-bit data-memory or register value	6-62
ADRK	Add to current auxiliary register	Adds an 8-bit constant to an auxiliary register	6-65
AND	Bitwise AND	Performs an AND operation on two 16-bit values Can be used to clear values in the interrupt enable register (IER) and the interrupt flag register (IFR)	6-66
ANDB	Bitwise AND with short value	Performs an AND operation on an 8-bit value and an accumulator half (AH or AL)	6-71
ASP	Align stack pointer	If the stack pointer (SP) points to an odd address, ASP increments SP by 1. This aligns SP to an even address.	6-72
ASR	Arithmetic shift right	Shifts AH or AL right by the amount you specify. During the shift, the value is sign extended.	6-74
B	Branch	Uses a 16-bit offset to perform an offset branch	6-77
BANZ	Branch if auxiliary register not equal to zero	Uses a 16-bit offset to perform an offset branch	6-80

Table 6–1. Alphabetical Instruction Set Summary (Continued)

Mnemonic	Description	Comments	Page
CALL	Call	Stores the return address to the stack and then performs an absolute branch to a specified 22-bit address	6-82
CLRC	Clear status bits	Clears one or more of certain bits in status registers ST0 and ST1	6-84
CMP	Compare	Uses a subtraction to compare two values. The result is not stored but is reflected by flag bits.	6-87
CMPB	Compare with short value	Uses a subtraction to compare AH or AL with an 8-bit constant. The result is not stored but is reflected by flag bits.	6-91
CMPL	Compare with long value	Uses a subtraction to compare ACC with a 32-bit data-memory value or a 22-bit auxiliary register value. The result is not stored but is reflected by flag bits.	6-93
DEC	Decrement specified value by 1	Acts on a 16-bit data-memory location or register	6-96
EALLOW	Allow access to emulation registers	Enables access to the memory-mapped emulation registers	6-98
EDIS	Disallow access to emulation registers	Disables access to the memory-mapped emulation registers	6-99
ESTOP0	Emulator software breakpoint	Used to create a software breakpoint	6-100
ESTOP1	Embedded software breakpoint	Used to create an embedded software breakpoint	6-101
FFC	Fast function call	Stores the return address to auxiliary register XAR7 and then performs an absolute branch to a specified 22-bit address	6-102
IACK	Interrupt acknowledge	Drives a specified 16-bit value on the low 16 bits of the data-write data bus, DWDB(15:0). Can be used in an interrupt service routine to inform external hardware that a certain interrupt is being serviced by the CPU.	6-103
IDLE	Idle until interrupt	Places CPU in a dormant state until it is awakened by an enabled or nonmaskable hardware interrupt	6-104

Table 6–1. Alphabetical Instruction Set Summary (Continued)

Mnemonic	Description	Comments	Page
INC	Increment specified value by 1	Acts on a 16-bit data-memory location or register	6-106
INTR	Software interrupt	Can be used to initiate maskable interrupts $\overline{\text{INT1}}$–$\overline{\text{INT14}}$, DLOGINT, RTOSINT, and NMI. For those interrupts with bits in the interrupt enable register (IER) and the interrupt flag register (IER), the bits are cleared.	6-108
IRET	Return from interrupt and restore register pairs	Restores the program counter (PC) value and other register values that were saved to the stack by an interrupt operation	6-113
ITRAP0	Instruction trap 0	Causes an illegal-instruction trap, as with a TRAP #19 instruction	6-114
ITRAP1	Instruction trap 1	Causes an illegal-instruction trap, as with a TRAP #19 instruction	6-116
LB	Long branch	Performs an absolute branch to a specified 22-bit address	6-118
LOOPNZ	Loop while not zero	Repeats until a specified test results in 0	6-119
LOOPZ	Loop while zero	Repeats until a specified test results in a nonzero value	6-122
LSL	Logical shift left	Shifts AH, AL, or ACC left by the amount you specify	6-125
LSR	Logical shift right	Shifts AH or AL right by the amount you specify. During the shift, the value is not sign extended.	6-129
MAC	Multiply and accumulate with preload to T register	Adds the P register value to ACC, loads T register and then multiplies T register value by value from program memory	6-132
MOV	Move value	Copies a value from one location to another, or loads a location with a specified value	6-137
MOVA	Load T register and add previous product to accumulator	Loads the T register and adds the P register value to ACC	6-154
MOVB	Move short value	Loads a register with an 8-bit constant or moves a specified byte from one location to another	6-158

Table 6–1. Alphabetical Instruction Set Summary (Continued)

Mnemonic	Description	Comments	Page
MOVH	Store high word	Stores the high word of the P register or ACC to data-memory or to a register	6-168
MOVL	Move long value	Enables loads and stores that use ACC and 32-bit data-memory locations	6-172
MOVP	Load T register and load previous product to accumulator	Loads the T register and stores the P register value to ACC	6-176
MOVS	Load T register and subtract previous product from accumulator	Loads the T register and subtracts the P register value from ACC	6-179
MOVU	Load accumulator with unsigned word	Loads AL with an unsigned 16-bit value and clears AH	6-183
MOVW	Load entire data page pointer	Loads the entire data page pointer (DP) with a 16-bit constant. One syntax of the MOV instruction enables you to load only the 10 LSBs of the DP.	6-185
MPY	Multiply	Multiplies a 16-bit value by another 16-bit value	6-186
MPYA	Multiply and accumulate previous product	Adds the P register value to ACC and then multiplies two 16-bit values	6-190
MPYB	Multiply signed value by unsigned short value	Multiplies a signed 16-bit value by an unsigned 8-bit value	6-195
MPYS	Multiply and subtract previous product	Subtracts the P register value from ACC and then multiplies a 16-bit value by another 16-bit value	6-197
MPYU	Unsigned multiply	Multiplies an unsigned 16-bit value by another unsigned 16-bit value	6-201
MPYXU	Multiply signed value by unsigned value	Multiplies a signed 16-bit value by an unsigned 16-bit value	6-204
NASP	Unalign stack pointer	Undoes a previous alignment of SP performed by the ASP instruction.	6-207
NEG	Negative of accumulator value	Finds the negative of the value in AH, AL, or ACC	6-208
NOP	No operation	Can be used to purposely create inactive cycles. Can also be used to increment an auxiliary register or the stack pointer without performing any other task.	6-212

Table 6–1. Alphabetical Instruction Set Summary (Continued)

Mnemonic	Description	Comments	Page
NORM	Normalize accumulator	Can be used to remove extra sign bits from a value in ACC	6-214
NOT	Complement of accumulator value	Finds complement of AH, AL, or ACC	6-217
OR	Bitwise OR	Performs an OR operation on two 16-bit values Can be used to set values in the interrupt enable register (IER) and the interrupt flag register (IFR). If the AND instruction sets an IFR and the interrupt is enabled, the interrupt is serviced.	6-219
ORB	Bitwise OR with short value	Performs an OR operation on an 8-bit value and an accumulator half (AH or AL)	6-223
POP	Restore from stack	Copies a 16-bit value or a 32-bit register pair from the stack to a data-memory location or a register	6-224
PREAD	Read from program memory	Loads a 16-bit data-memory location or register with a value from program memory	6-233
PUSH	Save value on stack	Copies a 16-bit value or a 32-bit register pair to the stack	6-236
PWRITE	Write to program memory	Loads a program-memory location with a value from a 16-bit data-memory location or register	6-244
RET	Return	Loads the program counter (PC) from the stack	6-247
RETE	Return with interrupts enabled	Loads the program counter (PC) from the stack and clears the interrupt global mask bit (INTM). By clearing INTM, it enables maskable interrupts.	6-248
ROL	Rotate accumulator left	Can be seen as the left rotation of a 33-bit value that is the concatenation of the carry bit (C) and ACC.	6-249
ROR	Rotate accumulator right	Can be seen as the right rotation of a 33-bit value that is the concatenation of the carry bit (C) and ACC.	6-250
RPT	Repeat next instruction	Causes the following instruction to repeat a specified number of times	6-251

Table 6–1. Alphabetical Instruction Set Summary (Continued)

Mnemonic	Description	Comments	Page
SAT	Saturate accumulator	Fills ACC with a saturation value that reflects the net overflow represented in the overflow counter (OVC)	6-254
SB	Short branch	Uses an 8-bit offset to perform an offset branch	6-257
SBBU	Subtract unsigned value plus inverse borrow from accumulator	Unsigned value is a 16-bit data-memory or register value	6-260
SBRK	Subtract from specified auxiliary register	Subtracts an 8-bit constant from an auxiliary register	6-263
SETC	Set status bits	Sets one or more of specified bits in status registers ST0 and ST1	6-264
SFR	Shift accumulator right	Shifts ACC right by the amount you specify. During the shift, the value is sign extended if the sign-extension mode bit (SXM) is 1 or is not sign extended if SXM is 0.	6-267
SPM	Set product shift mode	Sets the product shift mode bits (PM), which determine how certain instructions shift the P register value	6-271
SUB	Subtract value from specified location	Subtracts a 16-bit value from a data-memory location or a register. This value can be from data-memory or a register, or can be a constant.	6-273
SUBB	Subtract short value from specified register	Subtracts a 7-bit constant from an auxiliary register or the stack pointer, or subtracts an 8-bit constant from ACC	6-280
SUBCU	Conditional subtraction	Used for 16-bit division	6-284
SUBL	Subtract long value from accumulator	Subtracts a 32-bit data-memory value or a 22-bit auxiliary register value from ACC	6-288
SUBU	Subtract unsigned value from accumulator	Unsigned value is a 16-bit data-memory or register value	6-292
SXTB	Sign extend least significant byte of accumulator half	Sign extends the least significant byte of AH or AL	6-296
TBIT	Test specified bit	Tests a specified bit of a 16-bit data-memory location or register. The value of the bit is reflected by the test/control flag bit (TC).	6-298

Table 6–1. Alphabetical Instruction Set Summary (Continued)

Mnemonic	Description	Comments	Page
TEST	Test for accumulator equal to zero	Uses a subtraction to compare ACC with 0. The result is not stored, but is reflected by flag bits.	6-301
TRAP	Software trap	Can be used to initiate any interrupt. Unlike the INTR instruction, the TRAP instruction never affects bits in the IER and IFR.	6-303
XOR	Bitwise exclusive OR	Performs an exclusive OR operation on two 16-bit values	6-307
XORB	Bitwise exclusive OR with short value	Performs an exclusive OR operation on an 8-bit value and an accumulator half (AH or AL)	6-310

6.1.2 Summary by Operation Type

Table 6–2 through Table 6–13 provide a summary of the instruction syntaxes according to the following functional groups:

- ❑ Address register operations (see Table 6–2)
- ❑ Push and pop stack operations (see Table 6–3 on page 6-10)
- ❑ AX (AH, AL) operations (see Table 6–4 on page 6-12)
- ❑ AX (AH, AL) byte operations (see Table 6–5 on page 6-13)
- ❑ ACC operations (see Table 6–6 on page 6-13)
- ❑ ACC 32-Bit operations (see Table 6–7 on page 6-15)
- ❑ Operations on memory or register (see Table 6–8 on page 6-15)
- ❑ Data move operations (see Table 6–9 on page 6-16)
- ❑ Program flow operations (see Table 6–10 on page 6-16)
- ❑ Math operations (see Table 6–11 on page 6-17)
- ❑ Control operations (see Table 6–12 on page 6-18)
- ❑ Emulation operations (see Table 6–13 on page 6-20)

The cycle times shown are the minimum cycle times. The rightmost column of each table row lists the page where you can find more details about an instruction syntax.

Table 6–2. Address Register Operations (AR0–AR7, XAR6, XAR7, DP, SP)

Instruction Syntax		Words	Cycles	Status Bits		Page
				Affected By	Affects	
ADDB	*aux*, *#7bit*	1	1	–	–	6-51
ADDB	SP, *#7bit*	1	1	–	–	6-51
ADRK	*#8bit*	1	1	ARP	–	6-65
MOV	AR*x*, *loc*	1	1	–	–	6-137
MOV	XAR*n*, *locLong*	1	1	–	–	6-137
MOV	XAR*n*, *#22bit*	2	1	–	–	6-137
MOV	DP, *#10bit*	1	1	–	–	6-137
MOV	*loc*, AR*x*	1	1	–	N, Z	6-137
MOV	*locLong*, XAR*n*	1	1	–	–	6-137
MOVB	AR*x*, *#8bit*	1	1	–	–	6-158
MOVW	DP, *#16bit*	2	1	–	–	6-185
SBRK	*#8bit*	1	1	ARP	–	6-263
SUBB	*aux*, *#7bit*	1	1	–	–	6-280
SUBB	SP, *#7bit*	1	1	–	–	6-280

Table 6–3. Push and Pop Stack Operations

Instruction Syntax		Words	Cycles	Status Bits		Page
				Affected By	Affects	
POP	*loc*	1	2	–	N, Z	6-224
POP	DBGIER	1	5	–	–	6-224
POP	DP	1	1	–	–	6-224
POP	ST0	1	1	–	C, N, OVM, OVC, PM, SXM, TC, V, Z	6-224
POP	ST1	1	5	–	ARP, DBGM, EALLOW, INTM, PAGE0, SPA, VMAP	6-224
POP	T:ST0	1	1	–	C, N, OVM, OVC, PM, SXM, TC, V, Z	6-224

Table 6–3. Push and Pop Stack Operations (Continued)

Instruction Syntax		Words	Cycles	Status Bits		Page
				Affected By	Affects	
POP	DP:ST1	1	5	–	ARP, DBGM, EALLOW, INTM, PAGE0, SPA, VMAP	6-224
POP	PH:PL	1	1	–	–	6-224
POP	AR1:AR0	1	1	–	–	6-224
POP	AR3:AR2	1	1	–	–	6-224
POP	AR5:AR4	1	1	–	–	6-224
POP	XAR*n*	1	1	–	–	6-224
PUSH	*loc*	1	2	–	–	6-236
PUSH	DBGIER	1	1	–	–	6-236
PUSH	DP	1	1	–	–	6-236
PUSH	IFR	1	1	–	–	6-236
PUSH	ST0	1	1	–	–	6-236
PUSH	ST1	1	1	–	–	6-236
PUSH	T:ST0	1	1	–	–	6-236
PUSH	DP:ST1	1	1	–	–	6-236
PUSH	PH:PL	1	1	–	–	6-236
PUSH	AR1:AR0	1	1	–	–	6-236
PUSH	AR3:AR2	1	1	–	–	6-236
PUSH	AR5:AR4	1	1	–	–	6-236
PUSH	XAR*n*	1	1	–	–	6-236

Table 6–4. AX (AH, AL) Operations

Instruction Syntax		Words	Cycles	Status Bits		Page
				Affected By	Affects	
ADD	A*X*, *loc*	1	1	–	C, N, V, Z	6-43
ADDB	A*X*, *#8BitSigned*	1	1	–	C, N, V, Z	6-51
AND	A*X*, *loc*	1	1	–	N, Z	6-66
AND	A*X*, *loc*, *#16BitMask*	2	1	–	N, Z	6-66
ANDB	A*X*, *#8BitMask*	1	1	–	N, Z	6-71
ASR	A*X*, *shift*	1	1	–	C, N, Z	6-74
ASR	A*X*, T	1	1	–	C, N, Z	6-74
CMP	A*X*, *loc*	1	1	–	C, N, Z	6-87
CMPB	A*X*, *#8bit*	1	1	–	C, N, Z	6-91
LSL	A*X*, *shift*	1	1	–	C, N, Z	6-125
LSL	A*X*, T	1	1	–	C, N, Z	6-125
LSR	A*X*, *shift*	1	1	–	C, N, Z	6-129
LSR	A*X*, T	1	1	–	C, N, Z	6-129
MOV	A*X*, *loc*	1	1	–	N, Z	6-137
MOV	*loc*, A*X*	1	1	–	N, Z	6-137
MOVB	A*X*, *#8bit*	1	1	–	N, Z	6-158
NEG	A*X*	1	1	–	C, N, V, Z	6-208
NOT	A*X*	1	1	–	N, Z	6-217
OR	A*X*, *loc*	1	1	–	N, Z	6-219
ORB	A*X*, *#8BitMask*	1	1	–	N, Z	6-223
SUB	A*X*, *loc*	1	1	–	C, N, V, Z	6-273
XOR	A*X*, *loc*	1	1	–	N, Z	6-307
XORB	A*X*, *#8BitMask*	1	1	–	N, Z	6-310

Table 6–5. AX (AH, AL) Byte Operations

Instruction Syntax		Words	Cycles	Status Bits		Page
				Affected By	Affects	
MOVB	A*X*.LSB, *loc*	1	1	–	N, Z	6-158
MOVB	*loc*, A*X*.LSB	1	1	–	N, Z	6-158
MOVB	A*X*.MSB, *loc*	1	1	–	N, Z	6-158
MOVB	*loc*, A*X*.MSB	1	1	–	N, Z	6-158
SXTB	A*X*	1	1	–	N, Z	6-296

Table 6–6. ACC Operations

Instruction Syntax		Words	Cycles	Status Bits		Page
				Affected By	Affects	
ABS	ACC	1	1	OVM	C, N, V, Z	6-41
ADD	ACC, *loc* << *shift*	1	1	OVM, SXM	C, N, V, Z	6-43
ADD	ACC, #*16BitSU* << *shift*	2	1	OVM, SXM	C, N, V, Z	6-43
ADD	ACC, P	1	1	OVM, PM	C, N, V, Z	6-43
ADDB	ACC, #*8bit*	1	1	OVM	C, N, V, Z	6-51
ADDCU	ACC, *loc*	1	1	OVM	C, N, V, Z	6-55
ADDU	ACC, *loc*	1	1	OVM	C, N, V, Z	6-62
LSL	ACC, *shift*	1	1	–	C, N, Z	6-125
LSL	ACC, T	1	1	–	C, N, Z	6-125
MOV	ACC, *loc* << *shift*	1	1	SXM	N, Z	6-137
MOV	ACC, #*16BitSU* << *shift*	2	1	SXM	N, Z	6-137
MOV	ACC, P	1	1	PM	N, Z	6-137
MOV	*loc*, ACC << *shift*	1	1	–	N, Z	6-137
MOVB	ACC, #*8bit*	1	1	–	N, Z	6-158
MOVH	*loc*, ACC << *shift*	1	1	–	N, Z	6-168
MOVU	ACC, *loc*	1	1	–	N, Z	6-183

† This instruction is repeatable. The number of cycles given is for nonrepeated execution.

Table 6–6. ACC Operations (Continued)

Instruction Syntax		Words	Cycles	Status Bits		Page
				Affected By	Affects	
NEG	ACC	1	1	OVM	C, N, V, Z	6-208
NORM	ACC, *aux*++	1	4†	–	N, TC, Z	6-214
NORM	ACC, *aux*––	1	4†	–	N, TC, Z	6-214
NOT	ACC	1	1	–	N, Z	6-217
ROL	ACC	1	1†	–	C, N, Z	6-249
ROR	ACC	1	1†	–	C, N, Z	6-250
SAT	ACC	1	1	OVC	C, N, OVC, V, Z	6-254
SBBU	ACC, *loc*	1	1	OVM	C, N, V, Z	6-260
SFR	ACC, *shift*	1	1	SXM	C, N, Z	6-267
SFR	ACC, T	1	1	SXM	C, N, Z	6-267
SUB	ACC, *loc* << *shift*	1	1	OVM, SXM	C, N, V, Z	6-273
SUB	ACC, #*16BitSU* << *shift*	2	1	OVM, SXM	C, N, V, Z	6-273
SUB	ACC, P	1	1	OVM, PM	C, N, V, Z	6-273
SUBB	ACC, #*8bit*	1	1	OVM	C, N, V, Z	6-280
SUBCU	ACC, *loc*	1	1†	–	C, N, OVC, V, Z	6-284
SUBU	ACC, *loc*	1	1	OVM	C, N, V, Z	6-292
TEST	ACC	1	1	–	N, Z	6-301

† This instruction is repeatable. The number of cycles given is for nonrepeated execution.

Table 6–7. ACC 32-Bit Operations

Instruction Syntax		Words	Cycles	Status Bits		Page
				Affected By	Affects	
ADDL	ACC, *locLong*	1	1	OVM	C, N, V, Z	6-58
CMPL	ACC, *locLong*	1	1	–	C, N, Z	6-93
MOVL	ACC, *locLong*	1	1	–	N, Z	6-172
MOVL	*locLong*, ACC	1	1	–	–	6-172
SUBL	ACC, *locLong*	1	1	OVM	C, N, V, Z	6-288

Table 6–8. Operations on Memory or Register

Instruction Syntax		Words	Cycles	Status Bits		Page
				Affected By	Affects	
ADD	*loc*, A*X*	1	1	–	C, N, V, Z	6-43
ADD	*loc*, *#16BitSigned*	2	1	–	C, N, V, Z	6-43
AND	*loc*, A*X*	1	1	–	N, Z	6-66
AND	*loc*, *#16BitMask*	2	1	–	N, Z	6-66
CMP	*loc*, *#16BitSigned*	2	1	–	C, N, Z	6-87
DEC	*loc*	1	1	–	C, N, V, Z	6-96
INC	*loc*	1	1	–	C, N, V, Z	6-106
LOOPNZ	*loc*, *#16BitMask*	2	5	–	LOOP, N, Z	6-119
LOOPZ	*loc*, *#16BitMask*	2	5	–	LOOP, N, Z	6-122
OR	*loc*, A*X*	1	1	–	N, Z	6-219
OR	*loc*, *#16BitMask*	2	1	–	N, Z	6-219
SUB	*loc*, A*X*	1	1	–	C, N, V, Z	6-273
TBIT	*loc*, *#BitNumber*	1	1	–	TC	6-298
XOR	*loc*, A*X*	1	1	–	N, Z	6-307
XOR	*loc*, *#16BitMask*	2	1	–	N, Z	6-307

Table 6–9. Data Move Operations

Instruction	Syntax	Words	Cycles	Status Bits: Affected By	Status Bits: Affects	Page
MOV	*(0:*16bit*), *loc*	2	2†	–	–	6-137
MOV	*loc*, #0	1	1†	–	N, Z	6-137
MOV	*loc*, #*16bit*	2	1†	–	N, Z	6-137
MOV	*loc*, *(0:*16bit*)	2	2†	–	N, Z	6-137
PREAD	*loc*, *XAR7	1	2†	–	N, Z	6-233
PWRITE	*XAR7, *loc*	1	5†	–	–	6-244

† This instruction is repeatable. The number of cycles given is for nonrepeated execution.

Table 6–10. Program Flow Operations

Instruction	Syntax	Words	Cycles	Status Bits: Affected By	Status Bits: Affects	Page
B	*16BitOffset*, *cond*	2	4/7†	–	V (if tested)	6-77
BANZ	*16BitOffset*, AR*x*––	2	2/4†	–	–	6-80
CALL	*22BitAddress*	2	4	–	–	6-82
CALL	*XAR7	1	4	–	–	6-82
FFC	XAR7, *22BitAddress*	2	4	–	–	6-102
IRET		1	8	–	ARP, C, DBGM, INTM, N, OVC, OVM, PAGE0, PM, SPA, SXM, TC, V, VMAP, Z	6-113
LB	*22BitAddress*	2	4	–	–	6-118
LB	*XAR7	1	4	–	–	6-118
RET		1	8	–	–	6-247
RETE		1	8	–	INTM	6-248
SB	*8BitOffset*, *cond*	1	4/7†	–	V (if tested)	6-257

† X/Y: X cycles are required if the branch is not taken. Y cycles are required if the branch is taken.

Table 6–11. Math Operations

Instruction Syntax		Words	Cycles	Status Bits		Page
				Affected By	Affects	
CMP	ACC, P	1	1	PM	C, N, Z	6-87
MAC	P, *loc*, 0:*pmem*	2	2†	OVM, PM	C, N, V, Z	6-132
MOV	PH, *loc*	1	1	–	–	6-137
MOV	PL, *loc*	1	1	–	–	6-137
MOV	P, ACC	1	1	–	–	6-137
MOV	T, *loc*	1	1	–	–	6-137
MOV	*loc*, P	1	1	PM	N, Z	6-137
MOV	*loc*, T	1	1	–	N, Z	6-137
MOVA	T, *loc*	1	1	OVM, PM	C, N, V, Z	6-154
MOVH	*loc*, P	1	1	PM	N, Z	6-168
MOVP	T, *loc*	1	1	PM	N, Z	6-176
MOVS	T, *loc*	1	1	OVM, PM	C, N, V, Z	6-179
MPY	ACC, *loc*, *#16BitSigned*	2	1	–	N, Z	6-186
MPY	ACC, T, *loc*	1	1	–	N, Z	6-186
MPY	P, T, *loc*	1	1	–	–	6-186
MPYA	P, *loc*, *#16BitSigned*	2	1	OVM, PM	C, N, V, Z	6-190
MPYA	P, T, *loc*	1	1	OVM, PM	C, N, V, Z	6-190
MPYB	ACC, T, *#8bit*	1	1	–	N, Z	6-195
MPYB	P, T, *#8bit*	1	1	–	–	6-195
MPYS	P, T, *loc*	1	1	OVM, PM	C, N, V, Z	6-197
MPYU	ACC, T, *loc*	1	1	–	N, Z	6-201
MPYU	P, T, *loc*	1	1	–	–	6-201
MPYXU	ACC, T, *loc*	1	1	–	N, Z	6-204
MPYXU	P, T, *loc*	1	1	–	–	6-204

† This instruction is repeatable. The number of cycles given is for nonrepeated execution.

Table 6–12. Control Operations

Instruction Syntax		Words	Cycles	Status Bits		Page
				Affected By	Affects	
AND	IER, #*16BitMask*	2	2	–	–	6-66
AND	IFR, #*16BitMask*	2	2	–	–	6-66
ASP		1	1	SPA	SPA	6-72
CLRC	*BitName1* { , *BitName2*}	1	1 or 2†	–	Specified status bit(s): C, DBGM, INTM, OVM, PAGE0, SXM, TC, VMAP	6-84
CLRC	*8BitMask*	1	1 or 2†	–	Specified status bit(s): C, DBGM, INTM, OVM, PAGE0, SXM, TC, VMAP	6-84
IACK	#*VectorValue*	2	1	–	–	6-103
IDLE		1	5	–	IDLESTAT	6-104
INTR	INT*i*	1	8	–	DBGM, EALLOW, IDLESTAT, INTM, LOOP	6-108
INTR	DLOGINT	1	8	–	DBGM, EALLOW, IDLESTAT, INTM, LOOP	6-108
INTR	RTOSINT	1	8	–	DBGM, EALLOW, IDLESTAT, INTM, LOOP	6-108
INTR	NMI	1	8	–	DBGM, EALLOW, IDLESTAT, INTM, LOOP	6-108
MOV	IER, *loc*	1	5	–	–	6-137
MOV	*loc*, IER	1	1	–	–	6-137
NASP		1	1	SPA	SPA	6-207
NOP		1	1‡	–	–	6-212

† If CLRC or SETC is to affect the INTM bit and/or the DBGM bit, the instruction requires 2 cycles; otherwise the instruction requires 1 cycle.
‡ This instruction is repeatable. The number of cycles given is for nonrepeated execution.

Table 6–12. Control Operations (Continued)

Instruction Syntax		Words	Cycles	Status Bits		Page
				Affected By	Affects	
NOP	**ind*	1	1‡	–	–	6-212
OR	IER, #*16BitMask*	2	2	–	–	6-219
OR	IFR, #*16BitMask*	2	2	–	–	6-219
RPT	*loc*	1	4	–	–	6-251
RPT	#*8bit*	1	1	–	–	6-251
SETC	*BitName1* { , *BitName2*}	1	1 or 2†	–	Specified status bit(s): C, DBGM, INTM, OVM, PAGE0, SXM, TC, VMAP	6-264
SETC	*8BitMask*	1	1 or 2†	–	Specified status bit(s): C, DBGM, INTM, OVM, PAGE0, SXM, TC, VMAP	6-264
SPM	*ShiftMode*	1	1	–	PM	6-271
TRAP	#*VectorNumber*	1	8	–	DBGM, EALLOW, IDLESTAT, INTM, LOOP	6-303

† If CLRC or SETC is to affect the INTM bit and/or the DBGM bit, the instruction requires 2 cycles; otherwise the instruction requires 1 cycle.
‡ This instruction is repeatable. The number of cycles given is for nonrepeated execution.

Table 6–13. Emulation Operations

Instruction Syntax	Words	Cycles	Status Bits		Page
			Affected By	Affects	
ABORTI	1	2	–	DBGM	6-40
EALLOW	1	4	–	EALLOW	6-98
EDIS	1	4	–	EALLOW	6-99
ESTOP0	1	1	–	–	6-100
ESTOP1	1	1	–	–	6-101
ITRAP0	1	8	–	DBGM, EALLOW, IDLESTAT, INTM, LOOP	6-114
ITRAP1	1	8	–	DBGM, EALLOW, IDLESTAT, INTM, LOOP	6-116

6.1.3 Summary by Opcode

Table 6–14 lists the instruction set in numerical order by opcode. The rightmost column of each table row lists the page where you can find more details about an instruction. To find out how addressing-mode information is encoded in opcodes, see section 5.7 on page 5-21.

Table 6–14. Instruction Set Summary by Opcode

Opcode or Opcode Range (Hexadecimal)	Instruction Syntax	Page
0000	ITRAP0	6-114
0001	ABORTI	6-40
0008–000F for 1st word 2nd word is *16BitOffset*	BANZ *16BitOffset*, AR*x*––	6-80
0010–001F	INTR INT*i*, INTR DLOGINT, or INTR RTOSINT	6-108
0020–003F	TRAP #*VectorNumber*	6-303
0040–007F for 1st word 2nd word is *22BitAddress*(15:0)	LB *22BitAddress*	6-118

Table 6–14. Instruction Set Summary by Opcode (Continued)

Opcode or Opcode Range (Hexadecimal)	Instruction Syntax	Page
0080–00BF for 1st word 2nd word is *22BitAddress*(15:0)	CALL *22BitAddress*	6-82
00C0–00FF for 1st word 2nd word is *22BitAddress*(15:0)	FCC XAR7, *22BitAddress*	6-102
0100–01FF	SUBU ACC, *loc*	6-292
0200–02FF	MOVB ACC, *#8bit*	6-158
0300–03FF	SUBL ACC, *locLong*	6-288
0400–04FF	SUB ACC, *loc* << 16	6-273
0500–05FF	ADD ACC, *loc* << 16	6-43
0600–06FF	MOVL ACC, *locLong*	6-172
0700–07FF	ADDL ACC, *locLong*	6-58
0800–08FF for 1st word 2nd word is *16BitSigned*	ADD *loc, #16BitSigned*	6-43
0900–09FF	ADDB ACC, #*8bit*	6-51
0A00–0AFF	INC *loc*	6-106
0B00–0BFF	DEC *loc*	6-96
0C00–0CFF	ADDCU ACC, *loc*	6-55
0D00–0DFF	ADDU ACC, *loc*	6-62
0E00–0EFF	MOVU ACC, *loc*	6-183
0F00–0FFF	CMPL ACC, *locLong*	6-93
1000–10FF	MOVA T, *loc*	6-154
1100–11FF	MOVS T, *loc*	6-179
1200–12FF	MPY ACC, T, *loc*	6-186
1300–13FF	MPYS P, T, *loc*	6-197
1400–14FF for 1st word 2nd word is *pmem*	MAC P, *loc,* 0:*pmem*	6-132
1500–15FF for 1st word 2nd word is *16BitSigned*	MPYA P, *loc, #16BitSigned*	6-190

Table 6–14. Instruction Set Summary by Opcode (Continued)

Opcode or Opcode Range (Hexadecimal)	Instruction Syntax	Page
1600–16FF	MOVP T, *loc*	6-176
1700–17FF	MPYA P, T, *loc*	6-190
1800–18FF for 1st word 2nd word is *16BitMask*	AND *loc, #16BitMask*	6-66
1900–19FF	SUBB ACC, *#8bit*	6-280
1A00–1AFF for 1st word 2nd word is *16BitMask*	OR *loc, #16BitMask*	6-219
1B00–1BFF for 1st word 2nd word is *16BitSigned*	CMP *loc, #16BitSigned*	6-87
1C00–1CFF for 1st word 2nd word is *16BitMask*	XOR *loc, #16BitMask*	6-307
1D00–1DFF	SBBU ACC, *loc*	6-260
1E00–1EFF	MOVL *locLong,* ACC	6-172
1F00–1FFF	SUBCU ACC, *loc*	6-284
2000–20FF	MOV *loc*, IER	6-137
2100–21FF	MOV *loc*, T	6-137
2200–22FF	PUSH *loc*	6-236
2300–23FF	MOV IER, *loc*	6-137
2400–24FF	PREAD *loc*, *XAR7	6-233
2500–25FF	MOV ACC, *loc* << 16	6-137
2600–26FF	PWRITE *XAR7, *loc*	6-244
2700–27FF	MOV PL, *loc*	6-137
2800–28FF for 1st word 2nd word is *16bit*	MOV *loc, #16bit*	6-137
2900–29FF	CLRC *BitName1* {, *BitName2*}... or CLRC *8BitMask*	6-84
2A00–2AFF	POP *loc*	6-224
2B00–2BFF	MOV *loc* , #0	6-137

Table 6–14. Instruction Set Summary by Opcode (Continued)

Opcode or Opcode Range (Hexadecimal)	Instruction Syntax	Page
2C00–2CFF for 1st word 2nd word is *16BitMask*	LOOPZ *loc, #16BitMask*	6-122
2D00–2DFF	MOV T, *loc*	6-137
2E00–2EFF for 1st word 2nd word is *16BitMask*	LOOPNZ *loc, #16BitMask*	6-119
2F00–2FFF	MOV PH, *loc*	6-137
3000–30FF	MPYXU ACC, T, *loc*	6-204
3100–31FF	MPYB P, T, *#8bit*	6-195
3200–32FF	MPYXU P, T, *loc*	6-204
3300–33FF	MPY P, T, *loc*	6-186
3400–34FF for 1st word 2nd word is *16BitSigned*	MPY ACC, *loc, #16BitSigned*	6-186
3500–35FF	MPYB ACC, T, *#8bit*	6-195
3600–36FF	MPYU ACC, T, *loc*	6-201
3700–37FF	MPYU P, T, *loc*	6-201
3800–38FF	MOVB AL.MSB, *loc*	6-158
3900–39FF	MOVB AH.MSB, *loc*	6-158
3B00–3BFF	SETC *BitName1* {, *BitName2*}... or SETC *8BitMask*	6-264
3C00–3CFF	MOVB *loc*, AL.LSB	6-158
3D00–3DFF	MOVB *loc*, AH.LSB	6-158
3F00–3FFF	MOV *loc*, P	6-137
4000–4FFF	TBIT *loc, #BitNumber*	6-298
5000–50FF	ORB AL, *#8BitMask*	6-223
5100–51FF	ORB AH, *#8BitMask*	6-223
5200–52FF	CMPB AL, *#8bit*	6-91
5300–53FF	CMPB AH, *#8bit*	6-91

Table 6–14. Instruction Set Summary by Opcode (Continued)

Opcode or Opcode Range (Hexadecimal)	Instruction Syntax	Page
5400–54FF	CMP AL, *loc*	6-87
5500–55FF	CMP AH, *loc*	6-87
5700–57FF	MOVH *loc*, P	6-168
5800–5FFF	MOV AR*x*, *loc*	6-137
6000–6FFF	SB *8BitOffset, cond*	6-257
7000–70FF	XOR AL, *loc*	6-307
7100–71FF	XOR AH, *loc*	6-307
7200–72FF	ADD *loc*, AL	6-43
7300–73FF	ADD *loc*, AH	6-43
7400–74FF	SUB *loc*, AL	6-273
7500–75FF	SUB *loc*, AH	6-273
7600	POP ST1	6-224
7601	POP DP:ST1	6-224
7602	IRET	6-113
7603	POP DP	6-224
7604	CALL *XAR7	6-82
7605	POP AR3:AR2	6-224
7606	POP AR5:AR4	6-224
7607	POP AR1:AR0	6-224
7608	PUSH ST1	6-236
7609	PUSH DP:ST1	6-236
760A	PUSH IFR	6-236
760B	PUSH DP	6-236
760C	PUSH AR5:AR4	6-236
760D	PUSH AR1:AR0	6-236
760E	PUSH DBGIER	6-236

Table 6–14. Instruction Set Summary by Opcode (Continued)

Opcode or Opcode Range (Hexadecimal)	Instruction Syntax	Page
760F	PUSH AR3:AR2	6-236
7610	RETE	6-248
7611	POP PH:PL	6-224
7612	POP DBGIER	6-224
7613	POP ST0	6-224
7614	RET	6-247
7615	POP T:ST0	6-224
7616	INTR NMI	6-108
7617	NASP	6-207
7618	PUSH ST0	6-236
7619	PUSH T:ST0	6-236
761A	EDIS	6-99
761B	ASP	6-72
761D	PUSH PH:PL	6-236
761F for 1st word 2nd word is *16bit*	MOVW DP, *#16bit*	6-185
7620	LB *XAR7	6-118
7621	IDLE	6-104
7622	EALLOW	6-98
7623 for 1st word 2nd word is *16BitMask*	OR IER, *#16BitMask*	6-219
7624	ESTOP1	6-101
7625	ESTOP0	6-100
7626 for 1st word 2nd word is *16BitMask*	AND IER, *#16BitMask*	6-66
7627 for 1st word 2nd word is *16BitMask*	OR IFR, *#16BitMask*	6-219

Table 6–14. Instruction Set Summary by Opcode (Continued)

Opcode or Opcode Range (Hexadecimal)	Instruction Syntax	Page
762F for 1st word 2nd word is *16BitMask*	AND IFR, *#16BitMask*	6-66
763F for 1st word 2nd word is *VectorValue*	IACK *#VectorValue*	6-103
7680–76BF for 1st word 2nd word is *22bit*(15:0)	MOV XAR6, *#22bit*	6-137
76C0–76FF for 1st word 2nd word is *22bit*(15:0)	MOV XAR7, *#22bit*	6-137
7700–77FF	NOP syntaxes	6-212
7800–7FFF	MOV *loc*, AR*x*	6-137
8000–8FFF	SUB ACC *loc, {<<shift3}* for *shift3* < 16	6-273
9000–90FF	ANDB AL, *#8BitMask*	6-71
9100–91FF	ANDB AH, *#8BitMask*	6-71
9200–92FF	MOV AL, *loc*	6-137
9300–93FF	MOV AH, *loc*	6-137
9400–94FF	ADD AL, *loc*	6-43
9500–95FF	ADD AH, *loc*	6-43
9600–96FF	MOV *loc*, AL	6-137
9700–97FF	MOV *loc*, AH	6-137
9800–98FF	OR *loc*, AL	6-219
9900–99FF	OR *loc*, AH	6-219
9A00–9AFF	MOVB AL, *#8bit*	6-158
9B00–9BFF	MOVB AH, *#8bit*	6-158
9C00–9CFF	ADDB AL, *#8BitSigned*	6-51
9D00–9DFF	ADDB AH, *#8BitSigned*	6-51
9E00–9EFF	SUB AL, *loc*	6-273
9F00–9FFF	SUB AH, *loc*	6-273

Table 6–14. Instruction Set Summary by Opcode (Continued)

Opcode or Opcode Range (Hexadecimal)	Instruction Syntax	Page
A000–AFFF	ADD ACC *loc, {<<shift3}* for *shift3* < 16	6-43
B000–B7FF	MOVH *loc*, ACC *{<<shift1}*	6-168
B800–BFFF	MOV *loc*, ACC *{<<shift1}*	6-137
C000–C0FF	AND *loc*, AL	6-66
C100–C1FF	AND *loc*, AH	6-66
C200–C2FF	MOV *locLong*, XAR6	6-137
C300–C3FF	MOV *locLong*, XAR7	6-137
C400–C4FF	MOV XAR6, *locLong*	6-137
C500–C5FF	MOV XAR7, *locLong*	6-137
C600–C6FF	MOVB AL.LSB, *loc*	6-158
C700–C7FF	MOVB AH.LSB, *loc*	6-158
C800–C8FF	MOVB *loc*, AL.MSB	6-158
C900–C9FF	MOVB *loc*, AH.MSB	6-158
CA00–CAFF	OR AL, *loc*	6-219
CB00–CBFF	OR AH, *loc*	6-219
CC00–CCFF for 1st word 2nd word is *16BitMask*	AND AL, *loc, #16BitMask*	6-66
CD00–CDFF for 1st word 2nd word is *16BitMask*	AND AH, *loc, #16BitMask*	6-66
CE00–CEFF	AND AL, *loc*	6-66
CF00–CFFF	AND AH, *loc*	6-66
D000–D7FF	MOVB AR*x, #8bit*	6-158
D8**0**0–DF**F**F		
If the digit in the highlighted position is in the range 0–7:	ADDB *aux, #7bit*	6-51
If the digit in the highlighted position is in the range 8–F:	SUBB *aux, #7bit*	6-280

Table 6–14. Instruction Set Summary by Opcode (Continued)

Opcode or Opcode Range (Hexadecimal)	Instruction Syntax	Page
E000–EFFF	MOV ACC, *loc, {<<shift3}* for *shift3* < 16	6-137
F000–F0FF	XORB AL, *#8BitMask*	6-310
F100–F1FF	XORB AH, *#8BitMask*	6-310
F200–F2FF	XOR *loc*, AL	6-307
F300–F3FF	XOR *loc*, AH	6-307
F400–F4FF for 1st word 2nd word is *16bit*	MOV *(0:*16bit*), *loc*	6-137
F500–F5FF for 1st word 2nd word is *16bit*	MOV *loc,*(0:*16bit*)	6-137
F600–F6FF	RPT *#8bit*	6-251
F700–F7FF	RPT *loc*	6-251
F800–FBFF	MOV DP, *#10bit*	6-137
FC00–FCFF	ADRK *#8bit*	6-65
FD00–FDFF	SBRK *#8bit*	6-263
FE00–FE7F	ADDB SP, #7bit	6-51
FE80–FEFF	SUBB SP, #7bit	6-280
FF00–FF0F for 1st word 2nd word is *16BitSU*	SUB ACC, *#16BitSU* {<<*shift2*}	6-273
FF10–FF1F for 1st word 2nd word is *16BitSU*	ADD ACC, *#16BitSU* {<<*shift2*}	6-43
FF20–FF2F for 1st word 2nd word is *16BitSU*	MOV ACC, *#16BitSU* {<<*shift2*}	6-137
FF30–FF3F	LSL ACC, *shift*	6-125
FF40–FF4F	SFR ACC, *shift*	6-267
FF50	LSL ACC, T	6-125
FF51	SFR ACC, T	6-267
FF52	ROR ACC	6-250
FF53	ROL ACC	6-249

Table 6–14. Instruction Set Summary by Opcode (Continued)

Opcode or Opcode Range (Hexadecimal)	Instruction Syntax	Page
FF54	NEG ACC	6-208
FF55	NOT ACC	6-217
FF56	ABS ACC	6-41
FF57	SAT ACC	6-254
FF58	TEST ACC	6-301
FF59	CMP ACC, P	6-87
FF5A	MOV P, ACC	6-137
FF5C	NEG AL	6-208
FF5D	NEG AH	6-208
FF5E	NOT AL	6-217
FF5F	NOT AH	6-217
FF60	SXTB AL	6-296
FF61	SXTB AH	6-296
FF62	LSR AL, T	6-129
FF63	LSR AH, T	6-129
FF64	ASR AL, T	6-74
FF65	ASR AH, T	6-74
FF66	LSL AL, T	6-125
FF67	LSL AH, T	6-125
FF68–FF6F	SPM *ShiftMode*	6-271
FF70–FF77	NORM ACC, *aux*––	6-214
FF78–FF7F	NORM ACC, *aux*++	6-214
FF80–FF8F	LSL AL, *shift*	6-125
FF90–FF9F	LSL AH, *shift*	6-125
FFA0–FFAF	ASR AL, *shift*	6-74
FFB0–FFBF	ASR AH, *shift*	6-74

Table 6–14. Instruction Set Summary by Opcode (Continued)

Opcode or Opcode Range (Hexadecimal)	Instruction Syntax	Page
FFC0–FFCF	LSR AL, *shift*	6-129
FFD0–FFDF	LSR AH, *shift*	6-129
FFE0–FFEF for 1st word 2nd word is *16BitOffset*	B *16BitOffset*, *cond*	6-77
FFFF	ITRAP1	6-116

6.2 Alignment of 32-Bit Accesses to Even Addresses

The ’27x core expects memory wrappers and peripheral-interface logic to align any 32-bit data read or write to an even address. If the address-generation logic generates an odd address, the memory wrapper or peripheral interface must begin reading or writing at the previous even address. This alignment does not affect the address values generated by the address-generation logic.

Consider Example 6–1, which shows a 32-bit read from data memory. In this example, DP direct addressing mode is used. The data page pointer and the offset point to an odd address, 00 0085_{16}. However, the CPU must receive the first word from an even address. It reads from addresses 00 0084_{16} and 00 0085_{16}. Then it loads the word at 00 0084_{16} to the low half of ACC and the word at 00 0085_{16} to the high half of ACC.

Example 6–1. 32-Bit Read From Data-Memory

```
MOVL   ACC, @5   ; DP = 2. Offset = 5. Address = 000085
                 ; Load ACC with 32–bit value at
                 ; addressed 32-bit location.
```

Before instruction

DP	0002
ACC	0000 0000

Data memory

000084	4444
000085	5555
000086	6666

After instruction

DP	0002
ACC	5555 4444

Data memory

000084	4444
000085	5555
000086	6666

Example 6–2 shows a 32-bit write. AR0 points to the odd address 00 0085_{16}. Due to alignment, the low half of ACC is saved at address 00 0084_{16} and the high half of ACC is saved at address 00 0085_{16}. AR0 is not modified; it still holds 85 after the operation.

Example 6–2. 32-Bit Write to Data Memory

```
MOVL   *+AR0[0], ACC   ; AR0 = 85
                       ; Save 32–bit ACC to 32–bit location
                       ; pointed to by AR0.
```

Before instruction

AR0	0085
ACC	aaaa bbbb

Data memory

000084	4444
000085	5555
000086	6666

After instruction

AR0	0085
ACC	aaaa bbbb

Data memory

000084	bbbb
000085	aaaa
000086	6666

Table 6–15 describes the mechanisms on the ’27x that generate 32-bit-wide data accesses.

Table 6–15. Mechanisms That Generate 32-Bit-Wide Data Accesses

Mechanism	32-Bit Operation(s)
Interrupt initiated by hardware or software (OR, INTR, or TRAP instruction).	Multiple register pairs are automatically saved on the stack. Each pair is saved in a single 32-bit store operation.
IRET instruction	Multiple register pairs are automatically restored from the stack. Each pair is restored in a single 32-bit load operation.
Any of the following syntaxes of the PUSH instruction: PUSH T:ST0 PUSH AR1:AR0 PUSH DP:ST1 PUSH AR3:AR2 PUSH PH:PL PUSH AR5:AR4 PUSH XAR*n*	A register pair is saved on the stack in a single 32-bit store operation.
Any of the following syntaxes of the POP instruction: POP T:ST0 POP AR1:AR0 POP DP:ST1 POP AR3:AR2 POP PH:PL POP AR5:AR4 POP XAR*n*	A register pair is restored from the stack in a single 32-bit load operation.

Either of the following syntaxes of the MOV instruction:
MOV *locLong*, XAR*n*
MOV XAR*n*, *locLong*
	When XAR6 or XAR7 is the source, this 22-bit value is stored in a single 32-bit store operation. The 6 MSBs of the value are stored as 0s.
	When XAR6 or XAR7 is loaded, 32 bits are read from data memory. The 22 LSBs are loaded to the auxiliary register.

MOVL instruction	ACC is loaded or stored using a single 32-bit operation.
ADDL, CMPL, or SUBL instruction	A 32-bit value is read from data memory before being added or subtracted from ACC.

6.3 How to Use the Instruction Descriptions

Section 6.4 contains detailed information on the instruction set. The description for each instruction presents the following categories of information:

- ❏ Syntax
- ❏ Operands
- ❏ Description
- ❏ Execution
- ❏ Status Bits
- ❏ Repeat
- ❏ Words

Some descriptions also include a category that shows one or more examples.

6.3.1 Syntax

Each instruction begins with a list of the available assembler syntax expressions. For example, the description for the ADD instruction begins with:

1: **ADD** **A***X*, *loc*

2: **ADD** *loc*, **A***X*

3: **ADD** **ACC,** *loc* {**<<** *shift3*}

4: **ADD** **ACC,** #*16BitSU* {**<<** *shift2*}

5: **ADD** **ACC, P**

6: **ADD** *loc*, #*16BitSigned*

Each syntax is preceded by a reference number that is used throughout the rest of the instruction description.

Following are the notations used in the syntax expressions:

italic symbols and **boldface characters**

Italic symbols in an instruction syntax represent variables. Boldface characters in an instruction syntax must be typed as shown.

Example: **ADD A***X***,** *loc*

The *X* indicates that you can use AH or AL for the first operand. You can use a variety of values for *loc*. **ADD**, the following **A**, and the comma are boldface to indicate that they are always present when you use this syntax. Samples of this syntax follow:

```
ADD AH, @T
ADD AL, *AR2++
```

{x}

Operand x is optional.

Example: **ADD ACC,** *loc* {**<<** *shift3*}

You must supply ACC and *loc*, as in this instruction:

```
ADD ACC, @2
```

You have the option of adding a *shift3* value, as in this instruction:

```
ADD ACC, @2 << 4
```

#

The symbol # is a prefix for constants used in immediate-constant addressing mode.

Example: **ADD** *loc***, #***16BitSigned*

The variable *16BitSigned* is preceded by the symbol # to indicate that you must enter a constant. Conversely, because *loc* is not preceded by the symbol #, it cannot be a constant. It must be a reference to a memory location or a register.

Finally, consider this code example:

```
StoreResult  MOV   *, AL     ;Store low word of accumulator
                              ;to location referenced by the
                              ;current auxiliary register.
```

An optional label, StoreResult, is used as a reference preceding 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).

6.3.2 Operands

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

ADD **A***X*, *loc*

ADD *loc*, **A***X*

ADD **ACC,** *loc* {**<<** *shift3*}

ADD **ACC,** #*16BitSU* {**<<** *shift2*}

ADD **ACC, P**

ADD *loc*, #*16BitSigned*

The operands category defines *16BitSigned*, *16BitSU*, **ACC**, **A***X*, *loc*, *shift3*, and *shift2*.

6.3.3 Opcode

The opcode category shows the 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. Opcode fields that do not directly relate to operands are either defined in the opcode category or in section 5.7, *Addressing-Mode Information in Opcodes*, on page 5-21. For example, this is one of the opcodes given for the ADD instruction:

Syntax 6: **ADD** *loc*, #*16BitSigned*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	See section 5.7 on page 5-21.							
16BitSigned															

The field *16BitSigned* relates directly to an operand defined in the operands category. Bits 7 through 0 of the first word depend on the addressing mode used. Because they require some explanation, the field contains a reference to section 5.7.

6.3.4 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.

6.3.5 Execution

The execution category presents an instruction operation sequence that describes the processing that takes place when the instruction is executed. 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 The content of x is assigned to register or location y.

Example: [addressed location] → ACC

The content of the specified data-memory location or register is put into the accumulator.

[x] << z The content of x is shifted left by z bit positions.

Example: [ACC] << 16

The content of ACC is shifted left by 16.

[x] >> z The content of x is shifted right by z bit positions.

Example: [ACC] >> 1

The content of ACC is shifted right by 1.

r(n) Bit n of register or location r.

Example: ACC(31) represents bit 31 of 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.

r1:r2 Registers r1 and r2 have been concatenated into one 32-bit data value. r2 is the low word; r1 is the high word.

Example: T:ST0 represents the concatenation of the T register and status register ST0. T forms the high word, and ST0 forms the low word.

6.3.6 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 how certain bits (if any) affect the execution of the instruction and how certain bits (if any) are affected by the instruction.

6.3.7 Repeat

Some of the instructions are repeatable. That is, if they are preceded by the RPT instruction, the CPU repeats them the number of times specified in the RPT instruction. The repeat category indicates whether the instruction is repeatable. If an instruction is repeatable, the repeat category provides tips for using this feature.

6.3.8 Words

The words category specifies the number of words (one or two) required to store the instruction in program memory. When the number of words depends on the chosen syntax, the words category specifies which syntaxes require one word and which require two words.

6.3.9 Examples

This category, when present, provides an example or examples that show the effects of the instruction on memory and/or registers. Program code is shown in a `special typeface`. The code is followed by a verbal and/or graphical description of the effects of that code. All memory and register values in the examples are shown as hexadecimal numbers.

6.4 Instruction Descriptions

This section contains detailed information on the instruction set for the ’27x. (For a summary of the instruction set, see section 6.1.) The instructions are presented alphabetically, and the description for each instruction provides the following categories of information:

- ❏ Syntax
- ❏ Operands
- ❏ Description
- ❏ Execution
- ❏ Status Bits
- ❏ Repeat
- ❏ Words

Most descriptions also include a category that shows one or more examples. For an explanation of how to use each of these categories, see section 6.3. The number of cycles needed for each instruction is included in the tables in section 6.1.2 (page 6-9).

Syntax **ABORTI**

Operands None

Opcode

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

Description This instruction is available for emulation purposes. Generally, a program uses the IRET instruction to return from an interrupt. The IRET instruction restores all the values that were saved to the stack during the automatic context save. In restoring status register ST1 and the debug status register (DBGSTAT), IRET restores the debug context that was present before the interrupt.

In some target applications, you might have interrupts that must not be returned from by the IRET instruction. Not using IRET can cause a problem for the emulation logic, because the emulation logic assumes the original debug context will be restored. The abort interrupt (ABORTI) instruction is provided as a means to indicate that the debug context will not be restored and the debug logic needs to be reset to its default state. As part of its operation, the ABORTI instruction:

- ❏ Sets the DBGM bit in ST1. This disables debug events.
- ❏ Modifies select bits in DBGSTAT. The effect is a resetting of the debug context. If the CPU was in the debug-halt state before the interrupt occurred, the CPU does not halt when the interrupt is aborted.

The ABORTI instruction does not modify the DBGIER, the IER, the INTM bit, or any analysis registers (for example, registers used for breakpoints, watchpoints, and data logging).

Execution Reset debug context.
Disable future debug events:
$1 \rightarrow$ DBGM

Status Bits ABORTI sets DBGM in ST1.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Words 1

Syntax **ABS ACC**

Operands **ACC** Accumulator

Opcode

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

Description If ACC holds a negative number, the ABS instruction replaces that number with its absolute value.

Execution

For ACC = 8000 0000_{16}:

If OVM = 1
    $7FFF\ FFFF_{16} \rightarrow ACC$
else
    $8000\ 0000_{16} \rightarrow ACC$
$1 \rightarrow V$
$0 \rightarrow C$
$[PC] + 1 \rightarrow PC$

For ACC not equal to 8000 0000_{16}:

If ACC < 0
    $-[ACC] \rightarrow ACC$
$0 \rightarrow C$
$[PC] + 1 \rightarrow PC$

Status Bits

OVM — If ACC is 8000 0000_{16} at the start of the operation, it is considered an overflow value. The ACC value after the operation depends on OVM:

- OVM = 0 ACC is filled again with 8000 0000_{16}.
- OVM = 1 ACC is filled with its greatest positive number, $7FFF\ FFFF_{16}$.

SXM — SXM does not affect the operation.

C — C is cleared.

N — If bit 31 of the result in ACC is 1, N is set; otherwise, N is cleared.

OVC — OVC is not affected by the operation.

V — If ACC = 8000 0000_{16} at the start of the operation, it is considered an overflow value, and V is set. Otherwise, V is not affected.

Z — If ACC = 0 after the ABS operation, Z is set; otherwise, Z is cleared.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Words 1

Example 1

```
ABS    ACC              ; ACC = 8000 0000 (hex)
; For OVM = 0: ACC kept at its greatest negative value
```

	Before instruction	After instruction
ACC	8000 0000	8000 0000
	C = X N = X	C = 0 N = 1
	V = X Z = X	V = 1 Z = 0

```
; For OVM = 1: ACC filled with its greatest positive value
```

	Before instruction	After instruction
ACC	8000 0000	7fff ffff
	C = X N = X	C = 0 N = 0
	V = X Z = X	V = 1 Z = 0

Example 2

```
ABS    ACC              ; ACC not equal to 8000 0000 (hex)
```

	Before instruction	After instruction
ACC	ffff fffe	0000 0002
	C = X N = X	C = 0 N = 0
	V = X Z = X	V unchanged Z = 0

Syntax

1: **ADD** **A***X*, *loc*
2: **ADD** *loc*, **A***X*
3: **ADD** **ACC,** *loc* {**<<** *shift3*}
4: **ADD** **ACC,** #*16BitSU* {**<<** *shift2*}
5: **ADD** **ACC, P**
6: **ADD** *loc*, #*16BitSigned*

Operands

16BitSigned 16-bit signed number from –32 768 to 32 767.

16BitSU If SXM = 0, *16BitSU* is an unsigned 16-bit number from 0 to 65 535. If SXM = 1, *16BitSU* is a signed 16-bit number from –32 768 to 32 767.

ACC Accumulator

A*X* Use AH (the high word of the accumulator) or AL (the low word of the accumulator).

loc Reference to a 16-bit location, either a data-memory location or a 16-bit register. Use any of these types of operands:

- @**0:***6bit* PAGE0-direct-addressing operand. Data page is 0. Offset is specified by *6bit*, a 6-bit constant from 0 to 63. PAGE0 direct addressing mode is available only when PAGE0 = 1.
- @*6bit* DP-direct-addressing operand. Data page is specified by the DP. Offset is specified by *6bit*, a 6-bit constant from 0 to 63.
- @*reg* Register-addressing operand. For *reg*, specify one of these register names:

AH	AL	PL	PH	T	SP
AR0	AR1	AR2	AR3	AR4	AR5
AR6	AR7				

- ***–SP[***6bit**]** PAGE0-stack-addressing operand. Address in data memory is specified as 0:(SP – *6bit*), where 0: indicates that the six MSBs are 0s and *6bit* is a 6-bit constant from 0 to 63. PAGE0 stack addressing mode is available only when PAGE0 = 0.

ind Indirect-addressing operand. Select one of the following operands (*x* is a number from 0 to 5; *n* is 6 or 7; *z* is a number from 0 to 7; *3bit* is a 3-bit constant):

*SP++	*AR*x*	*XAR*n*	*ARPz
*––SP	*AR*x*++	*XAR*n*++	*
	*––AR*x*	*––XARn	*++
	*+AR*x*[AR0]	*+XARn[AR0]	*––
	*+AR*x*[AR1]	*+XARn[AR1]	*0++
	*+AR*x*[*3bit*]	*+XARn[*3bit*]	*0––
	*AR6%++		

P Product register

shift3 Number from 0 to 16

shift2 Number from 0 to 15

Opcode

Syntax 1: **ADD** **A***X*, *loc*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	0	1	0	AX	See section 5.7.2 on page 5-23.							

Note: If AL is used in the instruction, AX = 0; if AH is used, AX = 1.

Syntax 2: **ADD** *loc*, **A***X*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	0	0	1	AX	See section 5.7.2 on page 5-23.							

Note: If AL is used in the instruction, AX = 0; if AH is used, AX = 1.

Syntax 3: **ADD** **ACC,** *loc* {**<<** *shift3*}

For *shift3* < 16

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	*shift3*				See section 5.7.2 on page 5-23.							

For *shift3* = 16

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	1	See section 5.7.2 on page 5-23.							

Syntax 4: **ADD ACC,** #*16BitSU* {**<<** *shift2*}

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	0	0	0	1	*shift2*			
16BitSU															

Syntax 5: **ADD ACC, P**

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

Note: This is the opcode for the following equivalent instruction: MOVA T, @T.

Syntax 6: **ADD** *loc***,** #*16BitSigned*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	See section 5.7.2 on page 5-23.							
16BitSigned															

Description

The general form for the ADD instruction is as follows:

ADD *location1***,** *operand2*

The value referenced or supplied by *operand2* is added to the value at *location1*, and the result is stored to *location1*.

In syntaxes 3 and 4, the SXM bit affects *operand2*. If SXM = 0, *operand2* is treated as unsigned. If SXM = 1, *operand2* is treated as signed. In syntaxes 3 and 4, a left shift can also be specified:

ADD *location1***,** *operand2* **<<** *shift*

The value given by *operand2* is left shifted before being added to the value at *location1*. During the shift, low-order bits are zero filled. If SXM = 1, the value is sign extended; if SXM = 0, high-order bits are zero filled.

Execution

Syntax 1: **ADD A***X***,** *loc*

[AX] + [addressed location] → AX*;*
[PC] +1 → PC

Syntax 2: **ADD** *loc*, **A***X*

[addressed location] + AX → addressed location
[PC] + 1 → PC

AX and the addressed value are treated as signed numbers.
The six most significant bits of XAR6 and XAR7 are not affected by loads to AR6 and AR7, respectively.

Syntax 3: **ADD ACC,** *loc* {**<<** *shift3*}

If SXM = 0
    [addressed location] is unsigned
If SXM = 1
    [addressed location] is signed
[ACC] + [addressed location] → ACC
[PC] + 1 → PC

If a shift is specified, the addressed value is shifted before the addition. During the shift, low-order bits are zero filled. If SXM = 1, the value is sign extended; if SXM = 0, high-order bits are zero filled.

Syntax 4: **ADD ACC,** #*16BitSU* {**<<** *shift2*}

If SXM = 0
    [ACC] + 16-bit unsigned constant → ACC
If SXM = 1
    [ACC] + 16-bit signed constant → ACC
[PC] + 2 → PC

If a shift is specified, the 16-bit constant is shifted before the addition. During the shift, low-order bits are zero filled. If SXM = 1, the value is sign extended; if SXM = 0, high-order bits are zero filled.

Syntax 5: **ADD ACC, P**

[ACC] + ([P] shifted as per PM bits) → ACC
[PC] + 1 → PC

Syntax 6: **ADD** *loc*, #*16BitSigned*

[addressed location] + 16-bit signed constant → addressed location
[PC] + 2 → PC

The addressed value is treated as a signed number.
The six most significant bits of XAR6 and XAR7 are not affected by loads to AR6 and AR7, respectively.

Status Bits

Syntaxes 1, 2, and 6:

OVM, SXM — Neither affects the operation.

C — If the addition generates a carry, C is set; otherwise, C is cleared.

N — If bit 15 of the result is 1, N is set; otherwise, N is cleared.

V — If an overflow occurs, V is set; otherwise, V is not affected.

Z — If the result is 0, Z is set; otherwise, Z is cleared.

Syntaxes 3, 4, and 5 (ACC is destination):

OVM, OVC — If the operation causes ACC to overflow, ACC and OVC depend on OVM:

- OVM = 0 — ACC is not filled with a saturation value. OVC is affected as follows:

  If the overflow was in the negative direction, OVC is decremented by 1; if the overflow was in the positive direction, OVC is incremented by 1.

- OVM = 1 — If the overflow was in the positive direction, ACC is filled with 7FFF FFFF$_{16}$. If the overflow was in the negative direction, ACC is filled with 8000 0000$_{16}$.

  OVC is not affected.

SXM — For syntaxes 3 and 4, SXM affects the 16-bit source operand as follows:

- SXM = 0 — The 16-bit source operand is treated as an unsigned number. The value is not sign extended during the operation.
- SXM = 1 — The 16-bit source operand is treated as a signed number. The value is signed extended during the operation.

PM — For syntax 5, the P value is shifted as defined by PM before it is added to ACC. (P itself is not modified.)

C — If the addition generates a carry, C is set; otherwise, C is cleared. *Exception:* If a shift of 16 is used, the ADD instruction can set C but not clear C.

N	After the addition, if bit 31 of ACC is 1, N is set; otherwise, N is cleared.
V	If an overflow occurs, V is set; otherwise, V is not affected.
Z	If the result is 0, Z is set; otherwise, Z is cleared.

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Words

Syntaxes 1, 2, 3, and 5: 1

Syntaxes 4 and 6: 2

Example 1

```
ADD    AH, @10
; Add referenced value to AH. Causes an overflow in AH.
```

Before instruction		After instruction	
DP	0003	DP	0003
ACC	7fff eeee	ACC	8001 eeee
Data memory		Data memory	
0000ca	0002	0000ca	0002
C = X	N = X	C = 0	N = 1
V = X	Z = X	V = 1	Z = 0

Example 2

```
ADD    @T, #–6
```

Before instruction		After instruction	
T	0006	T	0000
C = X	N = X	C = 1	N = 0
V = X	Z = X	V unchanged	Z = 1

Example 3

```
ADD    ACC, *AR5++ << 12
```

Before instruction		After instruction	
AR5	0086	AR5	0087
Data memory		Data memory	
000086	8000	000086	8000

; For SXM = 0: Data value *not* sign extended during operation
; ACC = [ACC] + (data value << 12) = 0000 0001_{16} + 0800 0000_{16}

ACC	0000 0001	ACC	0800 0001
C = X	N = X	C = 0	N = 0
V = X	Z = X	V unchanged	Z = 0

; For SXM = 1: Data value sign extended during operation
; ACC = [ACC] + (data value << 12) = 0000 0001_{16} + f800 0000_{16}

ACC	0000 0001	ACC	f800 0001
C = X	N = X	C = 0	N = 1
V = X	Z = X	V unchanged	Z = 0

Example 4

```
ADD    ACC, #0fffdh
```

; For SXM = 0: [ACC] + 0000 $fffd_{16}$ (constant unsigned)

Before instruction		After instruction	
ACC	0000 0002	ACC	0000 ffff
C = X	N = X	C = 0	N = 0
V = X	Z = X	V unchanged	Z = 0

; For SXM = 1: [ACC] + ffff $fffd_{16}$ (constant signed)

Before instruction		After instruction	
ACC	0000 0002	ACC	ffff ffff
C = X	N = X	C = 0	N = 1
V = X	Z = X	V unchanged	Z = 0

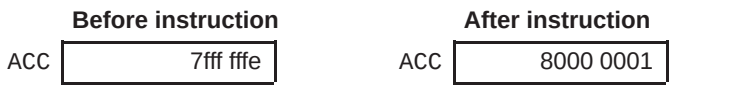
Example 5

```
ADD    ACC, @PL
```

	Before instruction		After instruction
P	aaaa 0003	P	aaaa 0003

```
; For OVM = 0: ACC overflows normally. OVC records overflow.
```

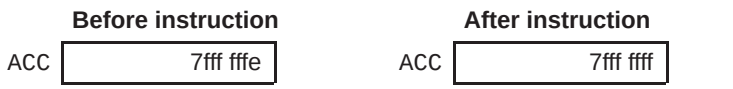
	Before instruction		After instruction
ACC	7fff fffe	ACC	8000 0001



```
OVC = 0                    OVC = 1
C = X      N = X           C = 0        N = 1
V = X      Z = X           V = 1        Z = 0
```

```
; For OVM = 1: ACC filled with saturation value
```

	Before instruction		After instruction
ACC	7fff fffe	ACC	7fff ffff



```
OVC = X                    OVC unchanged
C = X      N = X           C = 0        N = 0
V = X      Z = X           V = 1        Z = 0
```

Syntax

1: **ADDB** **A***X*, #*8BitSigned*
2: **ADDB** **ACC,** #*8bit*
3: **ADDB** *aux*, #*7bit*
4: **ADDB** **SP,** #*7bit*

Operands

7bit	7-bit number from 0 to 127
8bit	8-bit number from 0 to 255
8BitSigned	8-bit signed number from –128 to 127
aux	Use one of the following auxiliary register names: AR0 AR1 AR2 AR3 AR4 AR5 XAR6 XAR7
A*X*	Use AH (the high word of the accumulator) or AL (the low word of the accumulator).
ACC	Accumulator
SP	Stack pointer

Opcode

Syntax 1: **ADDB** **A***X*, #*8BitSigned*

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

Note: If AL is used in the instruction, AX = 0; if AH is used, AX = 1.

Syntax 2: **ADDB** **ACC,** #*8bit*

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

Syntax 3: **ADDB** *aux*, #*7bit*

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

The following table shows the relationship between *aux* and auxNumber:

aux	**auxNumber**	***aux***	**auxNumber**
AR0	0	AR4	4
AR1	1	AR5	5
AR2	2	XAR6	6
AR3	3	XAR7	7

Syntax 4: **ADDB** **SP,** #*7bit*

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

Description

The general form of the ADDB instruction is as follows:

ADDB *location1***,** *short value*

The short (7- or 8-bit) value is added to the value in *location1*, and the result is stored in *location1*. When a value is added to the SP or one of the auxiliary registers, the addition is performed by the address register arithmetic unit (ARAU). When a value is added to AH, AL, or ACC, the addition is performed by the arithmetic logic unit (ALU).

Execution

Syntax 1: **ADDB** **A***X***,** #*8BitSigned*

[AX] + 8-bit signed constant → AX
[PC] + 1 → PC

Syntax 2: **ADDB** **ACC,** #*8bit*

[ACC] + 8-bit unsigned constant → ACC
[PC] + 1 → PC

Syntax 3: **ADDB** *aux***,** #*7bit*

[auxiliary register] + 7-bit unsigned constant → auxiliary register
[PC] + 1 → PC

Syntax 4: **ADDB** **SP,** #*7bit*

[SP] + 7-bit unsigned constant → SP
[PC] + 1 → PC

Status Bits

Syntax 1:

OVM, SXM	Neither affects the operation.
C	If the addition generates a carry, C is set; otherwise, C is cleared.
N	After the addition, if bit 15 of the result is 1, N is set; otherwise, N is cleared.
V	If an overflow occurs, V is set; otherwise, V is not affected.
Z	If the result is 0, Z is set; otherwise, Z is cleared.

Syntax 2:

OVM, OVC	If the operation causes ACC to overflow, ACC and OVC depend on OVM:
	OVM = 0 ACC is not filled with a saturation value. OVC is affected as follows: If the overflow was in the negative direction, OVC is decremented by 1; if the overflow was in the positive direction, OVC is incremented by 1.
	OVM = 1 If the overflow was in the positive direction, ACC is filled with 7FFF FFFF$_{16}$. If the overflow was in the negative direction, ACC is filled with 8000 0000$_{16}$. OVC is not affected.
SXM	SXM does not affect the operation.
C	If the addition generates a carry, C is set; otherwise, C is cleared.
N	After the addition, if bit 31 of ACC is 1, N is set; otherwise, N is cleared.
V	If an overflow occurs, V is set; otherwise, V is not affected.
Z	If the result is 0, Z is set; otherwise, Z is cleared.

Syntaxes 3 and 4:

OVM, SXM	Neither affects the operation.
C, N, V, Z	None are affected by the operation.

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Words

1

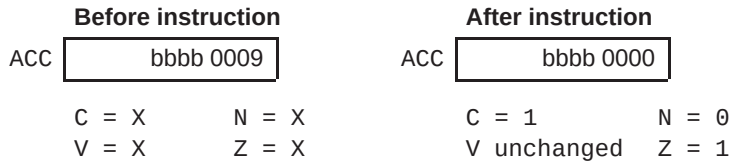
Example 1

```
ADDB   XAR6, #2
```

	Before instruction		After instruction	
XAR6	10 0001		XAR6	10 0003
C = X	N = X		C unchanged	N unchanged
V = X	Z = X		V unchanged	Z unchanged

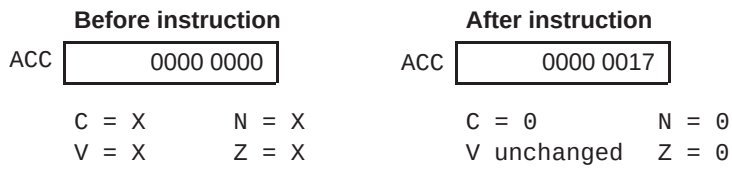
Example 2

```
ADDB   AL, #–9
```



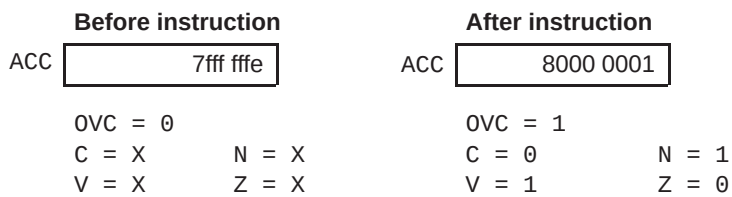
Example 3

```
ADDB   ACC, #23
```

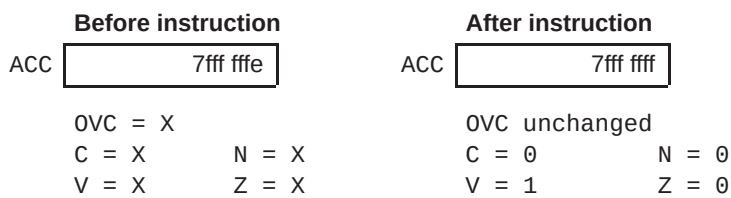


Example 4

```
ADDB   ACC, #3

; For OVM = 0: ACC overflows normally. OVC records overflow.
```




```
; For OVM = 1: ACC filled with saturation value
```



Syntax **ADDCU ACC,** *loc*

Operands

ACC Accumulator

loc Reference to a 16-bit location, either a data-memory location or a 16-bit register. Use any of these types of operands:

- **@0:***6bit* PAGE0-direct-addressing operand. Data page is 0. Offset is specified by *6bit*, a 6-bit constant from 0 to 63. PAGE0 direct addressing mode is available only when PAGE0 = 1.
- *@6bit* DP-direct-addressing operand. Data page is specified by the DP. Offset is specified by *6bit*, a 6-bit constant from 0 to 63.
- *@reg* Register-addressing operand. For *reg*, specify one of these register names:

AH	AL	PL	PH	T	SP
AR0	AR1	AR2	AR3	AR4	AR5
AR6	AR7				

- ***–SP[***6bit***]*** PAGE0-stack-addressing operand. Address in data memory is specified as 0:(SP – *6bit*), where 0: indicates that the six MSBs are 0s and *6bit* is a 6-bit constant from 0 to 63. PAGE0 stack addressing mode is available only when PAGE0 = 0.
- ***ind*** Indirect-addressing operand. Select one of the following operands (*x* is a number from 0 to 5; *n* is 6 or 7; *z* is a number from 0 to 7; *3bit* is a 3-bit constant):

*SP++	*AR*x*	*XAR*n*	*ARPz
*––SP	*AR*x*++	*XAR*n*++	*
	*––AR*x*	*––XARn	*++
	*+AR*x*[AR0]	*+XARn[AR0]	*––
	*+AR*x*[AR1]	*+XARn[AR1]	*0++
	*+AR*x*[*3bit*]	*+XARn[*3bit*]	*0––
	*AR6%++		

Opcode

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	0	0	See section 5.7.2 on page 5-23.							

Description The contents of the addressed location and the value of the carry bit are added to the accumulator with sign extension suppressed. The content of the addressed location is treated as an unsigned number. The carry bit can then be affected by the operation (see the *Status Bits* category).

Execution

[ACC] + [addressed location] + [C] → ACC
[PC] + 1 → PC

Status Bits

OVM, OVC — If the operation causes ACC to overflow, ACC and OVC depend on OVM:

- OVM = 0 — ACC is not filled with a saturation value. OVC is affected as follows:

  If the overflow was in the negative direction, OVC is decremented by 1; if the overflow was in the positive direction, OVC is incremented by 1.

- OVM = 1 — If the overflow was in the positive direction, ACC is filled with 7FFF $FFFF_{16}$. If the overflow was in the negative direction, ACC is filled with 8000 0000_{16}.

  OVC is not affected.

SXM — SXM does not affect the operation.

C — If the addition generates a carry, C is set; otherwise, C is cleared.

N — After the addition, if bit 31 of ACC is 1, N is set; otherwise, N is cleared.

V — If an overflow occurs, V is set; otherwise, V is not affected.

Z — If the result is 0, Z is set; otherwise, Z is cleared.

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Words

1

Example 1

```
ADDCU  ACC, @6

; Data value treated as unsigned
; ACC = [ACC] + fffc16 + [C] = 2 + 65 532 + 1 = 0000 ffff16
```

Before instruction		After instruction	
DP	0003	DP	0003
ACC	0000 0002	ACC	0000 ffff
Data memory		Data memory	
0000c6	fffc	0000c6	fffc

```
C = 1      N = X          C = 0        N = 0
V = X      Z = X          V unchanged  Z = 0
```

Example 2

```
ADDCU  ACC, @AR2

; AR2 value treated as unsigned
; ACC = [ACC] + [AR2] + [C] = 7fff fffe16 + 216 + 016
```

Before instruction		After instruction	
AR2	0002	AR2	0002

```
; For OVM = 0: ACC overflows normally. OVC records overflow.
```

Before instruction		After instruction	
ACC	7fff fffe	ACC	8000 0000

```
OVC = 0                   OVC = 1
C = 0      N = X          C = 0        N = 1
V = X      Z = X          V = 1        Z = 0

; For OVM = 1: ACC filled with saturation value
```

Before instruction		After instruction	
ACC	7fff fffe	ACC	7fff ffff

```
OVC = X                   OVC unchanged
C = 0      N = X          C = 0        N = 0
V = X      Z = X          V = 1        Z = 0
```

Syntax **ADDL ACC,** *locLong*

Operands

ACC Accumulator

locLong Reference to a 32-bit data-memory location or one of the 22-bit auxiliary registers (XAR6 or XAR7). Use any of these types of operands:

- **@0:***6bit* PAGE0-direct-addressing operand. Data page is 0. Offset is specified by *6bit*, a 6-bit constant from 0 to 63. PAGE0 direct addressing mode is available only when PAGE0 = 1.
- @*6bit* DP-direct-addressing operand. Data page is specified by the DP. Offset is specified by *6bit*, a 6-bit constant from 0 to 63.
- @**XAR***n* Register-addressing operand. For **XAR***n*, specify XAR6 or XAR7.
- ***–SP[***6bit***]*** PAGE0-stack-addressing operand. Address in data memory is specified as 0:(SP – *6bit*), where 0: indicates that the six MSBs are 0s and *6bit* is a 6-bit constant from 0 to 63. PAGE0 stack addressing mode is available only when PAGE0 = 0.
- ***ind* Indirect-addressing operand. Select one of the following operands (*x* is a number from 0 to 5; *n* is 6 or 7; *z* is a number from 0 to 7; *3bit* is a 3-bit constant):

*SP++	*AR*x*	*XAR*n*	*ARPz
*––SP	*AR*x*++	*XAR*n*++	*
	*––AR*x*	*––XARn	*++
	*+AR*x*[AR0]	*+XARn[AR0]	*––
	*+AR*x*[AR1]	*+XARn[AR1]	*0++
	*+AR*x*[*3bit*]	*+XARn[*3bit*]	*0––
	*AR6%++		

Opcode

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	1	See section 5.7.2 on page 5-23.							

Description

The content of the location addressed by *locLong* is added to the content of ACC, and the result is placed in ACC. The addressed location is either a data-memory location holding a 32-bit value or one of the extended auxiliary registers (XAR6 or XAR7) holding a 22-bit value. To specify XAR6, use the register-addressing operand @XAR6; to specify XAR7, use @XAR7. If @XAR6 or @XAR7 is used, the corresponding auxiliary-register value is not sign extended; it is treated as an unsigned number. If a register-addressing operand other than @XAR6 or @XAR7 is used, an illegal-instruction trap (TRAP #19 instruction) is generated.

The ’27x core expects memory wrappers and peripheral-interface logic to force any 32-bit access, like that of the ADDL instruction, to align to an even address. This alignment is described in section 6.2 on page 6-31.

Execution

If *locLong* is @XAR6 or @XAR7
    [ACC] + unsigned [auxiliary register] → ACC
    [PC] + 1 → PC
else
    [ACC] + addressed 32-bit data-memory value → ACC
    [PC] + 1 → PC

Status Bits

OVM, OVC — If the operation causes ACC to overflow, ACC and OVC depend on OVM:

- OVM = 0 — ACC is not filled with a saturation value. OVC is affected as follows:

  If the overflow was in the negative direction, OVC is decremented by 1; if the overflow was in the positive direction, OVC is incremented by 1.

- OVM = 1 — If the overflow was in the positive direction, ACC is filled with 7FFF $FFFF_{16}$. If the overflow was in the negative direction, ACC is filled with 8000 0000_{16}.

  OVC is not affected.

SXM — SXM does not affect the operation.

C — If the addition generates a carry, C is set; otherwise, C is cleared.

N — After the addition, if bit 31 of ACC is 1, N is set; otherwise, N is cleared.

V — If an overflow occurs, V is set; otherwise, V is not affected.

Z — If the result is 0, Z is set; otherwise, Z is cleared.

Repeat

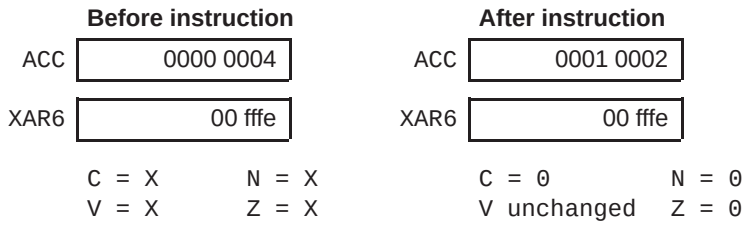
This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Words

1

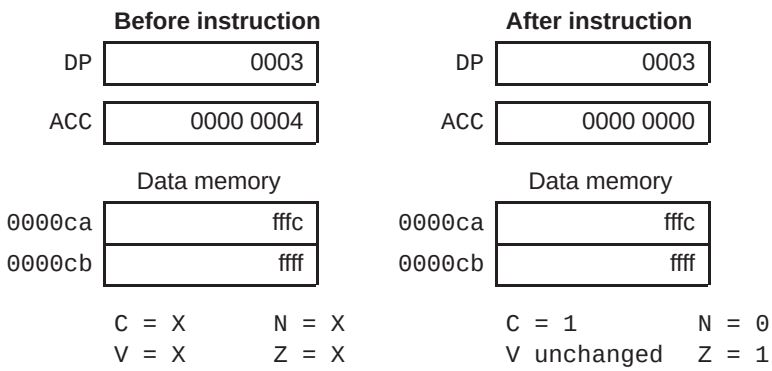
Example 1

```
ADDL   ACC, @XAR6

; XAR6 treated as unsigned
; ACC = [ACC] + [XAR6] = 4 + 65 534 = 0001 0002₁₆
```



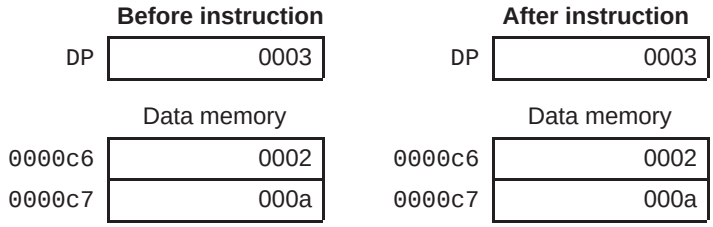
Example 2

```
ADDL   ACC, @10

; 32-bit value read from two consecutive 16-bit locations
; and treated as signed
; ACC = [ACC] + ffff fffc₁₆ = 4 + (–4) = 0
```

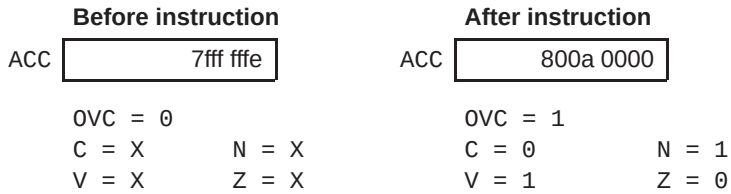


Example 3

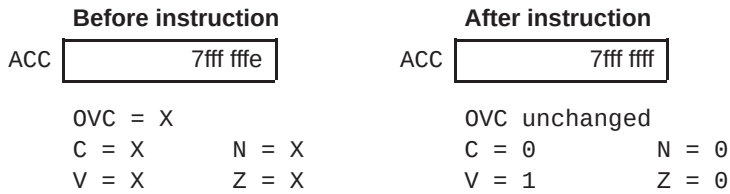
```
ADDL   ACC, @7

; Data value treated as signed. Read aligned to even address.
; ACC  = [ACC] + addressed 32-bit value
       = 7fff fffe16 + 000a 000216
```




```
; For OVM = 0: ACC overflows normally. OVC records overflow.
```




```
; For OVM = 1: ACC filled with saturation value
```



Syntax **ADDU ACC,** *loc*

Operands

ACC Accumulator

loc Reference to a 16-bit location, either a data-memory location or a 16-bit register. Use any of these types of operands:

- **@0:***6bit* PAGE0-direct-addressing operand. Data page is 0. Offset is specified by *6bit*, a 6-bit constant from 0 to 63. PAGE0 direct addressing mode is available only when PAGE0 = 1.
- @*6bit* DP-direct-addressing operand. Data page is specified by the DP. Offset is specified by *6bit*, a 6-bit constant from 0 to 63.
- @*reg* Register-addressing operand. For *reg*, specify one of these register names:

AH	AL	PL	PH	T	SP
AR0	AR1	AR2	AR3	AR4	AR5
AR6	AR7				

- ***–SP[***6bit***]*** PAGE0-stack-addressing operand. Address in data memory is specified as 0:(SP – *6bit*), where 0: indicates that the six MSBs are 0s and *6bit* is a 6-bit constant from 0 to 63. PAGE0 stack addressing mode is available only when PAGE0 = 0.
- ***ind* Indirect-addressing operand. Select one of the following operands (*x* is a number from 0 to 5; *n* is 6 or 7; *z* is a number from 0 to 7; *3bit* is a 3-bit constant):

*SP++	*AR*x*	*XAR*n*	*ARPz
*––SP	*AR*x*++	*XAR*n*++	*
	*––AR*x*	*––XARn	*++
	*+AR*x*[AR0]	*+XARn[AR0]	*––
	*+AR*x*[AR1]	*+XARn[AR1]	*0++
	*+AR*x*[*3bit*]	*+XARn[*3bit*]	*0––
	*AR6%++		

Opcode

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	0	1	See section 5.7.2 on page 5-23.							

Description The value addressed by *loc* is added to the content of ACC, and the result is placed in ACC. Sign extension is suppressed. The addressed value is treated as an unsigned number.

Execution

[ACC] + unsigned [addressed location] → ACC
[PC] + 1 → PC

Status Bits

OVM, OVC — If the operation causes ACC to overflow, ACC and OVC depend on OVM:

- OVM = 0 — ACC is not filled with a saturation value. OVC is affected as follows:

  If the overflow was in the negative direction, OVC is decremented by 1; if the overflow was in the positive direction, OVC is incremented by 1.

- OVM = 1 — If the overflow was in the positive direction, ACC is filled with 7FFF $FFFF_{16}$. If the overflow was in the negative direction, ACC is filled with 8000 0000_{16}.

  OVC is not affected.

SXM — SXM does not affect the operation.

C — If the addition generates a carry, C is set; otherwise, C is cleared.

N — After the addition, if bit 31 of ACC is 1, N is set; otherwise, N is cleared.

V — If an overflow occurs, V is set; otherwise, V is not affected.

Z — If the result is 0, Z is set; otherwise, Z is cleared.

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Words

1

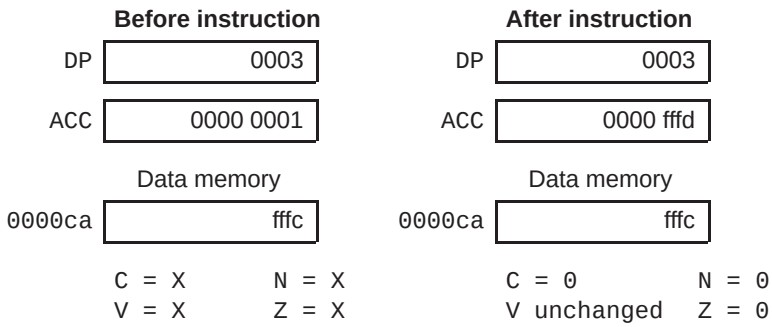
Example 1

```
ADDU   ACC, @AR6

; AR6 = XAR6(15:0), treated as unsigned
; ACC = [ACC] + [AR6] = 4 + 65 534 = 0001 0002(16)
```

	Before instruction	After instruction
ACC	0000 0004	0001 0002
XAR6	3a fffe	3a fffe
	C = X N = X	C = 0 N = 0
	V = X Z = X	V unchanged Z = 0

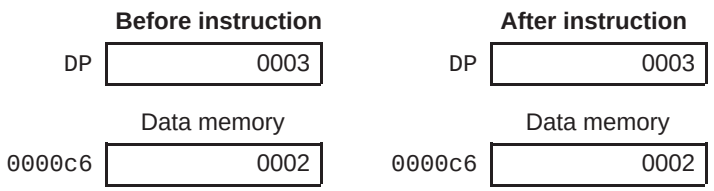
Example 2

```
ADDU   ACC, @10

; Data value treated as unsigned
; ACC = [ACC] + fffc16 = 1 + 65 532 = 0000 fffd16
```

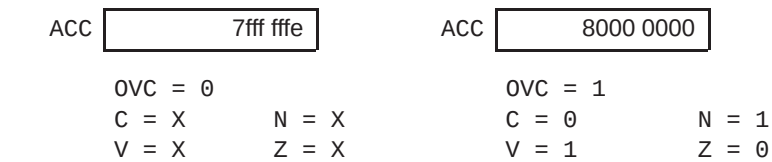


Example 3

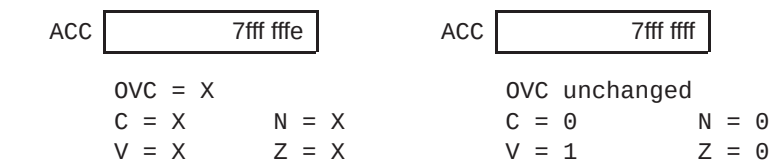
```
ADDU   ACC, @6

; Data value treated as unsigned
; ACC = [ACC] + addressed value = 7fff fffe16 +  0000 000216
```




```
; For OVM = 0: ACC overflows normally. OVC records overflow.
```




```
; For OVM = 1: ACC filled with saturation value
```



Syntax **ADRK** *#8bit*

Operands *8bit* 8-bit number from 0 to 255

Opcode

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

Description The 8-bit unsigned constant is added to the content of the *current auxiliary register*. The current auxiliary register is defined as the auxiliary register that is pointed to by the auxiliary register pointer (ARP). For example, if ARP = 2, the current auxiliary register is AR2; if ARP = 7, the current auxiliary register is XAR7.

Execution [current auxiliary register] + 8-bit constant → current auxiliary register
[PC] + 1 → PC

Status Bits

ARP	Points to the current auxiliary register
OVM, SXM	Neither affects the operation.
C, N, V, Z	None of these is affected by the operation.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Words 1

Example

```
ADRK   #8
; ARP = 3 (AR3 is current auxiliary register)
```

Before instruction		After instruction	
AR3	0000	AR3	0008
C = X	N = X	C unchanged	N unchanged
V = X	Z = X	V unchanged	Z unchanged

Syntax

1: **AND** **A***X*, *loc*
2: **AND** *loc*, **A***X*
3: **AND** **A***X*, *loc*, #*16BitMask*
4: **AND** **IER,** #*16BitMask*
5: **AND** **IFR,** #*16BitMask*
6: **AND** *loc*, #*16BitMask*

Operands

16BitMask 16-bit mask value from 0000_{16} to $FFFF_{16}$

A*X* Use AH (the high word of the accumulator) or AL (the low word of the accumulator).

loc Reference to a 16-bit location, either a data-memory location or a 16-bit register. Use any of these types of operands:

- **@0:***6bit* PAGE0-direct-addressing operand. Data page is 0. Offset is specified by *6bit*, a 6-bit constant from 0 to 63. PAGE0 direct addressing mode is available only when PAGE0 = 1.
- @*6bit* DP-direct-addressing operand. Data page is specified by the DP. Offset is specified by *6bit*, a 6-bit constant from 0 to 63.
- @*reg* Register-addressing operand. For *reg*, specify one of these register names:

AH	AL	PL	PH	T	SP
AR0	AR1	AR2	AR3	AR4	AR5
AR6	AR7				

- ***–SP[***6bit***]*** PAGE0-stack-addressing operand. Address in data memory is specified as 0:(SP – *6bit*), where 0: indicates that the six MSBs are 0s and *6bit* is a 6-bit constant from 0 to 63. PAGE0 stack addressing mode is available only when PAGE0 = 0.

ind Indirect-addressing operand. Select one of the following operands (*x* is a number from 0 to 5; *n* is 6 or 7; *z* is a number from 0 to 7; *3bit* is a 3-bit constant):

*SP++	*AR*x*	*XAR*n*	*ARPz
*––SP	*AR*x*++	*XAR*n*++	*
	*––AR*x*	*––XARn	*++
	*+AR*x*[AR0]	*+XARn[AR0]	*––
	*+AR*x*[AR1]	*+XARn[AR1]	*0++
	*+AR*x*[*3bit*]	*+XARn[*3bit*]	*0––
	*AR6%++		

IER Interrupt enable register

IFR Interrupt flag register

Opcode

Syntax 1: **AND** **A***X*, *loc*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	1	1	1	AX	See section 5.7.2 on page 5-23.							

Note: If AL is used in the instruction, AX = 0; if AH is used, AX = 1.

Syntax 2: **AND** *loc*, **A***X*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	AX	See section 5.7.2 on page 5-23.							

Note: If AL is used in the instruction, AX = 0; if AH is used, AX = 1.

Syntax 3: **AND** **A***X*, *loc*, #*16BitMask*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	1	1	0	AX	See section 5.7.2 on page 5-23.							
16BitMask															

Note: If AL is used in the instruction, AX = 0; if AH is used, AX = 1.

Syntax 4: **AND** **IER,** #*16BitMask*

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	0	1	0	0	1	1	0
16BitMask															

Syntax 5: **AND** **IFR,** #*16BitMask*

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	0	1	0	1	1	1	1
16BitMask															

Syntax 6: **AND** *loc*, #*16BitMask*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	0	0	0	See section 5.7.2 on page 5-23.							
16BitMask															

Description

For all syntaxes except 3, the general form for the AND instruction is as follows:

AND *location1*, *operand2*

A bitwise AND operation is performed with the value referenced or supplied by *operand2* and the value at *location1*, and the result is stored to *location1*.

For syntax 3, the general form is as follows:

AND **A***X*, *location2*, *operand2*

A*X* represents either AL or AH. A bitwise AND operation is performed with the value referenced or supplied by *operand2* and the value at *location2*, and the result is stored to AL or AH.

Execution

Syntax 1: **AND** **A***X*, *loc*

[AX] AND [addressed location] → AX
[PC] +1 → PC

Syntax 2: **AND** *loc*, **A***X*

[addressed location] AND [AX] → addressed location
[PC] + 1 → PC

The six most significant bits of XAR6 and XAR7 are not affected by loads to AR6 and AR7, respectively.

Syntax 3: **AND** **A***X*, *loc*, #*16BitMask*

[addressed location] AND 16-bit constant → AX
[PC] + 2 → PC

Syntax 4: **AND** **IER,** #*16BitMask*

[IER] AND 16-bit constant → IER
[PC] + 2 → PC

Syntax 5: **AND** **IFR,** #*16BitMask*

[IFR] AND 16-bit constant → IFR
[PC] + 2 → PC

Syntax 6: **AND** *loc*, #*16BitMask*

[addressed location] AND 16-bit constant → [addressed location]
[PC] + 2 → PC

The six most significant bits of XAR6 and XAR7 are not affected by loads to AR6 and AR7, respectively.

Status Bits

Syntaxes 1, 2, 3, and 6:

OVM, SXM	Neither affects the operation.
C, V	Neither are affected by the operation.
N	If bit 15 of the result is 1, N is set; otherwise, N is cleared.
Z	If the result is 0, Z is set; otherwise, Z is cleared.

Syntaxes 4 and 5 (involving IER and IFR):

OVM, SXM	Neither affects the operation.
C, N, V, Z	None are affected by the operation.

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Words

Syntaxes 1 and 2: 1
Syntaxes 3, 4, 5, and 6: 2

Example 1

```
AND    @10, AH
; Result at 00 00ca16 = (value at 00 00ca16) AND [AH]
```

Before instruction		After instruction	
DP	0003	DP	0003
ACC	0011 aaaa	ACC	0011 aaaa
Data memory		Data memory	
0000ca	ffee	0000ca	0000

```
C = X      N = X          C unchanged  N = 0
V = X      Z = X          V unchanged  Z = 1
```

Example 2

```
AND    AL, @AR2, #8800h      ; Hexadecimal mask used
; AL = [AR2] AND 880016
```

Before instruction		After instruction	
ACC	3636 3636	ACC	3636 8000
AR2	f3f3	AR2	f3f3

```
C = X      N = X          C unchanged  N = 1
V = X      Z = X          V unchanged  Z = 0
```

Example 3

```
AND    IER, #0101h           ; Hexadecimal mask used
```

Before instruction		After instruction	
IER	00ff	IER	0001

```
C = X      N = X          C unchanged  N unchanged
V = X      Z = X          V unchanged  Z unchanged
```

Syntax **ANDB** **A***X*, #*8BitMask*

Operands

A*X* Use AH (the high word of the accumulator) or AL (the low word of the accumulator).

8BitMask 8-bit mask value from 00_{16} to FF_{16}

Opcode

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

Note: If AL is used in the instruction, AX = 0; if AH is used, AX = 1.

Description A bitwise AND operation is performed with the 8-bit mask value and $00FF_{16}$ to form a 16-bit mask value. Then a bitwise AND operation is performed with the 16-bit mask value and the content of the specified accumulator half (AH or AL); the result is placed in that accumulator half.

Execution

[AX] AND (8-bit constant AND $00FF_{16}$) $\rightarrow$ AX
[PC] + 1 $\rightarrow$ PC

Status Bits

OVM, SXM	Neither affects the operation.
C, V	Neither are affected by the operation.
N	Because bit 15 of the result is 0, N is cleared.
Z	If the result is 0, Z is set; otherwise, Z is cleared.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Words 1

Example

```
ANDB   AH, #03h
```

Before instruction

ACC | fffe bbbb |

```
C = X      N = X
V = X      Z = X
```

After instruction

ACC | 0002 bbbb |

```
C unchanged   N = 0
V unchanged   Z = 0
```

Syntax **ASP**

Operands None

Opcode

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	0	0	1	1	0	1	1

Description This instruction ensures that the stack pointer (SP) is aligned to an even address. If the least significant bit (LSB) of SP is 1, SP points to an odd address and must be moved. SP is incremented by 1, and the SPA bit is set as a record of this alignment.

If, instead, the ASP instruction finds the LSB of SP to be 0, SP already points to an even address. SP is left unchanged, and the SPA bit is cleared to indicate that no alignment has taken place.

In either case, the change to the SPA bit is made in the decode 2 phase of the pipeline.

If you wish to undo a previous alignment done by the ASP instruction, use the NASP instruction.

Execution

```
If SP(0) = 1
    1 → SPA
    [SP] + 1 → SP
    [PC] +1 → PC
else
    0 → SPA
    [PC] + 1 → PC
```

Status Bits

SPA	If SP holds an odd address before the operation, SPA is set; otherwise, SPA is cleared.
OVM, SXM	Neither affects the operation.
C, N, V, Z	None are affected by the operation.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Words 1

Example 1

```
ASP         ;SP holds an even address
```

Example 2

```
ASP         ;SP holds an odd address
```

Syntax

1: **ASR** **A***X*, *shift*

2: **ASR** **A***X*, **T**

Operands

A*X*	Use AH (the high word of the accumulator) or AL (the low word of the accumulator).
shift	Number from 1 to 16
T	Multiplicand register

Opcode

Syntax 1: **ASR** **A***X*, *shift*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	0	1	AX	(*shift* – 1)			

Note: If AL is used in the instruction, AX = 0; if AH is used, AX = 1.

Syntax 2: **ASR** **A***X*, **T**

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

Note: If AL is used in the instruction, AX = 0; if AH is used, AX = 1.

Description

The content of AH or AL is shifted right by an amount specified by *shift* or by the four least significant bits (LSBs) of the T register, T(3:0):

- ***shift.*** Specify a constant from 1 to 16 as the second operand of the instruction. During the shift operation, the value is sign extended. In addition, the last bit to be shifted out of the register is stored in the carry bit (C).
- **T(3:0).** Specify T as the second operand of the instruction. The four LSBs of T enable a shift from 0 to 15. During the shift operation, the value is sign extended. If the T register specifies a shift of 0, C is cleared; otherwise, C is filled with the last bit to be shifted out of AH or AL.

If you need to perform this kind of shift but without sign extension, use the logical shift right (LSR) instruction. To perform a logical or arithmetic right shift on ACC, use the SFR instruction.

Execution

Syntax 1: **ASR A*X*, *shift***

[AX(shift – 1)] → C
[AX] shifted right and sign extended → AX
[PC] +1 → PC

Syntax 2: **ASR A*X*, T**

shift = T(3:0)
If shift = 0
    0 → C
    [AX] → AX
    [PC] + 1 → PC
else
    [AX(shift – 1)] → C
    [AX] shifted right and sign extended → AX
    [PC] + 1 → PC

Status Bits

Syntax 1: **ASR A*X*, *shift***

OVM, SXM	Neither affects the operation.
C	The last bit to be shifted out of AH or AL is stored in C.
N	If bit 15 of the result is 1, N is set; otherwise, N is cleared.
V	V is not affected by the operation.
Z	If the result is 0, Z is set; otherwise, Z is cleared.

Syntax 2: **ASR A*X*, T**

OVM, SXM	Neither affects the operation.
C	If the T register specifies a shift of 0, C is cleared; otherwise, the last bit to be shifted out of AH or AL is stored in C.
N	If bit 15 of the result is 1, N is set; otherwise, N is cleared. Even if the T register specifies a shift of 0, the value of AH or AL is tested for the negative condition and N is affected.
V	V is not affected by the operation.
Z	If the result is 0, Z is set; otherwise, Z is cleared. Even if the T register specifies a shift of 0, the value of AH or AL is tested for the zero condition and Z is affected.

Repeat

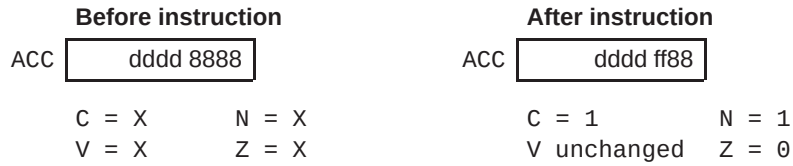
This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Words

1

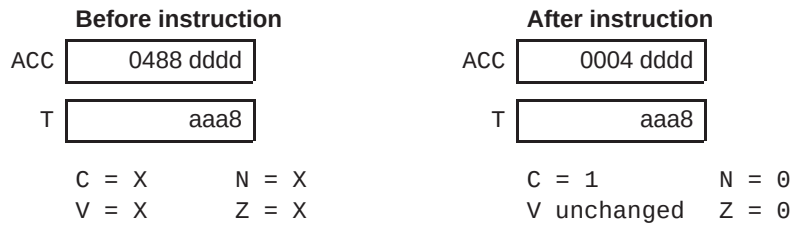
Example 1

```
ASR    AL, 8
```

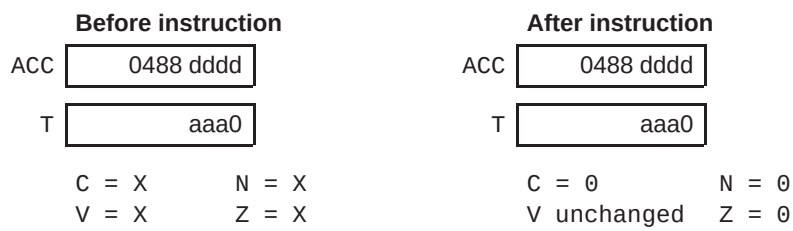


Example 2

```
ASR    AH, T          ; T(3:0) holds shift count
; For T(3:0) > 0
```




```
; For T(3:0) = 0
```



Syntax **B** *16BitOffset*, *cond*

Operands

16BitOffset 16-bit signed offset from –32 768 to 32 767

cond Condition to be tested. See Table 6–16.

Opcode

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	1	1	0	See Table 6–16.			
16BitOffset															

Description

If the specified condition is true, the specified 16-bit signed offset is added to the current PC value (the start address of the B instruction). This forces program control to the new address, (PC + *16BitOffset*). The 16-bit constant is sign extended to 22 bits before the addition. As shown by the following example, the offset is with respect to the address of the B instruction.

```
label:    .               ; code specified by user
          .
          .
          B     offset,EQ ; <– PC
                          ; offset = label – PC
```

If the specified condition is not true, the PC is incremented by 2 to force program control to the instruction that follows the B instruction.

Table 6–16 shows the conditions that can be tested.

Table 6–16. Conditions and Their Corresponding Opcode Segments and Flag Tests

cond	**Condition**	**Condition Code in Opcode**	**Flag Test Performed**
NEQ	Not equal to 0	0000	Z = 0
EQ	Equal to 0	0001	Z = 1
GT	Greater than 0	0010	Z = 0 AND N = 0
GEQ	Greater than or equal to 0	0011	N = 0
LT	Less than 0	0100	N = 1
LEQ	Less than or equal to 0	0101	Z = 1 OR N = 1
HI	Higher	0110	C = 1 AND Z = 0
HIS or C	Higher or same or C = 1	0111	C = 1
LO or NC	Lower or C = 0	1000	C = 0
LOS	Lower or same	1001	C = 0 OR Z = 1
NOV	No overflow	1010	V = 0
OV	Overflow	1011	V = 1
NTC	TC = 0	1100	TC = 0
TC	TC = 1	1101	TC = 1
(reserved)	–	1110	–
UNC	Unconditional	1111	None

Execution

If specified condition is true
    [PC] + 16-bit signed offset → PC
else,
    [PC] + 2 → PC

Status Bits

OVM, SXM	Neither affects the operation.
C, N, Z	None of these are affected by the operation.
V	If the condition of V is tested (*cond* is OV or NOV), V is cleared.

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Words

2

Example 1

```
B  –150, GT

; Test for greater–than condition.
; If N = 0 and Z = 0, branch backward 150 addresses.
```

Before instruction		After instruction	
C = X	N = X	C unchanged	N unchanged
V = X	Z = X	V unchanged	Z unchanged

Example 2

```
B  Continue, NOV

; Test for no–overflow condition.
; If V = 0, branch to the address labeled Continue.
```

Before instruction		After instruction	
C = X	N = X	C unchanged	N unchanged
V = X	Z = X	V = 0	Z unchanged

Syntax

BANZ *16BitOffset*, **AR***x***––**

Operands

16BitOffset 16-bit signed offset from –32 768 to 32 767

AR*x* Use one of the following:
AR0 AR1 AR2 AR3 AR4 AR5
AR6 AR7

Opcode

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	1	auxNumber		
16BitOffset															

The following table shows the relationship between the specified auxiliary register and auxNumber:

Auxiliary Register	auxNumber	Auxiliary Register	auxNumber
AR0	0	AR4	4
AR1	1	AR5	5
AR2	2	AR6	6
AR3	3	AR7	7

Description

If the content of the specified auxiliary register is not equal to 0, the 16-bit signed offset is added to the PC value (the start address of the branch instruction). This forces program control to the new address, (PC + *16BitOffset*). The 16-bit offset is sign extended to 22 bits before the addition. Then the content of the auxiliary register is decremented by 1. As shown by following example, the offset is with respect to address of the BANZ instruction.

```
label: .                    ;code specified by user
       .
       .
       BANZ      offset,EQ ; <– PC
                           ; offset = label – PC
```

If the auxiliary register value is 0, the PC is incremented by 2, forcing program control to the instruction that follows the BANZ instruction. Then the content of the auxiliary register is decremented by 1.

When one of the extended auxiliary registers (XAR6 or XAR7) is used, only the 16 LSBs (AR6 or AR7) are involved in the operation. That is, only AR6/AR7 is tested for the zero condition and then decremented; the six most significant bits of the extended auxiliary register are not affected.

Execution

If AR*x* ≠ 0
    [PC] + 16-bit signed constant → PC
    [AR*x*] – 1 → AR*x*
else
    [PC] + 2 → PC
    [AR*x*] – 1 → AR*x*

Status Bits

OVM, SXM Neither affects the operation.

C, N, V, Z None of these are affected by the operation.

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Words

2

Example 1

```
BANZ   Loop, AR7--

; If AR7 is not 0, branch to address labeled Loop.
; Decrement AR7 by 1 regardless.
```

	Before instruction		After instruction
XAR7	aa 0003	XAR7	aa 0002
	ARP = X		ARP unchanged

Example 2

```
BANZ   -5, AR4--

; If AR4 is not 0, branch backward 5 addresses.
; Decrement AR4 by 1 regardless.
```

	Before instruction		After instruction
AR4	0002	AR4	0001
	ARP = X		ARP unchanged

Syntax

1: **CALL** *22BitAddress*

2: **CALL** ***XAR7**

Operands

22BitAddress 22-bit program-memory address from $00\,0000_{16}$ to $3F\,FFFF_{16}$

XAR7 Extended auxiliary register XAR7

Opcode

Syntax 1: **CALL** *22BitAddress*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	1	0	*22BitAddress*(21:16)					
22BitAddress(15:0)															

Syntax 2: **CALL** ***XAR7**

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	0	0	0	0	1	0	0

Description

The CALL instruction first stores the incremented PC value (return address) to the stack. Then it causes a branch by loading the PC with a 22-bit value. This 22-bit value is supplied in the instruction (syntax 1) or is taken from extended auxiliary register XAR7 (syntax 2).

Execution

Syntax 1: **CALL** *22BitAddress*

[PC] + 2 → PC
[PC(15:0)] → address referenced by SP
[SP] + 1 → SP
([PC(21:16)] AND $003F_{16}$) → address referenced by SP
22-bit constant → PC

Syntax 2: **CALL** ***XAR7**

[PC] + 1 → PC
[PC(15:0)] → address referenced by SP
[SP] + 1 → SP
([PC(21:16)] AND $003F_{16}$) → address referenced by SP
[XAR7] → PC

Status Bits

OVM, SXM Neither affects the operation.

C, N, V, Z None of these are affected by the operation.

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Words

Syntax 1: 2

Syntax 2: 1

Example 1

```
CALL   86h      ; Call subroutine at the given hexadecimal
                ; address. Store return address on stack.
```

Example 2

```
CALL   *XAR7    ; Call subroutine at the address in XAR7
```

Syntax

1: **CLRC** *BitName1* { , *BitName2* } ... { , *BitName8* }

2: **CLRC** *8BitMask*

Operands

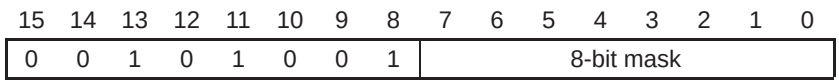
8BitMask 8-bit mask value from 00_{16} to FF_{16}

BitName Choose one of the following bits in status registers ST0 and ST1. More than one of these bits can be used at once, separated by commas.

SXM	Sign-extension mode bit (bit 0 of ST0)
OVM	Overflow mode bit (bit 1 of ST0)
TC	Test/control flag bit (bit 2 of ST0)
C	Carry bit (bit 3 of ST0)
INTM	Interrupt global mask bit (bit 0 of ST1)
DBGM	Debug enable mask bit (bit 1 of ST1)
PAGE0	PAGE0 addressing modes configuration bit (bit 2 of ST1)
VMAP	Vector map bit (bit 3 of ST1)

Opcode

If you use syntax 2, the 8-bit mask value you specify is embedded in the eight low-order bits of the opcode. If you use syntax 1, the CPU derives the corresponding 8-bit mask and then embeds it in the eight low-order bits of the opcode. Both syntaxes use the following opcode:



Description

The specified status bit or bits are cleared. You can specify from one to eight of the allowable status bits using syntax 1 or syntax 2. Here is an example of syntax 1 (the order of the operands does not matter):

```
CLRC PAGE0, DBGM, INTM, TC, SXM
```

This clears status bits PAGE0, DBGM, INTM, TC, and SXM. To perform the same operation with syntax 2, you would use:

```
CLRC 01110101b
```

The b indicates that, in this case, the operand is a binary number. This operand is a mask value that relates to the status bits in this way:

Mask-bit position	7	6	5	4	3	2	1	0
Mask value	0	1	1	1	0	1	0	1
Status bit	VMAP	PAGE0	DBGM	INTM	C	TC	OVM	SXM

Where a mask bit is 1, the corresponding status bit is cleared; where a mask bit is 0, the corresponding bit is not affected. As summarized in Table 6–17:

- Some of the status bits are changed in the execute phase of the pipeline, and others are changed in the decode 2 phase of the pipeline. The CLRC INTM instruction enables interrupts at the end of the decode 2 phase of the pipeline.
- Any CLRC instruction that clears DBGM and/or INTM takes an additional cycle to execute.

Table 6–17. Status Bits as Affected by the CLRC Instruction

Mask-Bit Position	Status Bit	Value Changed In	Cycles
0	SXM	Execute phase	1
1	OVM	Execute phase	1
2	TC	Execute phase	1
3	C	Execute phase	1
4	INTM	Decode 2 phase	2
5	DBGM	Decode 2 phase	2
6	PAGE0	Decode 2 phase	1
7	VMAP	Decode 2 phase	1

Execution

Clear specified status bit(s)
[PC] + 1 → PC

Status Bits

The operation is not affected by OVM or SXM. Any (allowable) status bit that is specified in the instruction is cleared. No others are affected.

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Words

1

Example 1

```
CLRC   SXM                    ; Clear SXM (SXM = 0)
```

Example 2

There are two ways to clear VMAP, PAGE0, and OVM with CLRC:

```
CLRC   VMAP, PAGE0, OVM       ; bit names used
CLRC   #11000010b             ; binary code used
```

Syntax

1: **CMP** **A***X*, *loc*
2: **CMP** **ACC, P**
3: **CMP** *loc*, #*16BitSigned*

Operands

16BitSigned 16-bit signed number from –32 768 to 32 767.

ACC Accumulator

A*X* Use AH (the high word of the accumulator) or AL (the low word of the accumulator).

loc Reference to a 16-bit location, either a data-memory location or a 16-bit register. Use any of these types of operands:

- **@0:***6bit* PAGE0-direct-addressing operand. Data page is 0. Offset is specified by *6bit*, a 6-bit constant from 0 to 63. PAGE0 direct addressing mode is available only when PAGE0 = 1.
- *@6bit* DP-direct-addressing operand. Data page is specified by the DP. Offset is specified by *6bit*, a 6-bit constant from 0 to 63.
- *@reg* Register-addressing operand. For *reg*, specify one of these register names:

AH	AL	PL	PH	T	SP
AR0	AR1	AR2	AR3	AR4	AR5
AR6	AR7				

- ***–SP[***6bit***]*** PAGE0-stack-addressing operand. Address in data memory is specified as 0:(SP – *6bit*), where 0: indicates that the six MSBs are 0s and *6bit* is a 6-bit constant from 0 to 63. PAGE0 stack addressing mode is available only when PAGE0 = 0.

***ind** Indirect-addressing operand. Select one of the following operands (*x* is a number from 0 to 5; *n* is 6 or 7; *z* is a number from 0 to 7; *3bit* is a 3-bit constant):

*SP++	*AR*x*	*XAR*n*	*ARPz
*––SP	*AR*x*++	*XAR*n*++	*
	*––AR*x*	*––XARn	*++
	*+AR*x*[AR0]	*+XARn[AR0]	*––
	*+AR*x*[AR1]	*+XARn[AR1]	*0++
	*+AR*x*[*3bit*]	*+XARn[*3bit*]	*0––
	*AR6%++		

P Product register

Opcode

Syntax 1: **CMP** **A***X*, *loc*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	0	1	0	AX	See section 5.7.2 on page 5-23.							

Note: If AL is used in the instruction, AX = 0; if AH is used, AX = 1.

Syntax 2: **CMP** **ACC, P**

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

Syntax 3: **CMP** *loc*, *#16BitSigned*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	0	1	1	See section 5.7.2 on page 5-23.							
16BitSigned															

Description

The general form of the CMP instruction is

CMP *operand1*, *operand2*

The value referenced or supplied by *operand2* is subtracted from the value referenced by *operand1*. The relationship between operand1 and operand2 is then reflected by the Z and N bits. If the values are the same, the result is 0 (Z = 1). If operand1 is greater than operand2, the result is positive (N = 0) and nonzero (Z = 0). If operand1 is less than operand2, the result is negative (N = 1).

The compare operation treats the setting of the N flag different from the SUB operation. For more details, see section 2.3, *Status Register ST0*, on page 2-13.

Execution

Syntax 1: **CMP** **A*X*,** *loc*

Change C, N, and Z based on following calculation and output of ALU:
[AX] – [addressed location] → ALU output
[PC] +1 → PC

Syntax 2: **CMP** **ACC, P**

Change C, N, and Z based on following calculation and output of ALU:
[ACC] – ([P] shifted as per PM bits) → ALU output
[PC] +1 → PC

Syntax 3: **CMP** *loc*, #*16BitSigned*

Change C, N, and Z based on following calculation and output of ALU:
[addressed location] – 16-bit signed constant → ALU output
[PC] +2 → PC

The addressed memory or register value is treated as a signed number.

Status Bits

OVM, SXM	Neither affects the operation.
PM	For syntax 2, the P register value is shifted as defined by PM before it is subtracted from ACC. (P itself is not modified.)
V, OVC	V and OVC are not affected by the operation.
C	If the subtraction generates a borrow, C is cleared; otherwise, C is set.
N	If the result is negative, N is set; otherwise, N is cleared. The CMP instruction assumes infinite precision when it determines the sign of the result. For example, consider the subtraction $8000\ 0000_{16} – 0000\ 0001_{16}$. If the precision were limited to 32 bits, the result would cause an overflow to the positive number $7FFF\ FFFF_{16}$, and N would be cleared. However, because the CMP instruction assumes infinite precision, it would set N to indicate that $8000\ 0000_{16} – 0000\ 0001_{16}$ actually results in a negative number.
Z	If the result is 0, Z is set; otherwise, Z is cleared.

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

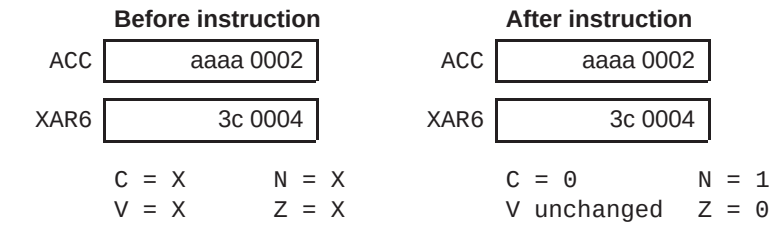
Words

Syntaxes 1 and 2: 1

Syntax 3: 2

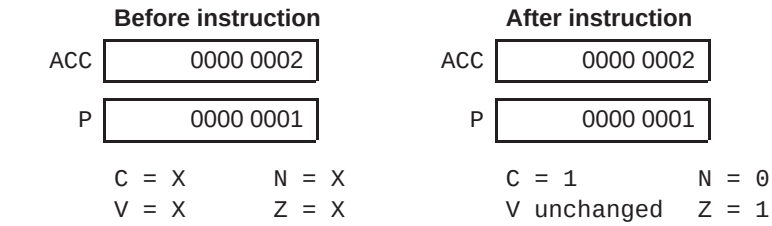
Example 1

```
CMP    AL, @AR6

; [AL] – [AR6] = 0002₁₆ – 0004₁₆ = –2
```



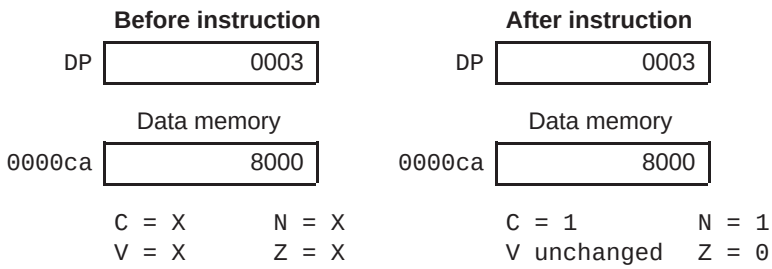
Example 2

```
CMP    ACC, P

; Product shift mode is PM = 0 (shift left by 1).
; [ACC] – ([P] << 1) = 0000 0002₁₆ – 0000 0002₁₆ = 0
```



Example 3

```
CMP    @10, #1

; (Value at 00 00ca₁₆) – 1 = 8000₁₆ –  0001₁₆
```



Syntax **CMPB** **A***X*, #*8bit*

Operands

8bit 8-bit number from 0 to 255

A*X* Use AH (the high word of the accumulator) or AL (the low word of the accumulator).

Opcode

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

Note: If AL is used in the instruction, AX = 0; if AH is used, AX = 1.

Description

The supplied 8-bit unsigned number is subtracted from content of A*X* (AH or AL). The relationship between A*X* and *8bit* is then reflected by the Z and N bits. If the values are the same, the result is 0 (Z = 1). If A*X* is greater than *8bit*, the result is positive (N = 0) and nonzero (Z = 0). If A*X* is less than *8bit*, the result is negative (N = 1).

The compare operation treats the setting of the N flag different from the SUB operation. For more details, see section 2.3, *Status Register ST0*, on page 2-13.

Execution

Change C, N, and Z based on following calculation and output of ALU:

[AX] – 8-bit unsigned number → ALU output
[PC] +1 → PC

Status Bits

OVM, SXM Neither affects the operation.

C If the subtraction generates a borrow, C is cleared; otherwise, C is set.

N If the result is negative, N is set; otherwise, N is cleared. The CMP instruction assumes infinite precision when it determines the sign of the result. For example, consider the subtraction $8000_{16} – 0001_{16}$. If the precision were limited to 16 bits, the result would cause an overflow to the positive number $7FFF_{16}$, and N would be cleared. However, because the CMP instruction assumes infinite precision, it would set N to indicate that $8000_{16} – 0001_{16}$ actually results in a negative number.

V V is not affected by the operation.

Z If the result is 0, Z is set; otherwise, Z is cleared.

Repeat

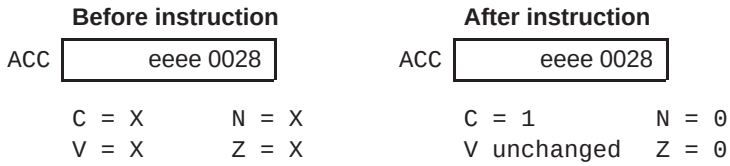
This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Words 1

Example 1

```
CMPB   AL, #20
```

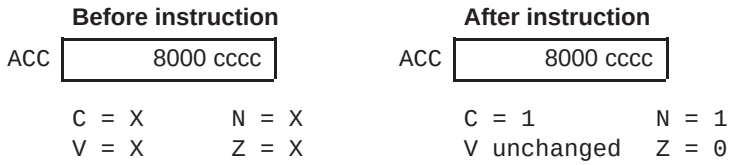
; [AL] – 20 = 40 – 20 = 20 = 0014_{16}



Example 2

```
CMPB   AH, #8
```

; [AH] – 8 = 8000_{16} – 0008_{16}



Syntax **CMPL** **ACC,** *locLong*

Operands

ACC Accumulator

locLong Reference to a 32-bit data-memory location or one of the 22-bit auxiliary registers (XAR6 or XAR7). Use any of these types of operands:

- **@0:***6bit* PAGE0-direct-addressing operand. Data page is 0. Offset is specified by *6bit*, a 6-bit constant from 0 to 63. PAGE0 direct addressing mode is available only when PAGE0 = 1.
- @*6bit* DP-direct-addressing operand. Data page is specified by the DP. Offset is specified by *6bit*, a 6-bit constant from 0 to 63.
- @**XAR***n* Register-addressing operand. For **XAR***n*, specify XAR6 or XAR7.
- ***–SP[***6bit***]*** PAGE0-stack-addressing operand. Address in data memory is specified as 0:(SP – *6bit*), where 0: indicates that the six MSBs are 0s and *6bit* is a 6-bit constant from 0 to 63. PAGE0 stack addressing mode is available only when PAGE0 = 0.
- ***ind* Indirect-addressing operand. Select one of the following operands (*x* is a number from 0 to 5; *n* is 6 or 7; *z* is a number from 0 to 7; *3bit* is a 3-bit constant):

*SP++	*AR*x*	*XAR*n*	*ARPz
*––SP	*AR*x*++	*XAR*n*++	*
	*––AR*x*	*––XARn	*++
	*+AR*x*[AR0]	*+XARn[AR0]	*––
	*+AR*x*[AR1]	*+XARn[AR1]	*0++
	*+AR*x*[*3bit*]	*+XARn[*3bit*]	*0––
	*AR6%++		

Opcode

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	1	1	See section 5.7.2 on page 5-23.							

Description

The value referenced by *locLong* is subtracted from the content of ACC. The variable *locLong* references either a 32-bit data-memory value or one of the 22-bit extended auxiliary register values (XAR6 or XAR7). If XAR6 or XAR7 is used, the value is treated as an unsigned number. The relationship between ACC and *locLong* is then reflected by the Z and N bits. If the values are the same, the result is 0 (Z = 1). If ACC is greater than *locLong*, the result is positive (N = 0) and nonzero (Z = 0). If ACC is less than *locLong*, the result is negative (N = 1).

The compare operation treats the setting of the N flag different from the SUB operation. For more details, see section 2.3, *Status Register ST0*, on page 2-13.

The ’27x core expects memory wrappers and peripheral-interface logic to force any 32-bit access, like that of the CMPL instruction, to align to an even address. This alignment is described in section 6.2 on page 6-31.

Execution

Change C, N, and Z based on following calculation and output of ALU:

[ACC] – addressed 22/32-bit value → ALU output

[PC] +1 → PC

Status Bits

OVM, SXM	Neither affects the operation.
V, OVC	V and OVC are not affected by the operation.
C	If the subtraction generates a borrow, C is cleared; otherwise, C is set.
N	If the result is negative, N is set; otherwise, N is cleared. The CMP instruction assumes infinite precision when it determines the sign of the result. For example, consider the subtraction $8000\ 0000_{16} – 0000\ 0001_{16}$. If the precision were limited to 32 bits, the result would cause an overflow to the positive number $7FFF\ FFFF_{16}$, and N would be cleared. However, because the CMP instruction assumes infinite precision, it would set N to indicate that $8000\ 0000_{16} – 0000\ 0001_{16}$ actually results in a negative number.
Z	If the result is 0, Z is set; otherwise, Z is cleared.

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Words

1

Example 1

```
CMPL   ACC, @XAR6

; XAR6 treated as unsigned
; [ACC] – [XAR6] = 0000 0002₁₆ – 003f fff4₁₆
```

	Before instruction	After instruction
ACC	0000 0002	0000 0002
XAR6	3f fff4	3f fff4

Before instruction		After instruction	
C = X	N = X	C = 0	N = 1
V = X	Z = X	V unchanged	Z = 0

Example 2

```
CMPL   ACC, @10

; Addressed 32–bit data value treated as signed
; [ACC] – 32-bit value =   0000 0002₁₆ – ffff fff4₁₆
```

	Before instruction	After instruction
DP	0003	0003
ACC	0000 0002	0000 0002

Data memory	Before instruction	After instruction
0000ca	fff4	fff4
0000cb	ffff	ffff

Before instruction		After instruction	
C = X	N = X	C = 0	N = 0
V = X	Z = X	V unchanged	Z = 0

Example 3

```
CMPL   ACC, *AR4++

; Addressed 32–bit data value treated as signed
; Read aligned to even address
; [ACC] – 32-bit value = 8000 0000₁₆ – 1000 0001₁₆
```

	Before instruction	After instruction
AR4	0087	0089
ACC	8000 0000	8000 0000

Data memory	Before instruction	After instruction
000086	0001	0001
000087	1000	1000

Before instruction		After instruction	
ARP = X		ARP = 4	
C = X	N = X	C = 1	N = 1
V = X	Z = X	V unchanged	Z = 0

Syntax

DEC *loc*

Operands

loc Reference to a 16-bit location, either a data-memory location or a 16-bit register. Use any of these types of operands:

- **@0:***6bit* PAGE0-direct-addressing operand. Data page is 0. Offset is specified by *6bit*, a 6-bit constant from 0 to 63. PAGE0 direct addressing mode is available only when PAGE0 = 1.
- @*6bit* DP-direct-addressing operand. Data page is specified by the DP. Offset is specified by *6bit*, a 6-bit constant from 0 to 63.
- @*reg* Register-addressing operand. For *reg*, specify one of these register names:

AH	AL	PL	PH	T	SP
AR0	AR1	AR2	AR3	AR4	AR5
AR6	AR7				

- ***–SP[***6bit***]*** PAGE0-stack-addressing operand. Address in data memory is specified as 0:(SP – *6bit*), where 0: indicates that the six MSBs are 0s and *6bit* is a 6-bit constant from 0 to 63. PAGE0 stack addressing mode is available only when PAGE0 = 0.
- ***ind* Indirect-addressing operand. Select one of the following operands (*x* is a number from 0 to 5; *n* is 6 or 7; *z* is a number from 0 to 7; *3bit* is a 3-bit constant):

*SP++	*AR*x*	*XAR*n*	*ARPz
*––SP	*AR*x*++	*XAR*n*++	*
	*––AR*x*	*––XARn	*++
	*+AR*x*[AR0]	*+XARn[AR0]	*––
	*+AR*x*[AR1]	*+XARn[AR1]	*0++
	*+AR*x*[*3bit*]	*+XARn[*3bit*]	*0––
	*AR6%++		

Opcode

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	1	1	See section 5.7.2 on page 5-23.							

Description

Subtract 1 from the value referenced by *loc*. The referenced value is treated as a signed number. If AR6 or AR7 is used, the six most significant bits of the extended auxiliary register (XAR6 or XAR7) are not affected.

Execution

[addressed location] – 1 → addressed location
[PC] + 1 → PC

Status Bits

OVM, SXM	Neither affects the operation.
C	If the subtraction generates a borrow, C is cleared; otherwise, C is set.
N	If bit 15 of the result is 1, N is set; otherwise, N is cleared.
V	If an overflow occurs, V is set; otherwise, V is not affected.
Z	If the result is 0, Z is set; otherwise, Z is cleared.

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Words

1

Example 1

```
DEC    @7
```

Before instruction		After instruction	
DP	0002	DP	0002
Data memory		Data memory	
000087	0002	000087	0001
C = X	N = X	C = 1	N = 0
V = X	Z = X	V unchanged	Z = 0

Example 2

```
DEC    @T

; The subtraction causes an overflow.
```

Before instruction		After instruction	
T	8000	T	7fff
C = X	N = X	C = 1	N = 0
V = X	Z = X	V = 1	Z = 0

Syntax **EALLOW**

Operands None

Opcode

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	0	1	0	0	0	1	0

Description This instruction sets the EALLOW bit in status register ST1. When this bit is set, the ’27x allows access to the memory-mapped emulation registers.

Execution Enable access to emulation registers:
1 → EALLOW

Status Bits This instruction sets the EALLOW bit in status register ST1.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Words 1

Syntax **EDIS**

Operands None

Opcode

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	0	0	1	1	0	1	0

Description This instruction clears the EALLOW bit in status register ST1. When this bit is cleared, the ’27x does not allow access to the memory-mapped emulation registers.

Execution Disable access to emulation registers:
0 → EALLOW

Status Bits This instruction clears the EALLOW bit in status register ST1.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Words 1

Syntax **ESTOP0**

Operands None

Opcode

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	0	1	0	0	1	0	1

Description This instruction is available for emulation purposes. It is used to create a software breakpoint.

When an emulator is connected to the ’27x and emulation is enabled, this instruction causes the ’27x to halt, regardless of the state of the DBGM bit in status register ST1. In addition, ESTOP0 does not increment the PC.

When an emulator is not connected or when a debug program has disabled emulation, the ESTOP0 instruction is treated the same way as a NOP instruction. It simply advances the PC to the next instruction.

Execution

```
If emulator connected and emulation enabled
    Halt processor.
else
    PC + 1 → PC
```

Status Bits This instruction is neither affected by nor affects ST0 or ST1.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Words 1

Syntax **ESTOP1**

Operands None

Opcode

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	0	1	0	0	1	0	0

Description This instruction is available for emulation purposes. It is used to create an embedded software breakpoint.

When an emulator is connected to the ’27x, this instruction will cause the ’27x to halt, regardless of the state of the DBGM bit in status register ST1. Before halting the processor, ESTOP1 increments the PC so that it points to the instruction following ESTOP1.

When an emulator is not connected or when a debug program has disabled emulation, the ESTOP1 instruction is treated the same way as a NOP instruction. It simply advances the PC to the next instruction.

Execution

If emulator connected and emulation enabled
    PC + 1 → PC
    Halt processor.
else
    PC + 1 → PC

Status Bits This instruction is neither affected by nor affects ST0 or ST1.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Words 1

Syntax **FFC XAR7,** *22BitAddress*

Operands

22BitAddress 22-bit program-memory address from $00\,0000_{16}$ to $3F\,FFFF_{16}$

XAR7 Extended auxiliary register XAR7

Opcode

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	1	1	*22BitAddress*(21:16)					
22BitAddress(15:0)															

Description

PC + 2 (the address of the next instruction) is stored in auxiliary register XAR7. Then the PC is filled with the 22-bit value referenced by *22BitAddress*, forcing program control to that address.

This instruction, in conjunction with the LB *XAR7 instruction, can be used to implement a fast function call and return. Here is an example:

```
FFC  XAR7,FuncA

FuncA:
     .
     .        ; Code for Function A
     .
     LB   *XAR7
```

The CALL instruction stores the return address to the stack, using two stack locations. In addition, the standard CALL-RET combination takes more cycles. The RET must perform two data transfers to restore the PC from the stack, but LB *XAR7 restores the PC with one data transfer from XAR7.

Execution

[PC] + 2 → temp
temp → XAR7
22-bit address → PC

Status Bits

OVM, SXM Neither affects the operation.

C, N, V, Z None of these are affected by the operation.

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Words 2

Example

```
FCC    XAR7, 86h     ; Store return address in XAR7. Call
                     ; subroutine at the given hexadecimal
                     ; address.
```

Syntax **IACK** #*VectorValue*

Operands *VectorValue* 16-bit number from 0000_{16} to $FFFF_{16}$

Opcode

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	0	1	1	1	1	1	1
VectorValue															

Description The specified 16-bit constant value is output on the low 16 bits of the data-write data bus, DWDB(15:0). The IACK signal is asserted externally during the write operation. The IACK instruction does *not* affect the address bus; the address bus contains the address that was last driven on it. This instruction can be placed in an interrupt service routine to inform external hardware that the corresponding interrupt has been acknowledged by the CPU.

Execution Transfer 16-bit value to DWDB(15:0).
[PC] + 2 → PC

Status Bits OVM, SXM Neither affects the operation.

C, N, V, Z None of these are affected by the operation.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Words 2

Example

```
IACK   #5          ; External device translates
                   ; 5 as interrupt INT14.
```

Syntax **IDLE**

Operands None

Opcode

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	0	1	0	0	0	0	1

Description The IDLE instruction causes the following sequence of events:

1) The pipeline is flushed.
2) All outstanding memory cycles are completed.
3) The IDLESTAT bit of status register ST1 is set.
4) Clocks to the CPU are stopped, placing the device in the idle state. In the idle state, CLKOUT (the clock output from the core) and all clocks to blocks outside the core (including the emulation block) continue to operate, as long as CLKIN (the clock input to the core) is driven. The PC continues to hold the address of the IDLE instruction; the PC is *not* incremented before the CPU enters the idle state.
5) The IDLE output core signal is activated (driven high).
6) The device waits for an enabled or nonmaskable hardware interrupt. If such an interrupt occurs, the IDLESTAT bit is cleared, the PC is incremented by 1, and the device exits the idle state.

   If the interrupt is maskable, it must be enabled in the interrupt enable register (IER). However, the device exits the idle state regardless of the value of the interrupt global mask bit (INTM) of status register ST1.

After the device exits the idle state, the CPU must respond to the interrupt request. If the interrupt can be disabled by the INTM bit in status register ST1, the next event depends on INTM. If INTM = 0, the interrupt is enabled, and the CPU executes the corresponding interrupt service routine. On return from the interrupt, execution begins at the instruction following the IDLE instruction. If INTM = 1, the interrupt is blocked and the program immediately continues at the instruction following IDLE.

If the interrupt cannot be disabled by INTM, the CPU executes the corresponding interrupt service routine. On return from the interrupt, execution begins at the instruction following IDLE.

Execution Enter idle state and wait for enabled or nonmaskable interrupt.
If interrupt is received:
[PC] + 1 → PC
Exit idle state.

Status Bits

OVM, SXM	Neither affects the operation.
C, N, V, Z	None of these are affected by the operation.

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Words

1

Syntax **INC** *loc*

Operands

loc Reference to a 16-bit location, either a data-memory location or a 16-bit register. Use any of these types of operands:

- **@0:***6bit* PAGE0-direct-addressing operand. Data page is 0. Offset is specified by *6bit*, a 6-bit constant from 0 to 63. PAGE0 direct addressing mode is available only when PAGE0 = 1.
- @*6bit* DP-direct-addressing operand. Data page is specified by the DP. Offset is specified by *6bit*, a 6-bit constant from 0 to 63.
- @*reg* Register-addressing operand. For *reg*, specify one of these register names:

AH	AL	PL	PH	T	SP
AR0	AR1	AR2	AR3	AR4	AR5
AR6	AR7				

- ***–SP[***6bit***]*** PAGE0-stack-addressing operand. Address in data memory is specified as 0:(SP – *6bit*), where 0: indicates that the six MSBs are 0s and *6bit* is a 6-bit constant from 0 to 63. PAGE0 stack addressing mode is available only when PAGE0 = 0.
- ***ind* Indirect-addressing operand. Select one of the following operands (*x* is a number from 0 to 5; *n* is 6 or 7; *z* is a number from 0 to 7; *3bit* is a 3-bit constant):

*SP++	*AR*x*	*XAR*n*	*ARPz
*––SP	*AR*x*++	*XAR*n*++	*
	*––AR*x*	*––XARn	*++
	*+AR*x*[AR0]	*+XARn[AR0]	*––
	*+AR*x*[AR1]	*+XARn[AR1]	*0++
	*+AR*x*[*3bit*]	*+XARn[*3bit*]	*0––
	*AR6%++		

Opcode

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	1	0	See section 5.7.2 on page 5-23.							

Description Add 1 to the value referenced by *loc*. The referenced value is treated as a signed number. If AR6 or AR7 is used, the six most significant bits of the extended auxiliary register (XAR6 or XAR7) are not affected.

Execution [addressed location] + 1 → addressed location
[PC] + 1 → PC

Status Bits

OVM, SXM	Neither affects the operation.
C	If the addition generates a carry, C is set; otherwise, C is cleared.
N	If bit 15 of the result is 1, N is set; otherwise, N is cleared.
V	If an overflow occurs, V is set; otherwise, V is not affected.
Z	If the result is 0, Z is set; otherwise, Z is cleared.

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Words

1

Example 1

```
INC    @7
```

Before instruction		After instruction	
DP	0002	DP	0002
Data memory		Data memory	
000087	0002	000087	0003
C = X	N = X	C = 0	N = 0
V = X	Z = X	V unchanged	Z = 0

Example 2

```
INC    @T

; The addition causes an overflow.
```

Before instruction		After instruction	
T	7fff	T	8000
C = X	N = X	C = 0	N = 1
V = X	Z = X	V = 1	Z = 0

Syntax

1: **INTR** **INT***i*
2: **INTR** **DLOGINT**
3: **INTR** **RTOSINT**
4: **INTR** **NMI**

Operands

DLOGINT Data log interrupt vector

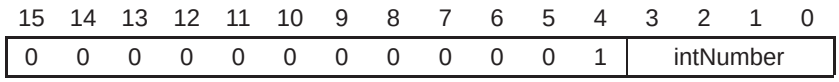
INT*i* Interrupt vector name. *i* ranges from 1 to 14.

NMI Nonmaskable interrupt vector

RTOSINT Real-time operating system interrupt vector

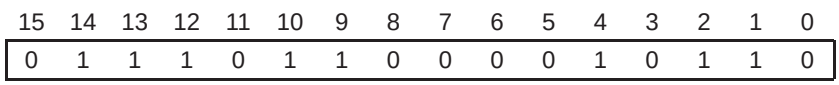
Opcode

Syntaxes 1, 2, and 3:



Interrupt Vector	intNumber	Interrupt Vector	intNumber
INT1	1	INT9	9
INT2	2	INT10	10
INT3	3	INT11	11
INT4	4	INT12	12
INT5	5	INT13	13
INT6	6	INT14	14
INT7	7	DLOGINT	
INT8	8	RTOSINT	

Syntax 4: **INTR** **NMI**



Description

The INTR instruction transfers program control to the interrupt service routine that corresponds to the vector specified in the instruction. The INTR instruction is not affected by the INTM bit in status register ST1. It is also not affected by enable bits the interrupt enable register (IER) or the debug interrupt enable register (DBGIER). Once the INTR instruction reaches the decode 2 phase of the pipeline, hardware interrupts cannot be serviced until the INTR instruction is finished executing (until the interrupt service routine begins).

Part of the operation involves saving pairs of 16-bit CPU registers. Each pair of registers is saved in a single 32-bit operation. The register forming the low word of the pair is saved first (to an even address); the register forming the high word of the pair is saved next (to the following odd address). For example, the first value saved is the concatenation of the T register and status register ST0 (T:ST0). ST0 is saved first, then T.

Execution

For INT*i*, DLOGINT, and RTOSINT:

Clear corresponding IFR bit:
    0 → IFR(intNumber)
Empty the pipeline.
Increment and temporarily store PC:
    [PC] + 1 → PC
    [PC] → temp
Fetch specified vector.
Increment stack pointer:
    [SP] + 1 → SP
Save register pairs:
    [T:ST0] → addresses referenced by SP
    [SP] + 2 → SP
    [AH:AL] → addresses referenced by SP
    [SP] + 2 → SP
    [PH:PL] → addresses referenced by SP
    [SP] + 2 → SP
    [AR1:AR0] → addresses referenced by SP
    [SP] + 2 → SP
    [DP:ST1] → addresses referenced by SP
    [SP] + 2 → SP
    [DBGSTAT:IER] → addresses referenced by SP
    [SP] + 2 → SP

Save return address:
  temp → addresses referenced by SP
  [SP] + 2 → SP
Clear corresponding IER bit:
  0 → IER(intNumber)
Disable interrupts $\overline{\text{INT1}}$–$\overline{\text{INT14}}$, DLOGINT, and RTOSINT:
  1 → INTM
Disable debug events:
  1 → DBGM
Disable access to emulation registers:
  0 → EALLOW
Clear LOOP and IDLESTAT flags:
  0 → LOOP
  0 → IDLESTAT
Load PC with fetched vector.
  vector → PC

For NMI:

Empty the pipeline.
Increment and temporarily store PC:
  [PC] + 1 → PC
  [PC] → temp
Fetch specified vector.
Increment stack pointer:
  [SP] + 1 → SP
Save register values:
  [T:ST0] → addresses referenced by SP
  [SP] + 2 → SP
  [AH:AL] → addresses referenced by SP
  [SP] + 2 → SP
  [PH:PL] → addresses referenced by SP
  [SP] + 2 → SP
  [AR1:AR0] → addresses referenced by SP
  [SP] + 2 → SP
  [DP:ST1] → addresses referenced by SP
  [SP] + 2 → SP
  [DBGSTAT:IER] → addresses referenced by SP
  [SP] + 2 → SP

Save return address:
temp → addresses referenced by SP
[SP] + 2 → SP
Disable interrupts $\overline{\text{INT1}}$–$\overline{\text{INT14}}$, DLOGINT, and RTOSINT:
1 → INTM
Disable debug events:
1 → DBGM
Disable access to emulation registers:
0 → EALLOW
Clear LOOP and IDLESTAT flags:
0 → LOOP
0 → IDLESTAT
Load PC with fetched vector.
vector → PC

Status Bits

OVM, SXM	Neither affects the operation.
C, N, V, Z	None of these are affected by the operation.
DBGM, INTM	Both are set by the operation. This disables debug events (DBGM = 1) and disables maskable interrupts (INTM = 1).
EALLOW, LOOP, IDLESTAT	All three are cleared by the operation. Clearing EALLOW disables access to emulation registers.

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Words

1

Example

```
INTR   INT3
```

Before instruction

IER	0004
IFR	0004
SP	00efff

Data memory

00efff	0000
00f000	0000
00f001	0000
00f002	0000
00f003	0000
00f004	0000
00f005	0000
00f006	0000
00f007	0000
00f008	0000
00f009	0000
00f00a	0000
00f00b	0000
00f00c	0000
00f00d	0000
00f00e	0000

```
INTM = X    DBGM = X
LOOP = X
EALLOW = X
IDLESTAT = X
```

After instruction

IER	0000
IFR	0000
SP	f00e

Data memory

00efff	0000	
00f000	[ST0]	
00f001	[T]	
00f002	[AL]	
00f003	[AH]	
00f004	[PL]	
00f005	[PH]	
00f006	[AR0]	
00f007	[AR1]	
00f008	[ST1]	
00f009	[DP]	
00f00a	[IER]	
00f00b	[DBGSTAT]	
00f00c	return address	(low half)
00f00d	return address	(high half)
00f00e	0000	

```
INTM = 1     DBGM = 1
LOOP = 0
EALLOW = 0
IDLESTAT = 0
```

Syntax **IRET**

Operands None

Opcode

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	0	0	0	0	0	1	0

Description The IRET instruction restores the PC value and other register values that were saved by an interrupt operation. The order in which the values are restored is opposite the order in which they were saved. The stack pointer is not forced to align to even addresses during the register restore operations.

Interrupts cannot be serviced from the time the IRET instruction reaches the decode 2 phase of the pipeline to the time it completes execution.

The ’27x core expects memory wrappers or peripheral-interface logic to force any 32-bit access, like that of the IRET instruction, to align to an even address. This alignment is described in section 6.2 on page 6-31.

Execution

Pop return address from stack:
[SP] – 2 → SP
[addresses referenced by SP] → PC
Restore register values:
[SP] – 2 → SP
[addresses referenced by SP] → IER then DBGSTAT
[SP] – 2 → SP
[addresses referenced by SP] → ST1 then DP
[SP] – 2 → SP
[addresses referenced by SP] → AR0 then AR1
[SP] – 2 → SP
[addresses referenced by SP] → PL then PH
[SP] – 2 → SP
[addresses referenced by SP] → AL then AH
[SP] – 2 → SP
[addresses referenced by SP] → ST0 then T
Decrement stack pointer:
[SP] – 1 → SP

Status Bits Neither OVM nor SXM affects the operation. The operation affects the following bit values of ST0: OVC, PM, V, N, Z, C, TC, OVM, and SXM. It also affects the following bit values of ST1: ARP, SPA, VMAP, PAGE0, DBGM, and INTM.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Words 1

Syntax **ITRAP0**

Operands None

Opcode

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

Description The ITRAP0 instruction causes an illegal-instruction trap, which operates like the TRAP #19 instruction. The ILLEGAL interrupt vector is fetched and the corresponding interrupt service routine is executed.

If the ’27x reads 0000_{16} from program memory, it generates the ITRAP0 instruction. If the ’27x reads $FFFF_{16}$, it generates the ITRAP1 instruction.

Execution

Empty the pipeline.
Increment and temporarily store PC:
    [PC] + 1 → PC
    [PC] → temp
Fetch ILLEGAL vector.
Increment stack pointer:
    [SP] + 1 → SP
Save register values:
    [T:ST0] → addresses referenced by SP
    [SP] + 2 → SP
    [AH:AL] → addresses referenced by SP
    [SP] + 2 → SP
    [PH:PL] → addresses referenced by SP
    [SP] + 2 → SP
    [AR1:AR0] → addresses referenced by SP
    [SP] + 2 → SP
    [DP:ST1] → addresses referenced by SP
    [SP] + 2 → SP
    [DBGSTAT:IER] → addresses referenced by SP
    [SP] + 2 → SP

Save return address:
temp → addresses referenced by SP
[SP] + 2 → SP
Disable interrupts $\overline{\text{INT1}}$–$\overline{\text{INT14}}$, DLOGINT, and RTOSINT:
1 → INTM
Disable debug events:
1 → DBGM
Disable access to emulation registers:
0 → EALLOW
Clear LOOP and IDLESTAT flags:
0 → LOOP
0 → IDLESTAT
Load PC with ILLEGAL vector.

Status Bits

OVM, SXM Neither affects the operation.

C, N, V, Z None of these are affected by the operation.

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Words

1

Syntax **ITRAP1**

Operands None

Opcode

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

Description The ITRAP1 instruction causes an illegal-instruction trap, which operates like the TRAP #19 instruction. The ILLEGAL interrupt vector is fetched and the corresponding interrupt service routine is executed.

If the ’27x reads $FFFF_{16}$ from program memory, it generates the ITRAP1 instruction. If the ’27x reads 0000_{16}, it generates the ITRAP0 instruction.

Execution

Empty the pipeline.
Increment and temporarily store PC:
    [PC] + 1 → PC
    [PC] → temp
Fetch ILLEGAL vector.
Increment stack pointer:
    [SP] + 1 → SP
Save register values:
    [T:ST0] → addresses referenced by SP
    [SP] + 2 → SP
    [AH:AL] → addresses referenced by SP
    [SP] + 2 → SP
    [PH:PL] → addresses referenced by SP
    [SP] + 2 → SP
    [AR1:AR0] → addresses referenced by SP
    [SP] + 2 → SP
    [DP:ST1] → addresses referenced by SP
    [SP] + 2 → SP
    [DBGSTAT:IER] → addresses referenced by SP
    [SP] + 2 → SP

Save return address:
temp → addresses referenced by SP
[SP] + 2 → SP
Disable interrupts $\overline{\text{INT1}}$–$\overline{\text{INT14}}$, DLOGINT, and RTOSINT:
1 → INTM
Disable debug events:
1 → DBGM
Disable access to emulation registers:
0 → EALLOW
Clear LOOP and IDLESTAT flags:
0 → LOOP
0 → IDLESTAT
Load PC with ILLEGAL vector.

Status Bits

OVM, SXM Neither affects the operation.

C, N, V, Z None of these are affected by the operation.

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Words

1

Syntax

1: **LB** *22BitAddress*

2: **LB** ***XAR7**

Operands

22BitAddress 22-bit program-memory address from $00\,0000_{16}$ to $3F\,FFFF_{16}$

XAR7 Extended auxiliary register XAR7

Opcode

Syntax 1: **LB** *22BitAddress*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	1	*22BitAddress*(21:16)					
22BitAddress(15:0)															

Syntax 2: **LB** ***XAR7**

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	0	1	0	0	0	0	0

Description

The LB instruction causes a branch by loading the PC with a 22-bit value. This 22-bit value is supplied in the instruction (syntax 1) or is taken from extended auxiliary register XAR7 (syntax 2).

Execution

Syntax 1: **LB** *22BitAddress*

22-bit constant → PC

Syntax 2: **LB** ***XAR7**

[XAR7] → PC

Status Bits

OVM, SXM Neither affects the operation.

C, N, V, Z None of these are affected by the operation.

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Words

Syntax 1: 2

Syntax 2: 1

Example 1

```
LB     86h      ; Branch to the given hexadecimal address.
```

Example 2

```
LB     *XAR7    ; Branch to the address in XAR7.
```

Syntax **LOOPNZ** *loc*, #*16BitMask*

Operands

16BitMask 16-bit mask value from 0000_{16} to $FFFF_{16}$

loc Reference to a 16-bit location, either a data-memory location or a 16-bit register. Use any of these types of operands:

@0:*6bit* PAGE0-direct-addressing operand. Data page is 0. Offset is specified by *6bit*, a 6-bit constant from 0 to 63. PAGE0 direct addressing mode is available only when PAGE0 = 1.

@*6bit* DP-direct-addressing operand. Data page is specified by the DP. Offset is specified by *6bit*, a 6-bit constant from 0 to 63.

@*reg* Register-addressing operand. For *reg*, specify one of these register names:

AH	AL	PL	PH	T	SP
AR0	AR1	AR2	AR3	AR4	AR5
AR6	AR7				

–SP[6bit***]** PAGE0-stack-addressing operand. Address in data memory is specified as 0:(SP – *6bit*), where 0: indicates that the six MSBs are 0s and *6bit* is a 6-bit constant from 0 to 63. PAGE0 stack addressing mode is available only when PAGE0 = 0.

*****_ind_ Indirect-addressing operand. Select one of the following operands (*x* is a number from 0 to 5; *n* is 6 or 7; *z* is a number from 0 to 7; *3bit* is a 3-bit constant):

*SP++	*AR*x*	*XAR*n*	*ARPz
*––SP	*AR*x*++	*XAR*n*++	*
	*––AR*x*	*––XARn	*++
	*+AR*x*[AR0]	*+XARn[AR0]	*––
	*+AR*x*[AR1]	*+XARn[AR1]	*0++
	*+AR*x*[*3bit*]	*+XARn[*3bit*]	*0––
	*AR6%++		

Opcode

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	1	1	1	0	See section 5.7.2 on page 5-23.							
16BitMask															

Description

The LOOPNZ instruction uses an AND operation to compare the value referenced by *loc* and the 16-bit mask value. The instruction performs this comparison repeatedly for as long as the result of the operation is not 0. The process is described as follows:

1) Set the LOOP bit in status register ST1.

2) Generate the address for the value referenced by *loc*.

3) If *loc* is an indirect-addressing operand, perform any specified modification to the SP or to the specified auxiliary register and/or the ARP.

4) Compare the addressed value with the mask value by using an AND operation.

5) If the result is 0, clear the LOOP bit, and increment the PC by 2. If the result is not 0, return to step 1.

The loop created by steps 1 through 5 can be interrupted by hardware interrupts. When an interrupt occurs, if the LOOPNZ instruction is still active, the return address saved on the stack points to the LOOPNZ instruction. Therefore, upon return from the interrupt, the LOOPNZ instruction is fetched again.

While the result of the AND operation is not 0, the LOOPNZ instruction begins again every five cycles in the decode 2 phase of the pipeline. Thus, the memory location or register is read once every five cycles. If you use an indirect addressing mode for *loc*, you can specify an increment or decrement for the pointer (SP or an auxiliary register). If you do, the pointer is modified each time in the decode 2 phase of the pipeline. This means that the mask value is compared with a new data-memory value each time.

The LOOPNZ instruction does not flush prefetched instructions from the pipeline. However, when an interrupt occurs, prefetched instructions are flushed.

When any interrupt occurs, the current state of the LOOP bit is saved as ST1 is saved on the stack. The LOOP bit in ST1 is then cleared by the interrupt. The LOOP bit is a passive status bit. The LOOPNZ instruction changes LOOP, but LOOP does not affect the instruction.

You can abort the LOOPNZ instruction within an interrupt service routine. Test the LOOP bit saved on the stack. If it is set, then increment (by 2) the return address on the stack. Upon return from the interrupt, this incremented address is loaded into the PC, and the instruction following LOOPNZ is executed.

Execution

While ([addressed location] AND 16-bit mask value) ≠ 0
    1 → LOOP
0 → LOOP
[PC] + 2 → PC

The while loop can be interrupted by hardware interrupts. The CPU refetches the LOOPNZ instruction after the interrupt is serviced.

Status Bits

OVM, SXM	Neither affects the operation.
C, V	Neither of these is affected by the operation.
N, Z	Both of these are affected by the AND operation that is performed by the LOOPNZ instruction: If bit 15 of the result is 1, N is set; otherwise, N is cleared. If the result is 0, Z is set; otherwise, Z is cleared.
LOOP	LOOP is repeatedly set while the result of the AND operation is not 0. LOOP is cleared when the result is 0. If an interrupt occurs before the LOOPNZ instruction enters the decode 2 phase of the pipeline, the instruction is flushed from the pipeline and, thus, does not affect the LOOP bit.

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Words

2

Syntax **LOOPZ** *loc*, #*16BitMask*

Operands

16BitMask 16-bit mask value from 0000_{16} to $FFFF_{16}$

loc Reference to a 16-bit location, either a data-memory location or a 16-bit register. Use any of these types of operands:

- **@0:***6bit* PAGE0-direct-addressing operand. Data page is 0. Offset is specified by *6bit*, a 6-bit constant from 0 to 63. PAGE0 direct addressing mode is available only when PAGE0 = 1.
- @*6bit* DP-direct-addressing operand. Data page is specified by the DP. Offset is specified by *6bit*, a 6-bit constant from 0 to 63.
- @*reg* Register-addressing operand. For *reg*, specify one of these register names:

AH	AL	PL	PH	T	SP
AR0	AR1	AR2	AR3	AR4	AR5
AR6	AR7				

- ***–SP[***6bit***]*** PAGE0-stack-addressing operand. Address in data memory is specified as 0:(SP – *6bit*), where 0: indicates that the six MSBs are 0s and *6bit* is a 6-bit constant from 0 to 63. PAGE0 stack addressing mode is available only when PAGE0 = 0.
- ***ind* Indirect-addressing operand. Select one of the following operands (*x* is a number from 0 to 5; *n* is 6 or 7; *z* is a number from 0 to 7; *3bit* is a 3-bit constant):

*SP++	*AR*x*	*XAR*n*	*ARPz
*––SP	*AR*x*++	*XAR*n*++	*
	*––AR*x*	*––XARn	*++
	*+AR*x*[AR0]	*+XARn[AR0]	*––
	*+AR*x*[AR1]	*+XARn[AR1]	*0++
	*+AR*x*[*3bit*]	*+XARn[*3bit*]	*0––
	*AR6%++		

Opcode

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	1	1	0	0	See section 5.7.2 on page 5-23.							
16BitMask															

Description

The LOOPZ instruction uses an AND operation to compare the value referenced by *loc* and the 16-bit mask value. The instruction performs this comparison repeatedly for as long as the result of the operation is 0. The process can be described as follows:

1) Set the LOOP bit in status register ST1.

2) Generate the address for the value referenced by *loc*.

3) If *loc* is an indirect-addressing operand, perform any specified modification to the SP or to the specified auxiliary register and/or the ARP.

4) Compare the addressed value with the mask value by using an AND operation.

5) If the result is not 0, clear the LOOP bit, and increment the PC by 2. If the result is 0, return to step 1.

The loop created by steps 1 through 5 can be interrupted by hardware interrupts. When an interrupt occurs, if the LOOPZ instruction is still active, the return address saved on the stack points to the LOOPZ instruction. Therefore, upon return from the interrupt, the LOOPZ instruction is fetched again.

While the result of the AND operation is 0, the LOOPZ instruction begins again every five cycles in the decode 2 phase of the pipeline. Thus, the memory location or register is read once every five cycles. If you use an indirect addressing mode for *loc*, you can specify an increment or decrement for the pointer (SP or an auxiliary register). If you do, the pointer is modified each time in the decode 2 phase of the pipeline. This means that the mask value is compared with a new data-memory value each time.

The LOOPZ instruction does not flush prefetched instructions from the pipeline. However, when an interrupt occurs, prefetched instructions are flushed.

When any interrupt occurs, the current state of the LOOP bit is saved as ST1 is saved on the stack. The LOOP bit in ST1 is then cleared by the interrupt. The LOOP bit is a passive status bit. The LOOPZ instruction changes LOOP, but LOOP does not affect the instruction.

You can abort the LOOPZ instruction within an interrupt service routine. Test the LOOP bit saved on the stack. If it is set, then increment (by 2) the return address on the stack. Upon return from the interrupt, this incremented address is loaded into the PC, and the instruction following LOOPZ is executed.

Execution

While ([addressed location] AND 16-bit mask value) = 0
    1 → LOOP
0 → LOOP
[PC] + 2 → PC

The while loop can be interrupted by hardware interrupts. The CPU refetches the LOOPZ instruction after the interrupt is serviced.

Status Bits

OVM, SXM	Neither affects the operation.
C, V	Neither of these is affected by the operation.
N, Z	Both of these are affected by the AND operation that is performed by the LOOPZ instruction: If bit 15 of the result is 1, N is set; otherwise, N is cleared. If the result is 0, Z is set; otherwise, Z is cleared.
LOOP	LOOP is repeatedly set while the result of the AND operation is 0. LOOP is cleared when the result is not 0. If an interrupt occurs before the LOOPZ instruction enters the decode 2 phase of the pipeline, the instruction is flushed from the pipeline and, thus, does not affect the LOOP bit.

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Words

2

See section 8.4 on page 8-17 for more information about and examples of the LOOPZ and LOOPNZ instructions.

Syntax

1: **LSL** **A***X*, *shift*

2: **LSL** **A***X*, **T**

3: **LSL** **ACC,** *shift*

4: **LSL** **ACC, T**

Operands

ACC Accumulator

A*X* Use AH (the high word of the accumulator) or AL (the low word of the accumulator).

shift Number from 1 to 16

T Multiplicand register

Opcode

Syntax 1: **LSL** **A***X*, *shift*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	0	0	AX	(*shift* –1)			

Note: If AL is used in the instruction, AX = 0; if AH is used, AX = 1.

Syntax 2: **LSL** **A***X*, **T**

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

Note: If AL is used in the instruction, AX = 0; if AH is used, AX = 1.

Syntax 3: **LSL** **ACC,** *shift*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	0	0	1	1	(*shift* –1)			

Syntax 4: **LSL** **ACC, T**

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

Description

The content of AH, AL, or ACC is shifted left by an amount specified by *shift* or by the four least significant bits (LSBs) of the T register, T(3:0):

- ***shift*.** Specify a constant from 1 to 16 as the second operand of the instruction. During the shift operation, the low-order bits are zero filled. In addition, the last bit to be shifted out of the register is stored in the carry bit (C).
- **T(3:0).** Specify T as the second operand of the instruction. The four LSBs of T enable a shift from 0 to 15. During the shift operation, the low-order bits are zero filled. If the T register specifies a shift of 0, C is cleared; otherwise, C is filled with the last bit to be shifted out of AH/AL/ACC.

Execution

Syntax 1: **LSL A*X*, *shift***

[AX(16 – shift)] → C
[AX] shifted left → AX
[PC] + 1 → PC

During shift, low-order bits are zero filled.

Syntax 2: **LSL A*X*, T**

```
shift = T(3:0)
If shift = 0
    0 → C
    [AX] → AX
    [PC] + 1 → PC
else
    [AX(16 – shift)] → C
    [AX] shifted left → AX
    [PC] + 1 → PC
```

During shift, low-order bits are zero filled.

Syntax 3: **LSL ACC,** *shift*

[ACC(32 – shift)] → C
[ACC] shifted left → ACC
[PC] + 1 → PC

During shift, low-order bits are zero filled.

Syntax 4: **LSL ACC, T**

shift = T(3:0)
If shift = 0
    0 → C
    [ACC] → ACC
    [PC] + 1 → PC
else
    [ACC(32 – shift)] → C
    [ACC] shifted left → ACC
    [PC] + 1 → PC

During shift, low-order bits are zero filled.

Status Bits

Syntaxes 1 and 3 (shift specified in instruction):

OVM, SXM	Neither affects the operation.
V, OVC	V and OVC are not affected by any LSL operation.
C	The last bit to be shifted out of AH/AL/ACC is stored in C.
N	If the most significant bit of the result is 1, N is set; otherwise, N is cleared.
Z	If the result is 0, Z is set; otherwise, Z is cleared.

Syntaxes 2 and 4 (shift specified by T(3:0)):

OVM, SXM	Neither affects the operation.
V, OVC	V and OVC are not affected by any LSL operation.
C	If the T register specifies a shift of 0, C is cleared; otherwise, the last bit to be shifted out of AH or AL is stored in C.
N	If the MSB of the result is 1, N is set; otherwise, N is cleared. Even if the T register specifies a shift of 0, the value of AH/AL/ACC is tested for the negative condition, and N is affected.
Z	If the result is 0, Z is set; otherwise, Z is cleared. Even if the T register specifies a shift of 0, the value of AH/AL/ACC is tested for the zero condition, and Z is affected.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Words 1

Example 1

```
LSL    AL, 4
```

Before instruction		After instruction	
ACC	dddd 1800	ACC	dddd 8000
C = X	N = X	C = 1	N = 1
V = X	Z = X	V unchanged	Z = 0

Example 2

```
LSL    ACC, T          ; T(3:0) holds shift count

; For T(3:0) > 0
```

Before instruction		After instruction	
ACC	1400 dddd	ACC	400d ddd0
T	aaa4	T	aaa4
C = X	N = X	C = 1	N = 0
V = X	Z = X	V unchanged	Z = 0

```
; For T(3:0) = 0
```

Before instruction		After instruction	
ACC	1400 dddd	ACC	1400 dddd
T	aaa0	T	aaa0
C = X	N = X	C = 0	N = 0
V = X	Z = X	V unchanged	Z = 0

Syntax

1: **LSR** **A***X*, *shift*

2: **LSR** **A***X*, **T**

Operands

A*X* Use AH (the high word of the accumulator) or AL (the low word of the accumulator).

shift Number from 1 to 16

T Multiplicand register

Opcode

Syntax 1: **LSR** **A***X*, *shift*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	1	0	AX	(*shift* –1)			

Note: If AL is used in the instruction, AX = 0; if AH is used, AX = 1.

Syntax 2: **LSR** **A***X*, **T**

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

Note: If AL is used in the instruction, AX = 0; if AH is used, AX = 1.

Description

The content of AH or AL is shifted right by an amount specified by *shift* or by the four least significant bits (LSBs) of the T register, T(3:0):

- ***shift.*** Specify a constant from 1 to 16 as the second operand of the instruction. During the shift operation, the high-order bits are zero filled. In addition, the last bit to be shifted out of the register is stored in the carry bit (C).
- **T(3:0).** Specify T as the second operand of the instruction. The four LSBs of T enable a shift from 0 to 15. During the shift operation, the high-order bits are zero filled. If the T register specifies a shift of 0, C is cleared; otherwise, C is filled with the last bit to be shifted out of AH or AL.

The AH or AL value is *not* sign extended during the shift; instead, the high-order bits are zero filled. If you need to perform this kind of shift with sign extension, use the arithmetic shift right (ASR) instruction.

Execution

Syntax 1: **LSR** **A***X*, *shift*

[AX(shift – 1)] → C
[AX] shifted right → AX
[PC] + 1 → PC

During shift, high-order bits are zero filled.

Syntax 2: **LSR A*X*, T**

shift = T(3:0)
If shift = 0
    0 → C
    [AX] → AX
    [PC] + 1 → PC
else
    [AX(shift – 1)] → C
    [AX] shifted right → AX
    [PC] + 1 → PC

During shift, high-order bits are zero filled.

Status Bits

Syntax 1: **LSR A*X*,** *shift*

OVM, SXM	Neither affects the operation.
C	The last bit to be shifted out of AH or AL is stored in C.
N	If bit 15 of the result is 1, N is set; otherwise, N is cleared.
V	V is not affected by the operation.
Z	If the result is 0, Z is set; otherwise, Z is cleared.

Syntax 2: **LSR A*X*, T**

OVM, SXM	Neither affects the operation.
C	If the T register specifies a shift of 0, C is cleared; otherwise, the last bit to be shifted out of AH or AL is stored in C.
N	If bit 15 of the result is 1, N is set; otherwise, N is cleared. Even if the T register specifies a shift of 0, the value of AH or AL is tested for the negative condition, and N is affected.
V	V is not affected by the operation.
Z	If the result is 0, Z is set; otherwise, Z is cleared. Even if the T register specifies a shift of 0, the value of AH or AL is tested for the zero condition, and Z is affected.

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Words

1

Example 1

```
LSR    AL, 8
```

Before instruction

ACC	dddd 8888

```
C = X      N = X
V = X      Z = X
```

After instruction

ACC	dddd 0088

```
C = 1          N = 0
V unchanged    Z = 0
```

Example 2

```
LSR    AH, T          ; T(3:0) holds shift count
;For T(3:0) > 0
```

Before instruction

ACC	0488 dddd
T	aaa8

```
C = X      N = X
V = X      Z = X
```

After instruction

ACC	0004 dddd
T	aaa8

```
C = 1          N = 0
V unchanged    Z = 0
;For T(3:0) = 0
```

Before instruction

ACC	0488 dddd
T	aaa0

```
C = X      N = X
V = X      Z = X
```

After instruction

ACC	0488 dddd
T	aaa0

```
C = 0          N = 0
V unchanged    Z = 0
```

Syntax

MAC **P,** *loc*, **0:***pmem*

Operands

loc Reference to a 16-bit location, either a data-memory location or a 16-bit register. Use any of these types of operands:

- **@0:***6bit* PAGE0-direct-addressing operand. Data page is 0. Offset is specified by *6bit*, a 6-bit constant from 0 to 63. PAGE0 direct addressing mode is available only when PAGE0 = 1.
- @*6bit* DP-direct-addressing operand. Data page is specified by the DP. Offset is specified by *6bit*, a 6-bit constant from 0 to 63.
- @*reg* Register-addressing operand. For *reg*, specify one of these register names:

AH	AL	PL	PH	T	SP
AR0	AR1	AR2	AR3	AR4	AR5
AR6	AR7				

- ***–SP[***6bit***]*** PAGE0-stack-addressing operand. Address in data memory is specified as 0:(SP – *6bit*), where 0: indicates that the six MSBs are 0s and *6bit* is a 6-bit constant from 0 to 63. PAGE0 stack addressing mode is available only when PAGE0 = 0.
- ***ind*** Indirect-addressing operand. Select one of the following operands (*x* is a number from 0 to 5; *n* is 6 or 7; *z* is a number from 0 to 7; *3bit* is a 3-bit constant):

*SP++	*AR*x*	*XAR*n*	*ARPz
*––SP	*AR*x*++	*XAR*n*++	*
	*––AR*x*	*––XARn	*++
	*+AR*x*[AR0]	*+XARn[AR0]	*––
	*+AR*x*[AR1]	*+XARn[AR1]	*0++
	*+AR*x*[*3bit*]	*+XARn[*3bit*]	*0––
	*AR6%++		

P Product register

pmem 16-bit number between 0000_{16} and $FFFF_{16}$ representing the 16 LSBs of a program-memory address. The six MSBs are 0s.

Opcode

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	1	0	0	See section 5.7.2 on page 5-23.							
pmem															

Description

The MAC instruction:

- ❑ Adds the previous product (in the P register), 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 T register with the content of the addressed data-memory or register location.
- ❑ Multiplies the data-memory or register value in the T register by the contents of the addressed program-memory location. The multiplicands are treated as signed numbers. The result is placed in the P register.

Execution

[ACC] + ([P] shifted as per PM bits) $\rightarrow$ ACC
[addressed data-memory or register location] $\rightarrow$ T;
[T] $\times$ [addressed program-memory location] $\rightarrow$ P
[PC] + 2 $\rightarrow$ PC

The multiplicands are treated as signed numbers.

Status Bits

OVM, OVC — If the operation causes ACC to overflow, ACC and OVC depend on OVM:

- OVM = 0 — ACC is not filled with a saturation value. OVC is affected as follows:

  If the overflow was in the negative direction, OVC is decremented by 1; if the overflow was in the positive direction, OVC is incremented by 1.

- OVM = 1 — If the overflow was in the positive direction, ACC is filled with 7FFF FFFF$_{16}$. If the overflow was in the negative direction, ACC is filled with 8000 0000$_{16}$.

  OVC is not affected.

SXM — SXM does not affect the operation.

PM — The P value is shifted as defined by PM before it is added to ACC. (P itself is not modified.)

C — If the addition of P to ACC generates a carry, C is set; otherwise, C is cleared.

N — After the addition of P to ACC, if bit 31 of ACC is 1, N is set; otherwise, N is cleared.

V — If the addition of P to ACC causes ACC to overflow, V is set; otherwise, V is not affected.

Z — If the addition of P to ACC results in ACC = 0, Z is set; otherwise, Z is cleared.

Repeat

This instruction is repeatable. See the description for the RPT instruction.

When the MAC instruction is repeated, the program-memory address contained in the PC is incremented by 1 during each repetition. Thus, a new program-memory multiplicand is used each repetition. If you use indirect addressing to specify the other multiplicand (in data memory), new values can be accessed for this multiplicand as well. Otherwise, this multiplicand is a constant throughout the repeat operation. For example, if a register is the source of one multiplicand, that same register value is accessed during every repetition.

Once the repeat loop begins, it cannot be interrupted by any interrupt.

Words

2

Example 1

```
MAC    P, @10, 0:1000h

; Product shift mode is PM = 3 (right shift by 2, sign extend)
; ACC = [ACC] + ([P] >> 2) = 7fff ffff16 + 0000 000216
; P    = [a16] × (value at 00 100016) = a16 × 216 = 1416
```

	Before instruction	After instruction
DP	0003	0003
P	0000 0008	0000 0014
T	0001	000a
Data memory 0000ca	000a	000a
Program memory 001000	0002	0002

```
; For OVM = 0: ACC overflows normally. OVC records overflow.
```

ACC	7fff ffff	ACC	8000 0001
	OVC = 0		OVC = 1
	C = X N = X		C = 0 N = 1
	V = X Z = X		V = 1 Z = 0

```
; For OVM = 1: ACC filled with saturation value
```

ACC	7fff ffff	ACC	7fff ffff
	OVC = X		OVC unchanged
	C = X N = X		C = 0 N = 0
	V = X Z = X		V = 1 Z = 0

Example 2

```
RPT#2 || MAC   P, *AR2++, 0:1000h

; Product shift mode is PM = 1 (no shift)
; ACC = 0 + [P] + (a x 3) + (9 x 2) = 0 + 8 + 1e + 12 = 38
; P = 8, then (a x 3), then (9 x 2), then (8 x 1)
; Status bit changes associated with final ACC operation
```

Before instruction

AR2	00c9
ACC	0000 0000
P	0000 0008
T	0001

Data memory

0000c9	000a
0000ca	0009
0000cb	0008

Program memory

001000	0003
001001	0002
001002	0001

```
ARP = X
C = X       N = X
V = X       Z = X
```

After instruction

AR2	00cc
ACC	0000 0038
P	0000 0008
T	0008

Data memory

0000c9	000a
0000ca	0009
0000cb	0008

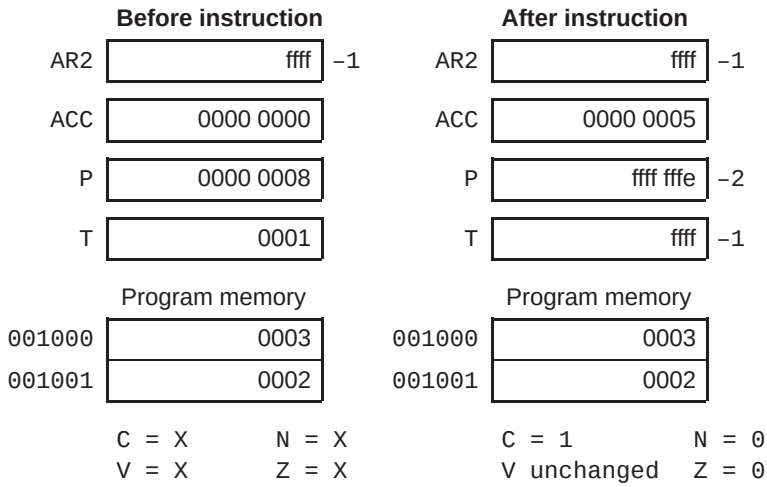
Program memory

001000	0003
001001	0002
001002	0001

```
ARP = 2
C = 0          N = 0
V unchanged    Z = 0
```

Example 3

```
RPT#1 || MAC   P, @AR2, 0:1000h

; Product shift mode is PM = 1 (no shift)
; ACC = 0 + [P] + (–1 x 3) = 8 – 3 = 5
; P = 8, then (–1 x 3), then (–1 x 2)
; Status bit changes associated with final ACC operation
```



Syntax

1: **MOV** ***(0:**16bit**),** *loc*
2: **MOV** **A***X***,** *loc*
3: **MOV** **ACC,** *loc* {**<<** *shift3*}
4: **MOV** **ACC,** #*16BitSU* {**<<** *shift2*}
5: **MOV** **ACC, P**
6: **MOV** **AR***x***,** *loc*
7: **MOV** **XAR***n***,** *locLong*
8: **MOV** **XAR***n***,** #*22bit*
9: **MOV** **DP,** #*10bit*
10: **MOV** **IER,** *loc*
11: **MOV** **P***X***,** *loc*
12: **MOV** **P, ACC**
13: **MOV** **T,** *loc*
14: **MOV** *loc***, #0**
15: **MOV** *loc***,** #*16bit*
16: **MOV** *loc***, *(0:**16bit**)**
17: **MOV** *loc***, A***X*
18: **MOV** *loc***, ACC** {**<<** *shift1*}
19: **MOV** *loc***, AR***x*
20: **MOV** *locLong***, XAR***n*
21: **MOV** *loc***, IER**
22: **MOV** *loc***, P**
23: **MOV** *loc***, T**

Operands

10bit — 10-bit number from 0 to 1023

16bit — 16-bit number from 0000_{16} to $FFFF_{16}$ (0 to 65 535)

16BitSU — If SXM = 0, *16BitSU* is an unsigned 16-bit number from 0 to 65 535. If SXM = 1, *16BitSU* is a signed 16-bit number from –32 768 to 32 767.

22bit — 22-bit number from 00 0000_{16} to 3F $FFFF_{16}$

ACC — Accumulator

AR*x* — Use one of the following:
AR0 AR1 AR2 AR3 AR4 AR5
AR6 AR7

A*X* — Use AH (the high word of the accumulator) or AL (the low word of the accumulator).

loc or *locLong* — The operand *loc* is a reference to a 16-bit location, either a data-memory location or a 16-bit register. The operand *locLong* is a reference to a 32-bit data-memory location or one of the 22-bit auxiliary registers (XAR6 or XAR7). Use any of these types of operands:

- *@***0:***6bit* — PAGE0-direct-addressing operand. Data page is 0. Offset is specified by *6bit*, a 6-bit constant from 0 to 63. PAGE0 direct addressing mode is available only when PAGE0 = 1.
- *@6bit* — DP-direct-addressing operand. Data page is specified by the DP. Offset is specified by *6bit*, a 6-bit constant from 0 to 63.

@reg Register-addressing operand. If the syntax uses *loc*, use one of the following for *reg*:

AH	AL	PL	PH	T	SP
AR0	AR1	AR2	AR3	AR4	AR5
AR6	AR7				

If the syntax uses *locLong*, you have these two options for *reg*:

XAR6 XAR7

–SP[6bit***]** PAGE0-stack-addressing operand. Address in data memory is specified as 0:(SP – *6bit*), where 0: indicates that the six MSBs are 0s and *6bit* is a 6-bit constant from 0 to 63. PAGE0 stack addressing mode is available only when PAGE0 = 0.

******ind* Indirect-addressing operand. Select one of the following operands (*x* is a number from 0 to 5; *n* is 6 or 7; *z* is a number from 0 to 7; *3bit* is a 3-bit constant):

*SP++	*AR*x*	*XAR*n*	*ARPz
*––SP	*AR*x*++	*XAR*n*++	*
	*––AR*x*	*––XARn	*++
	*+AR*x*[AR0]	*+XARn[AR0]	*––
	*+AR*x*[AR1]	*+XARn[AR1]	*0++
	*+AR*x*[*3bit*]	*+XARn[*3bit*]	*0––
	*AR6%++		

DP Data page pointer

IER Interrupt enable register

P*X* Use PH (the high word of the P register) or PL (the low word of the P register).

P Product register

shift1 Number from 0 to 8

shift2 Number from 0 to 15

shift3 Number from 0 to 16

T Multiplicand register

XAR*n* Use XAR6 or XAR7.

Opcode

Syntax 1: **MOV** ***(0:***16bit***),** *loc*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	See section 5.7.2 on page 5-23.							
16bit															

Syntax 2: **MOV** **A***X***,** *loc*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	0	0	1	AX	See section 5.7.2 on page 5-23.							

Note: If AL is used in the instruction, AX = 0; if AH is used, AX = 1.

Syntax 3: **MOV** **ACC,** *loc* {**<<** *shift3*}

For *shift3* < 16:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	*shift3*				See section 5.7.2 on page 5-23.							

For *shift3* = 16:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	1	See section 5.7.2 on page 5-23.							

Syntax 4: **MOV** **ACC,** #*16BitSU* {**<<** *shift2*}

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	0	0	1	0	*shift2*			
16BitSU															

Syntax 5: **MOV** **ACC, P**

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

Note: This is the opcode for the following equivalent instruction: MOVP T, @T.

Syntax 6: **MOV** **AR***x*, *loc*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	auxNumber			See section 5.7.2 on page 5-23.							

The following table shows the relationship between the specified auxiliary register and auxNumber:

Auxiliary Register	auxNumber	Auxiliary Register	auxNumber
AR0	0	AR4	4
AR1	1	AR5	5
AR2	2	AR6	6
AR3	3	AR7	7

Syntax 7: **MOV** **XAR***n*, *locLong*

For XAR6:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	1	0	0	See section 5.7.2 on page 5-23.							

For XAR7:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	1	0	1	See section 5.7.2 on page 5-23.							

Syntax 8: **MOV** **XAR***n*, #*22bit*

For XAR6:

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	0	*22bit*(21:16)					
22bit(15:0)															

For XAR7:

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	1	*22bit*(21:16)					
22bit(15:0)															

Syntax 9: **MOV DP,** #*10bit*

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

Syntax 10: **MOV IER,** *loc*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	0	1	1	See section 5.7.2 on page 5-23.							

Syntax 11: **MOV P***X***,** *loc*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	PX	1	1	1	See section 5.7.2 on page 5-23.							

Note: If PL is used in the instruction, PX = 0; if PH is used, PX = 1.

Syntax 12: **MOV P, ACC**

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

Syntax 13: **MOV T,** *loc*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	1	1	0	1	See section 5.7.2 on page 5-23.							

Syntax 14: **MOV** *loc***, #0**

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	1	0	1	1	See section 5.7.2 on page 5-23.							

Syntax 15: **MOV** *loc***,** #*16bit*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	1	0	0	0	See section 5.7.2 on page 5-23.							
16bit															

Syntax 16: **MOV** *loc*, ***(0:***16bit***)**

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	1	See section 5.7.2 on page 5-23.							
16bit															

Syntax 17: **MOV** *loc*, **A***X*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	0	1	1	AX	See section 5.7.2 on page 5-23.							

Note: If AL is used in the instruction, AX = 0; if AH is used, AX = 1.

Syntax 18: **MOV** *loc*, **ACC** {**<<** *shift1*}

For *shift1* > 0:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	(*shift1* – 1)			See section 5.7.2 on page 5-23.							

For *shift1* = 0 or no shift operand:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	0	1	1	0	See section 5.7.2 on page 5-23.							

Note: This is the opcode for the following equivalent syntax: MOV *loc*, AL.

Syntax19: **MOV** *loc*, **AR***x*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	auxNumber			See section 5.7.2 on page 5-23.							

The following table shows the relationship between the specified auxiliary register and auxNumber:

Auxiliary Register	auxNumber	Auxiliary Register	auxNumber
AR0	0	AR4	4
AR1	1	AR5	5
AR2	2	AR6	6
AR3	3	AR7	7

Syntax 20: **MOV** *locLong*, **XAR***n*

For XAR6:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	1	0	See section 5.7.2 on page 5-23.							

For XAR7:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	1	1	See section 5.7.2 on page 5-23.							

Syntax 21: **MOV** *loc*, **IER**

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	0	0	0	See section 5.7.2 on page 5-23.							

Syntax 22: **MOV** *loc*, **P**

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	1	1	1	See section 5.7.2 on page 5-23.							

Syntax 23: **MOV** *loc*, **T**

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	0	0	1	See section 5.7.2 on page 5-23.							

Description

The general form for the MOV instruction is as follows:

MOV *destination*, *source*

The value referenced or supplied by *source* is loaded to the data-memory location or register referenced by *destination*.

In syntaxes 3 and 4, the SXM bit affects *source*. If SXM = 0, *source* is treated as unsigned. If SXM = 1, *source* is treated as signed.

In syntaxes 3, 4, and 18, a left shift can be specified:

MOV *destination*, *source* **<<** *shift*

The value given by *source* is left shifted before being loaded to the location referenced by *destination*. During the shift, low-order bits are zero filled. In syntaxes 3 and 4, the result of the shift depends on SXM. If SXM = 1, the value is sign extended. If SXM = 0, the high-order bits are zero filled. Syntax 18 is not affected by SXM.

When AR6 or AR7 is read or loaded by the MOV instruction, the six most significant bits of the extended auxiliary register (XAR6 or XAR7) are not affected by the MOV instruction. When XAR6 or XAR7 is loaded from a 32-bit location, only the 22 LSBs of that location are loaded to XAR6/XAR7. If a 32-bit location is loaded from XAR6/XAR7, the 22 LSBs of that location are loaded from XAR6/XAR7, and the 10 MSBs are filled with 0s.

If register addressing is used for the operand *locLong*, you must use @XAR6 or @XAR7; if any other register is specified, the CPU generates an illegal-instruction trap.

The ’27x core expects memory wrappers or peripheral-interface logic to force 32-bit accesses, like those of syntaxes 7 and 20, to align to even addresses. This alignment is described in section 6.2 on page 6-31.

Execution

Syntax 1: **MOV** ***(0:***16bit***),** *loc*

[addressed location] → data-memory address 0:*16bit*
[PC] + 2 → PC

Syntax 2: **MOV** **A***X***,** *loc*

[addressed location] → AX
[PC] + 1 → PC

Syntax 3: **MOV** **ACC,** *loc* {**<<** *shift3*}

If SXM = 0
    [addressed location] is unsigned
If SXM = 1
    [addressed location] is signed
[addressed location] → ACC
[PC] + 1 → PC

If a shift is specified, the addressed value is shifted before the move. During the shift, low-order bits are zero filled. If SXM = 1, the value is sign extended; if SXM = 0, high-order bits are zero filled.

Syntax 4: **MOV** **ACC,** #*16BitSU* {**<<** *shift2*}

If SXM = 0
    16-bit unsigned constant → ACC
If SXM = 1
    16-bit signed constant → ACC
[PC] + 2 → PC

If a shift is specified, the 16-bit constant is shifted before the move. During the shift, low-order bits are zero filled. If SXM = 1, the value is sign extended; if SXM = 0, high-order bits are zero filled.

Syntax 5: **MOV** **ACC, P**

([P] shifted as per PM bits) → ACC
[PC] + 1 → PC

Syntax 6: **MOV** **AR***x***,** *loc*

[addressed location] → AR*x*
[PC] + 1 → PC

Syntax 7: **MOV** **XAR***n***,** *locLong*

If *locLong* = @XAR*i* (*i* = 6 or 7)
    [XAR*i*] → XAR*n*
    [PC] + 1 → PC
else
    [bits 21:0 of 32-bit addressed location] → XARn
    [PC] + 1 → PC

Syntax 8: **MOV** **XAR***n***,** #*22bit*

22-bit constant → XAR*n*
[PC] + 2 → PC

Syntax 9: **MOV** **DP,** #*10bit*

10-bit constant → DP(9:0)
[PC] + 1 → PC

DP(15:10) not affected.

Syntax 10: **MOV** **IER,** *loc*

[addressed location] → IER
[PC] + 1 → PC

Syntax 11: **MOV** **P***X*, *loc*

[addressed location] → PX
[PC] + 1 → PC

Syntax 12: **MOV** **P, ACC**

[ACC] → P
[PC] + 1 → PC

Syntax 13: **MOV** **T,** *loc*

[addressed location] → T
[PC] + 1 → PC

Syntax 14: **MOV** *loc*, **#0**

0000_{16} → addressed location
[PC] + 1 → PC

Syntax 15: **MOV** *loc*, #*16bit*

16-bit constant → addressed location
[PC] + 2 → PC

Syntax 16: **MOV** *loc*, ***(0:***16bit***)**

[data-memory address 0:*16bit*] → addressed location
[PC] + 2 → PC

Syntax 17: **MOV** *loc*, **A***X*

[AX] → addressed location
[PC] + 1 → PC

Syntax 18: **MOV** *loc*, **ACC** {**<<** *shift1*}

If *shift1* > 0
    [ACC] → shifter
    Shift left by *shift1*.
    shifter(15:0) → addressed location
    [PC] + 1 → PC

If *shift1* = 0 or no shift operand is used
    [AL] → addressed location
    [PC] + 1 → PC

During the shift, low-order bits are zero filled. ACC is not modified.

Syntax 19: **MOV** *loc*, **AR***x*

[AR*x*] → addressed location
[PC] + 1 → PC

Syntax 20: **MOV** *locLong*, **XAR***n*

If *locLong* = @XAR*i* (*i* = 6 or 7)
    [XAR*n*] → XAR*i*
    [PC] + 1 → PC
else
    ([XAR*n*] AND 003F FFFF$_{16}$) → 32-bit addressed location
    [PC] + 1 → PC

Syntax 21: **MOV** *loc*, **IER**

[IER] → addressed location
[PC] + 1 → PC

Syntax 22: **MOV** *loc*, **P**

Shift [P] as per PM bits.
16 LSBs of shifted [P] → addressed location
[PC] + 1 → PC

Syntax 23: **MOV** *loc*, **T**

[T] → addressed location
[PC] + 1 → PC

Status Bits

Syntaxes 1, 6–13, 20, and 21:

OVM, SXM	Neither affects the operation.
C, N, V, Z	None of these are affected by the operation.

Syntax 2 (**A***X* is destination):

OVM, SXM	Neither affects the operation.
C, V	C and V are not affected by the operation.
N	If bit 15 of AH/AL is 1, N is set; otherwise, N is cleared.
Z	If AH/AL is 0, Z is set; otherwise, Z is cleared.

Syntaxes 3, 4, and 5 (ACC is destination):

OVM	OVM does not affect the operation.
SXM	For syntaxes 3 and 4, SXM affects the 16-bit source operand as follows: SXM = 0 The 16-bit source operand is treated as an unsigned number. The value is not sign extended during the operation. SXM = 1 The 16-bit source operand is treated as a signed number. The value is sign extended during the operation.
PM	For syntax 5 (MOV ACC, P), the P value is shifted as defined by PM before it is stored to ACC. (P itself is not modified.)
C, OVC, V	None of these are affected by the operation.
N	After the move, if bit 31 of ACC is 1, N is set; otherwise, N is cleared.
Z	After the move, if ACC = 0, Z is set; otherwise, Z is cleared.

Syntaxes 14–19, 22, and 23 (*loc* is destination and **IER** is not source):

OVM, SXM	Neither affects the operation.
PM	For syntax 22 (MOV *loc*, P), the P value is shifted as defined by PM before it is stored to the destination. (P itself is not modified.)
C, V	C and V are not affected by the operation.
N	If AH or AL is loaded, N is affected as follows: If bit 15 of AH/AL is 1, N is set; otherwise, N is cleared. If neither AH nor AL is loaded, N is not affected.
Z	If AH or AL is loaded, Z is affected as follows: If AH/AL is 0, Z is set; otherwise, Z is cleared. If neither AH nor AL is loaded, Z is not affected.

Repeat

The following syntaxes of the MOV instruction are repeatable. (See the description for the RPT instruction.) The other syntaxes are not; if a nonrepeatable syntax follows a RPT instruction, it resets the repeat counter and executes only once.

Syntax 1: **MOV** ***(0:***16bit***),*** *loc*

Syntax 14: **MOV** *loc***, #0**

Syntax 15: **MOV** *loc*, #*16bit*

Syntax 16: **MOV** *loc*, ***(0:***16bit***)**

For syntax 1, the value ***(0:***16bit***)** is incremented by 1 at the end of each execution of the instruction. Thus, a new data-memory address is loaded during each repetition. If you use indirect addressing to specify the source (*loc*), a new value can be loaded each time as well. Otherwise, the same value is loaded to each consecutive data-memory address.

For syntaxes 14 and 15, the source is a constant. If you use indirect addressing to specify the destination (*loc*), a new location can be loaded with the constant during each execution of the instruction.

For syntax 16, the value ***(0:***16bit***)** is incremented by 1 at the end of each execution of the instruction. Thus, a new data-memory address is read during each repetition. If you use indirect addressing to specify the destination (*loc*), a new location can be loaded each time as well.

Once a repeat loop begins, it cannot be interrupted by any interrupt.

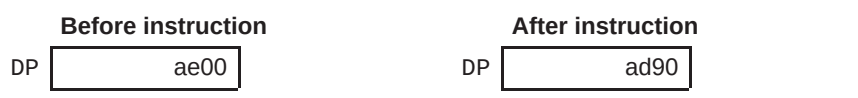
Words

Syntaxes 1, 4, 8, 15, and 16: 2

Syntaxes 2, 3, 5–7, 9–14, and 17–23: 1

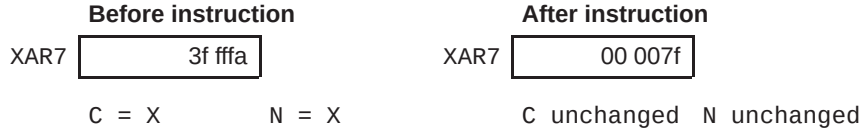
Example 1

```
MOV    DP, #190h
; Load 10 LSBs of DP.
```



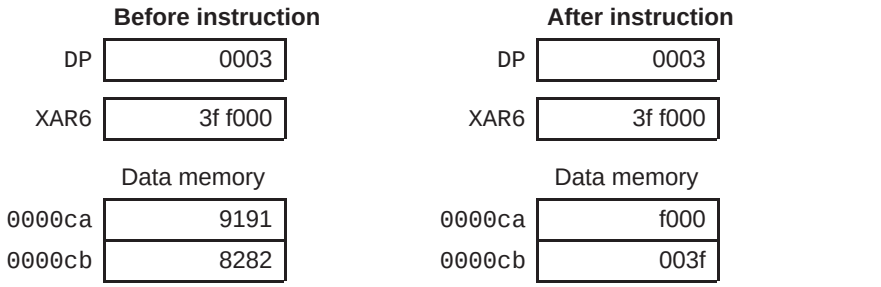
Example 2

```
MOV    XAR7, #00007fh
```



Example 3

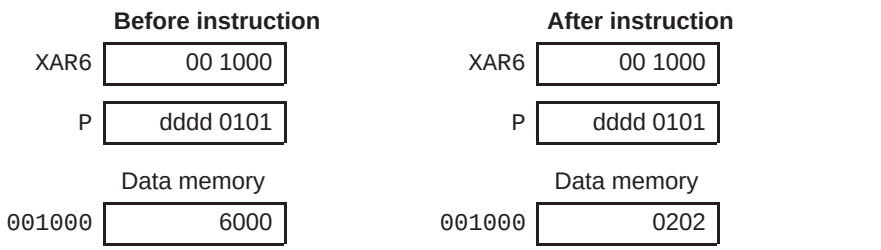
```
MOV    @0ah, XAR6

; Store XAR6 to two 16–bit locations in data memory.
```




```
C = X        N = X          C unchanged  N unchanged
V = X        Z = X          V unchanged  Z unchanged
```

Example 4

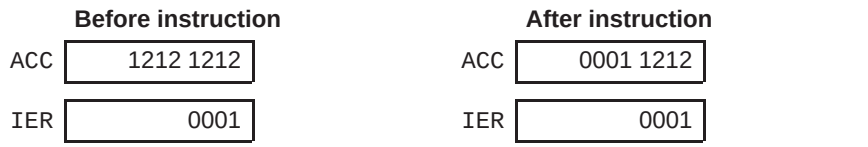
```
MOV    *XAR6, P

; Product shift mode is PM = 0 (left shift by 1)
```




```
ARP = X                          ARP = 6
C = X        N = X               C unchanged  N unchanged
V = X        Z = X               V unchanged  Z unchanged
```

Example 5

```
MOV    @AH, IER
```




```
C = X        N = X          C unchanged  N unchanged
V = X        Z = X          V unchanged  Z unchanged
```

Example 6

```
MOV    AH, AR2
```

Before instruction		After instruction	
ACC	1212 1212	ACC	8000 1212
AR2	8000	AR2	8000

```
C = X        N = X          C unchanged  N = 1
V = X        Z = X          V unchanged  Z = 0
```

Example 7

```
MOV    @AR4, T
```

Before instruction		After instruction	
AR4	2525	AR4	ffee
T	ffee	T	ffee

```
C = X        N = X          C unchanged  N unchanged
V = X        Z = X          V unchanged  Z unchanged
```

Example 8

```
MOV    @7, ACC << 4

; Result at 00 0087_16   = 16 LSBs of ([ACC] << 4)
                         = 8200 << 4 = 2000
```

Before instruction		After instruction	
DP	0002	DP	0002
ACC	aaaa 8200	ACC	aaaa 8200
Data memory		Data memory	
000087	2626	000087	2000

```
C = X        N = X          C unchanged  N unchanged
V = X        Z = X          V unchanged  Z unchanged
```

Example 9

```
MOV    ACC, @5 << 16
```

; ACC = (value at 00 0085_{16}) << 16 = 8fff 0000_{16}

	Before instruction	After instruction
DP	0002	0002
ACC	3939 3939	8fff 0000

Data memory

	Before instruction	After instruction
000085	8fff	8fff

Before instruction		After instruction	
C = X	N = X	C unchanged	N = 1
V = X	Z = X	V unchanged	Z = 0

Example 10

```
RPT  #1   ||   MOV  *(0:0086h), @5
```

; Copy value at 00 0085_{16} to the next two locations.

	Before instruction	After instruction
DP	0002	0002

Data memory

	Before instruction	After instruction
000085	0005	0005
000086	7979	0005
000087	4747	0005

Before instruction		After instruction	
C = X	N = X	C unchanged	N unchanged
V = X	Z = X	V unchanged	Z unchanged

Example 11

```
RPT #2   ||   MOV *--AR4, *(0:85h)

; Copy values:   from 00 0085h to 00 1002h
;                from 00 0086h to 00 1001h
;                from 00 0087h to 00 1000h
```

Before instruction

AR4	1003

Data memory

000085	0003
000086	0002
000087	0001
001000	0000
001001	0000
001002	0000

ARP = X
C = X N = X
V = X Z = X

After instruction

AR4	1000

Data memory

000085	0003
000086	0002
000087	0001
001000	0001
001001	0002
001002	0003

ARP = 4
C unchanged N unchanged
V unchanged Z unchanged

Example 12

```
RPT #1   ||   MOV *AR1++, #0

; Clear two consecutive data-memory locations.
```

Before instruction

AR1	0080

Data memory

000080	2828
000081	4545

ARP = X
C = X N = X
V = X Z = X

After instruction

AR1	0082

Data memory

000080	0000
000081	0000

ARP = 1
C unchanged N unchanged
V unchanged Z unchanged

Syntax

MOVA T, *loc*

Operands

loc Reference to a 16-bit location, either a data-memory location or a 16-bit register. Use any of these types of operands:

- **@0:***6bit* PAGE0-direct-addressing operand. Data page is 0. Offset is specified by *6bit*, a 6-bit constant from 0 to 63. PAGE0 direct addressing mode is available only when PAGE0 = 1.
- @*6bit* DP-direct-addressing operand. Data page is specified by the DP. Offset is specified by *6bit*, a 6-bit constant from 0 to 63.
- @*reg* Register-addressing operand. For *reg*, specify one of these register names:

AH	AL	PL	PH	T	SP
AR0	AR1	AR2	AR3	AR4	AR5
AR6	AR7				

- ***–SP[***6bit***]*** PAGE0-stack-addressing operand. Address in data memory is specified as 0:(SP – *6bit*), where 0: indicates that the six MSBs are 0s and *6bit* is a 6-bit constant from 0 to 63. PAGE0 stack addressing mode is available only when PAGE0 = 0.
- ***ind*** Indirect-addressing operand. Select one of the following operands (*x* is a number from 0 to 5; *n* is 6 or 7; *z* is a number from 0 to 7; *3bit* is a 3-bit constant):

*SP++	*AR*x*	*XAR*n*	*ARPz
*––SP	*AR*x*++	*XAR*n*++	*
	*––AR*x*	*––XARn	*++
	*+AR*x*[AR0]	*+XARn[AR0]	*––
	*+AR*x*[AR1]	*+XARn[AR1]	*0++
	*+AR*x*[*3bit*]	*+XARn[*3bit*]	*0––
	*AR6%++		

T Multiplicand register

Opcode

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	0	0	0	See section 5.7.2 on page 5-23.							

Description

The content of the location referenced by *loc* is loaded to the T register. Also, the content of the P register, shifted according to the value of the PM bits in status register ST0, is added to the content of ACC.

If you want to add P to ACC but leave T unchanged, you can use the instruction MOVA T, @T.

Execution

[addressed location] → T
[ACC] + ([P] shifted as per PM bits) → ACC
[PC] + 1 → PC

Status Bits

OVM, OVC — If the operation causes ACC to overflow, ACC and OVC depend on OVM:

- OVM = 0 — ACC is not filled with a saturation value. OVC is affected as follows:

  If the overflow was in the negative direction, OVC is decremented by 1; if the overflow was in the positive direction, OVC is incremented by 1.

- OVM = 1 — If the overflow was in the positive direction, ACC is filled with 7FFF FFFF$_{16}$. If the overflow was in the negative direction, ACC is filled with 8000 0000$_{16}$.

  OVC is not affected.

SXM — SXM does not affect the operation.

PM — The P value is shifted as defined by PM before it is added to ACC. (P itself is not modified.)

C — If the addition of P to ACC generates a carry, C is set; otherwise, C is cleared.

N — If bit 31 of the result in ACC is 1, N is set; otherwise, N is cleared.

V — If an overflow occurs, V is set; otherwise, V is not affected.

Z — If the result in ACC is 0, Z is set; otherwise, Z is cleared.

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Words

1

Example 1

```
MOVA   T, @T

; Product shift mode is PM = 5 (shift right by 4, sign extend)
; Status bit changes associated with ACC operation:
;      ACC = [ACC] + ([P] >> 4) = 1 + 1
```

	Before instruction	After instruction
ACC	0000 0001	0000 0002
P	0000 0010	0000 0010
T	5000	5000
	C = X N = X	C = 0 N = 0
	V = X Z = X	V unchanged Z = 0

Example 2

```
MOVA   T, @AL

; Product shift mode is PM = 7 (shift right by 6, sign extend)
; Status bit changes associated with ACC operation:
;      ACC = [ACC] + ([P] >> 6) = 8_16+ e_16 = 16_16
```

	Before instruction	After instruction
ACC	0000 0008	0000 0016
P	0000 0380	0000 0380
T	0000	0008
	C = X N = X	C = 0 N = 0
	V = X Z = X	V unchanged Z = 0

Example 3

```
MOVA   T, *SP++

; Product shift mode is PM = 1 (no shift)
; Status bit changes associated with ACC operation:
;      ACC = [ACC] + [P] = 7fff fffe16 + 0000 000316
```

	Before instruction	After instruction
SP	0080	0081
P	0000 0003	0000 0003
T	2323	0005
Data memory		
000080	0005	0005

```
;For OVM = 0: ACC overflows normally. OVC records overflow.
```

ACC	7fff fffe	8000 0001
	OVC = 0	OVC = 1
	C = X N = X	C = 0 N = 1
	V = X Z = X	V = 1 Z = 0

```
; For OVM = 1: ACC filled with saturation value
```

ACC	7fff fffe	7fff ffff
	OVC = X	OVC unchanged
	C = X N = X	C = 0 N = 0
	V = X Z = X	V = 1 Z = 0

Syntax

1: **MOVB** **A***X***.LSB,** *loc*
2: **MOVB** *loc***, A***X***.LSB**
3: **MOVB** **A***X***.MSB,** *loc*
4: **MOVB** *loc***, A***X***.MSB**
5: **MOVB** **A***X***,** #*8bit*
6: **MOVB** **ACC,** #*8bit*
7: **MOVB** **AR***x***,** #*8bit*

Operands

8bit 8-bit number from 0 to 255

ACC Accumulator

AR*x* Use one of the following:
AR0 AR1 AR2 AR3 AR4 AR5
AR6 AR7

A*X* Use AH (the high word of the accumulator) or AL (the low word of the accumulator). A*X*.LSB is the least significant byte of A*X*. A*X*.MSB is the most significant byte of A*X*.

loc Reference to a 16-bit location, either a data-memory location or a 16-bit register. Use any of these types of operands:

- **@0:***6bit* PAGE0-direct-addressing operand. Data page is 0. Offset is specified by *6bit*, a 6-bit constant from 0 to 63. PAGE0 direct addressing mode is available only when PAGE0 = 1.
- *@6bit* DP-direct-addressing operand. Data page is specified by the DP. Offset is specified by *6bit*, a 6-bit constant from 0 to 63.
- ***–SP[***6bit**]** PAGE0-stack-addressing operand. Address in data memory is specified as 0:(SP – *6bit*), where 0: indicates that the six MSBs are 0s and *6bit* is a 6-bit constant from 0 to 63. PAGE0 stack addressing mode is available only when PAGE0 = 0.

***ind** Indirect-addressing operand. Select one of the following operands (*x* is a number from 0 to 5; *n* is 6 or 7; *z* is a number from 0 to 7; *3bit* is a 3-bit constant):

*SP++	*AR*x*	*XAR*n*	*ARPz
*––SP	*AR*x*++	*XAR*n*++	*
	*––AR*x*	*––XARn	*++
	*+AR*x*[AR0]	*+XARn[AR0]	*––
	*+AR*x*[AR1]	*+XARn[AR1]	*0++
	*+AR*x*[*3bit*]	*+XARn[*3bit*]	*0––
	*AR6%++		

@*reg* Register-addressing operand. For *reg*, specify one of these register names:

AH	AL	PL	PH	T	SP
AR0	AR1	AR2	AR3	AR4	AR5
AR6	AR7				

Opcode

Syntax 1: **MOVB** **A*X*.LSB,** *loc*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	1	1	AX	See section 5.7.2 on page 5-23.							

Note: If AL is used in the instruction, AX = 0; if AH is used, AX = 1.

Syntax 2: **MOVB** *loc*, **A*X*.LSB**

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	1	0	AX	See section 5.7.2 on page 5-23.							

Note: If AL is used in the instruction, AX = 0; if AH is used, AX = 1.

Syntax 3: **MOVB** **A*X*.MSB,** *loc*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	0	0	AX	See section 5.7.2 on page 5-23.							

Note: If AL is used in the instruction, AX = 0; if AH is used, AX = 1.

Syntax 4: **MOVB** *loc*, **A*X*.MSB**

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	1	0	0	AX	See section 5.7.2 on page 5-23.							

Note: If AL is used in the instruction, AX = 0; if AH is used, AX = 1.

Syntax 5: **MOVB** **A***X*, #*8bit*

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

Note: If AL is used in the instruction, AX = 0; if AH is used, AX = 1.

Syntax 6: **MOVB** **ACC,** #*8bit*

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

Syntax 7: **MOVB** **AR***x*, #*8bit*

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

The following table shows the relationship between the specified auxiliary register and auxNumber:

Auxiliary Register	auxNumber	Auxiliary Register	auxNumber
AR0	0	AR4	4
AR1	1	AR5	5
AR2	2	AR6	6
AR3	3	AR7	7

Description

The general form for the MOVB instruction is as follows:

MOVB *destination*, *source*

The 8-bit unsigned value referenced or supplied by *source* is loaded to the data-memory location or register referenced by *destination*. When the destination is wider than 8 bits, the value is loaded into the least significant byte of the destination, and the higher-order bits are cleared to 0.

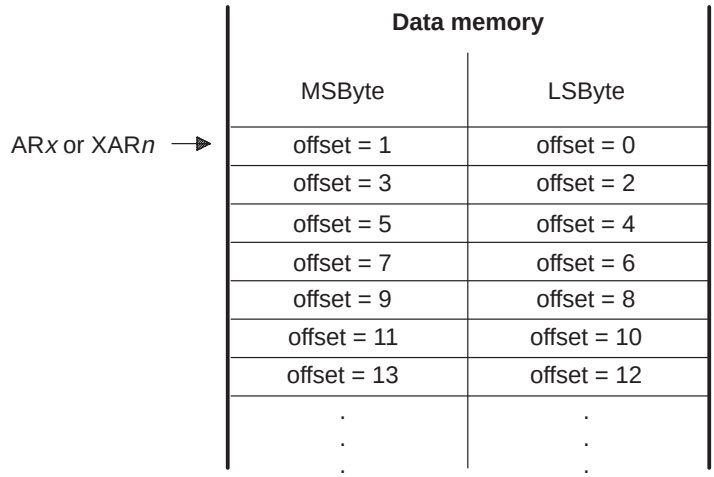
In syntaxes 1–4, the destinations are 8 bits wide. The least significant byte (LSByte) or most significant byte (MSByte) of the source is loaded to the LSByte or MSByte of the destination. A special case for any of these four syntaxes is the case when auxiliary-register direct addressing is used for the operand *loc* and an address offset is specified.

In indirect addressing, an auxiliary register acts as a pointer; in auxiliary-register indirect addressing, you can specify an offset to be added to the pointer. Because the source of the offset can be AR0, AR1, or a constant given in the instruction, there are three types of operands for using the offset feature:

- *+AR*x*[AR0] or *+XAR*n*[AR0]
- *+AR*x*[AR1] or *+XAR*n*[AR1]
- *+ARx[3-bit constant] or *+XAR*n*[3-bit constant]

Normally, the offset indicates a specific word relative to where the auxiliary register points. In syntaxes 1–4 of the MOVB instruction, the offset indicates a specific *byte* relative to where the auxiliary register points. Figure 6–1 shows the offset values needed to access particular bytes. If the offset is an odd number, an MSByte is accessed; if the offset is an even number, an LSByte is accessed.

Figure 6–1. Relationship Among Auxiliary Register, Offset, and Accessed Byte During Move to or From AX.LSB or AX.MSB

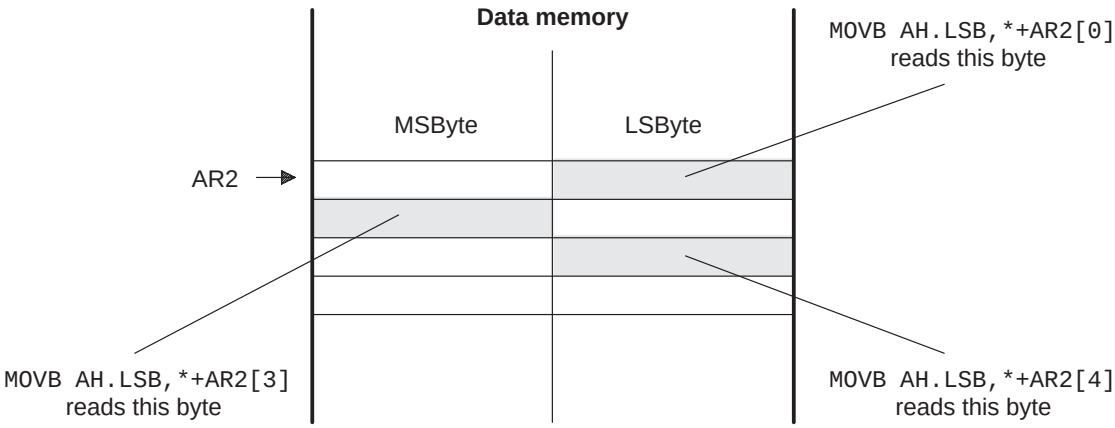


When AR0 or AR1 is the source for the offset, the MOVB instruction can access any one of 65 535 possible bytes relative to the selected auxiliary register. When a 3-bit constant is used for the offset, the offset must be in the range from 0 to 7, and the instruction can read only one of eight possible bytes relative to the selected auxiliary register.

Consider the following instructions, which all use syntax 1 and use AR2 as a pointer. The only difference among them is the offset in each case (0, 3, and 4).

```
MOVB   AH.LSB, *+AR2[0]
MOVB   AH.LSB, *+AR2[3]
MOVB   AH.LSB, *+AR2[4]
```

Each of them accesses a different byte relative to the address in AR2:



Execution

Syntax 1: **MOVB A*X*.LSB,** *loc*

For *loc* = *+AR*x*[offset] or *loc* = +XAR*n*[offset],
where offset is AR0, AR1, or a 3-bit constant:
If offset is an even number
[AR*x*/XAR*n*] + (offset >> 1) → temp
[Least significant byte of location given by temp] → AX.LSB
If offset is an odd number
[AR*x*/XAR*n*] + (offset >>1) → temp
[Most significant byte of location given by temp] → AX.LSB
00_{16} → AX.MSB
[PC] + 1 → PC

For other values of *loc*:
[Least significant byte of addressed location] → AX.LSB
00_{16} → AX.MSB
[PC] + 1 → PC

Syntax 2: **MOVB** *loc*, **A***X***.LSB**

For *loc* = *+ARx[offset] or *loc* = +XAR*n*[offset], where offset is AR0, AR1, or a 3-bit constant:
If offset is an even number
[AR*x*/XAR*n*] + (offset >> 1) → temp
AX.LSB → Least significant byte of location given by temp
Most significant byte not modified
If offset is an odd number
[AR*x*/XAR*n*] + (offset >>1) → temp
AX.LSB → Most significant byte of location given by temp
Least significant byte not modified
[PC] + 1 → PC

For other values of *loc*:
AX.LSB → Least significant byte of addressed location
Most significant byte not modified
[PC] + 1 → PC

Syntax 3: **MOVB** **A***X***.MSB,** *loc*

For *loc* = *+ARx[offset] or *loc* = +XAR*n*[offset], where offset is AR0, AR1, or a 3-bit constant:
If offset is an even number
[AR*x*/XAR*n*] + (offset >> 1) → temp
[Least significant byte of location given by temp] → AX.MSB
If offset is an odd number
[AR*x*/XAR*n*] + (offset >>1) → temp
[Most significant byte of location given by temp] → AX.MSB
AX.LSB not modified
[PC] + 1 → PC

For other values of *loc*:
[Least significant byte of addressed location] → AX.MSB
AX.LSB not modified
[PC] + 1 → PC

Syntax 4: **MOVB** *loc*, **A***X***.MSB**

For *loc* = *+AR*x*[offset] or *loc* = +XAR*n*[offset], where offset is AR0, AR1, or a 3-bit constant:
If offset is an even number
[AR*x*/XAR*n*] + (offset >> 1) → temp
AX.MSB → Least significant byte of location given by temp
Most significant byte not modified
If offset is an odd number
[AR*x*/XAR*n*] + (offset >>1) → temp
AX.MSB → Most significant byte of location given by temp
Least significant byte not modified
[PC] + 1 → PC

For other values of *loc*:
AX.MSB → Least significant byte of addressed location
Most significant byte not modified
[PC] + 1 → PC

Syntax 5: **MOVB** **A***X*, #*8bit*

(8-bit unsigned constant AND $00FF_{16}$) → AX
[PC] + 1 → PC

Syntax 6: **MOVB** **ACC,** #*8bit*

(8-bit unsigned constant AND $00FF_{16}$) → AL
0000_{16} → AH
[PC] + 1 → PC

Syntax 7: **MOVB** **AR***x*, #*8bit*

8-bit unsigned constant → AR*x*
[PC] + 1 → PC

Status Bits

Syntaxes 1, 3, 5, and 6 (A*X*, A*X*.LSB, or A*X*.MSB is loaded):

OVM, SXM	Neither affects the operation.
C, OVC, V	None of these are affected by the operation.
N	After the move, if bit 15 of A*X* is 1, N is set; otherwise, N is cleared.
Z	After the move, if A*X* = 0, Z is set; otherwise, Z is cleared.

Syntax 6 (ACC is loaded):

OVM, SXM	Neither affects the operation.
C, OVC, V	None of these are affected by the operation.
N	After the move, if bit 31 of ACC is 1, N is set; otherwise, N is cleared.
Z	After the move, if ACC = 0, Z is set; otherwise, Z is cleared.

Syntaxes 2 and 4 (*loc* is loaded):

OVM, SXM	Neither affects the operation.
C, V	C and V are not affected by the operation.
N	If AH or AL is loaded, N is affected as follows: If bit 15 of AH/AL is 1, N is set; otherwise, N is cleared. If neither AH nor AL is loaded, N is not affected.
Z	If AH or AL is loaded, Z is affected as follows: If AH/AL is 0, Z is set; otherwise, Z is cleared. If neither AH nor AL is loaded, Z is not affected.

Syntaxes 7 (An auxiliary register is loaded):

OVM, SXM	Neither affects the operation.
C, N, V, Z	None of these are affected by the operation.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Words 1

Example 1

```
MOVB   AH.LSB, *+AR5[AR1]

; Copy addressed byte to LSByte of AH, and clear MSByte of AH.
```

	Before instruction	After instruction
AR1	0004	0004
AR5	0090	0090
ACC	aaaa aaaa	0044 aaaa

Data memory		
000090	8000	8000
000091	7000	7000
000092	6044	6044

Before instruction	After instruction
ARP = X	ARP = 5
C = X N = X	C unchanged N = 0
V = X Z = X	V unchanged Z = 0

Example 2

```
MOVB   AH.MSB, *AR5++

; Copy addressed LSbyte to MSByte of AH.
```

	Before instruction	After instruction
AR5	0090	0091
ACC	2222 4444	0022 4444

Data memory		
000090	8800	8800

Before instruction	After instruction
ARP = X	ARP = 5
C = X N = X	C unchanged N = 0
V = X Z = X	V unchanged Z = 0

Example 3

```
MOVB   @AH, AL.LSB

; Copy LSByte of AL to LSByte of AH.
```

	Before instruction	After instruction
ACC	aaaa 8039	aa39 8039

Before instruction	After instruction
C = X N = X	C unchanged N = 1
V = X Z = X	V unchanged Z = 0

Example 4

```
MOVB   *–SP[3], AL.MSB

; PAGE0 stack addressing mode (PAGE0 = 0)
; Copy MSByte of AL to LSByte of addressed location
```

	Before instruction	After instruction
SP	0089	0089
ACC	5500 6700	5500 6700

Data memory	Before instruction	After instruction
000086	9900	9967
000087	8800	8800
000088	7700	7700
000089	6600	6600

Before instruction		After instruction	
C = X	N = X	C unchanged	N unchanged
V = X	Z = X	V unchanged	Z unchanged

Example 5

```
MOVB   AL, #5
```

	Before instruction	After instruction
ACC	eeee eeee	eeee 0005

Before instruction		After instruction	
C = X	N = X	C unchanged	N = 0
V = X	Z = X	V unchanged	Z = 0

Example 6

```
MOVB   ACC, #26h
```

	Before instruction	After instruction
ACC	bbbb bbbb	0000 0026

Before instruction		After instruction	
C = X	N = X	C unchanged	N = 0
V = X	Z = X	V unchanged	Z = 0

Example 7

```
MOVB   AR7, #0
```

	Before instruction	After instruction
XAR7	15 1515	15 0000

Before instruction		After instruction	
C = X	N = X	C unchanged	N unchanged
V = X	Z = X	V unchanged	Z unchanged

Syntax

1: **MOVH** *loc*, **P**

2: **MOVH** *loc*, **ACC** {**<<** *shift1*}

Operands

ACC Accumulator

loc Reference to a 16-bit location, either a data-memory location or a 16-bit register. Use any of these types of operands:

- **@0:***6bit* PAGE0-direct-addressing operand. Data page is 0. Offset is specified by *6bit*, a 6-bit constant from 0 to 63. PAGE0 direct addressing mode is available only when PAGE0 = 1.
- *@6bit* DP-direct-addressing operand. Data page is specified by the DP. Offset is specified by *6bit*, a 6-bit constant from 0 to 63.
- *@reg* Register-addressing operand. For *reg*, specify one of these register names:

AH	AL	PL	PH	T	SP
AR0	AR1	AR2	AR3	AR4	AR5
AR6	AR7				

- ***–SP[***6bit***]*** PAGE0-stack-addressing operand. Address in data memory is specified as 0:(SP – *6bit*), where 0: indicates that the six MSBs are 0s and *6bit* is a 6-bit constant from 0 to 63. PAGE0 stack addressing mode is available only when PAGE0 = 0.
- ***ind*** Indirect-addressing operand. Select one of the following operands (*x* is a number from 0 to 5; *n* is 6 or 7; *z* is a number from 0 to 7; *3bit* is a 3-bit constant):

*SP++	*AR*x*	*XAR*n*	*ARPz
*––SP	*AR*x*++	*XAR*n*++	*
	*––AR*x*	*––XARn	*++
	*+AR*x*[AR0]	*+XARn[AR0]	*––
	*+AR*x*[AR1]	*+XARn[AR1]	*0++
	*+AR*x*[*3bit*]	*+XARn[*3bit*]	*0––
	*AR6%++		

P Product register

shift1 Number from 0 to 8

Opcode

Syntax 1: **MOVH** *loc*, **P**

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	0	1	1	1	See section 5.7.2 on page 5-23.							

Syntax 2: **MOVH** *loc*, **ACC** {**<<** *shift1*}

For *shift1* > 0:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	0	(*shift1* – 1)			See section 5.7.2 on page 5-23.							

For *shift1* = 0 or no shift operand:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	0	1	1	1	See section 5.7.2 on page 5-23.							

Note: This is the opcode for the following equivalent syntax: MOV *loc*, AH.

Description

Syntax 1 of the MOVH instruction enables you to perform this operation:

1) Pass a copy of P to the shifter.
2) Shift the copy according to the PM bits in status register ST0.
3) Store the 16 MSBs of the shifted value to the addressed location.

Syntax 2 of the MOVH instruction enables you to perform this operation:

1) Pass a copy of ACC to the shifter.
2) Shift the copy left by the amount given by *shift1*. (This step is optional. If *shift1* = 0 or you do not use the operand << *shift1*, no shift occurs.)
3) Store the 16 MSBs of the (shifted) value to the addressed location.

When AR6 or AR7 is loaded, the six most significant bits of the extended auxiliary register (XAR6 or XAR7) are not affected.

Execution

Syntax 1: **MOVH** *loc*, **P**

[P] → shifter
Shift as per PM bits.
Shift right by 16.
shifter(15:0) → addressed location
[PC] + 1 → PC

P is not modified.

Syntax 2: **MOVH** *loc*, **ACC** {**<<** *shift1*}

If *shift1* > 0
[ACC] → shifter
Shift right by (16 – *shift1*).
shifter(15:0) → addressed location
[PC] + 1 → PC

If *shift1* = 0 or no shift operand is used
[AH] → addressed location
[PC] + 1 → PC

ACC is not modified.

Status Bits

OVM, SXM	Neither affects the operation.
PM	For syntax 1, the P value is shifted as defined by PM before it is stored to the addressed location. (P itself is not modified.)
C, V	C and V are not affected by the operation.
N	If AH or AL is loaded, N is affected as follows: If bit 15 of AH/AL is 1, N is set; otherwise, N is cleared. If neither AH nor AL is loaded, N is not affected.
Z	If AH or AL is loaded, Z is affected as follows: If AH/AL is 0, Z is set; otherwise, Z is cleared. If neither AH nor AL is loaded, Z is not affected.

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Words

1

Example 1

```
MOVH   @AL, P

; Product shift mode is PM = 1 (no shift)
; AL = 16 LSBs of ([P] >> 16) = 8000₁₆
```

Before instruction		After instruction	
ACC	3232 3232	ACC	3232 8000
P	8000 4000	P	8000 4000
C = X	N = X	C unchanged	N = 1
V = X	Z = X	V unchanged	Z = 0

Example 2

```
MOVH   @T, P

; Product shift mode is PM = 3 (shift right by 2, sign extend)
; T =  16 LSBs of ( ([P] >> 2) >> 16 )
;      16 LSBs of ( e000 1000_16 >> 16 )
```

	Before instruction	After instruction
T	5050	e000
P	8000 4000	8000 4000

```
C = X       N = X          C unchanged  N unchanged
V = X       Z = X          V unchanged  Z unchanged
```

Example 3

```
MOVH   *--AR0, ACC << 8

; Decrement AR0 before write to memory.
; Result at 00 008f_16 = 16 LSBs of ( ([ACC] << 8) >> 16 )
;                        16 LSBs of ( 00aa aa00_16 >> 16 )
```

	Before instruction	After instruction
AR0	0090	008f
ACC	8800 aaaa	8800 aaaa
Data memory		
00008f	4444	00aa

```
ARP = X                    ARP = 0
C = X       N = X          C unchanged  N unchanged
V = X       Z = X          V unchanged  Z unchanged
```

Example 4

```
MOVH   @AL, ACC
```

	Before instruction	After instruction
ACC	8800 aaaa	8800 8800

```
C = X       N = X          C unchanged  N = 1
V = X       Z = X          V unchanged  Z = 0
```

Syntax

1: **MOVL** **ACC,** *locLong*

2: **MOVL** *locLong***, ACC**

Operands

ACC Accumulator

locLong Reference to a 32-bit data-memory location or one of the 22-bit auxiliary registers (XAR6 or XAR7). Use any of these types of operands:

- **@0:***6bit* PAGE0-direct-addressing operand. Data page is 0. Offset is specified by *6bit*, a 6-bit constant from 0 to 63. PAGE0 direct addressing mode is available only when PAGE0 = 1.
- @*6bit* DP-direct-addressing operand. Data page is specified by the DP. Offset is specified by *6bit*, a 6-bit constant from 0 to 63.
- @**XAR***n* Register-addressing operand. For **XAR***n*, specify XAR6 or XAR7.
- ***–SP[***6bit***]*** PAGE0-stack-addressing operand. Address in data memory is specified as 0:(SP – *6bit*), where 0: indicates that the six MSBs are 0s and *6bit* is a 6-bit constant from 0 to 63. PAGE0 stack addressing mode is available only when PAGE0 = 0.
- ***ind* Indirect-addressing operand. Select one of the following operands (*x* is a number from 0 to 5; *n* is 6 or 7; *z* is a number from 0 to 7; *3bit* is a 3-bit constant):

*SP++	*AR*x*	*XAR*n*	*ARPz
*––SP	*AR*x*++	*XAR*n*++	*
	*––AR*x*	*––XARn	*++
	*+AR*x*[AR0]	*+XARn[AR0]	*––
	*+AR*x*[AR1]	*+XARn[AR1]	*0++
	*+AR*x*[*3bit*]	*+XARn[*3bit*]	*0––
	*AR6%++		

Opcode

Syntax 1: **MOVL** **ACC,** *locLong*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	See section 5.7.2 on page 5-23.							

Syntax 2: **MOVL** *locLong***, ACC**

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	1	0	See section 5.7.2 on page 5-23.							

Description

The general form for the MOVL instruction is as follows:

MOVL *destination*, *source*

The 22- or 32-bit value referenced by *source* is loaded to the 32-bit data-memory location or 22-bit auxiliary register referenced by *destination*. If register addressing is used for one of the operands, you must use @XAR6 or @XAR7; if any other register is specified, the CPU generates an illegal-instruction trap.

If the destination is loaded from XAR6/XAR7, the 22 LSBs of the destination are loaded from XAR6/XAR7, and the 10 MSBs are filled with 0s.

The ’27x core expects memory wrappers or peripheral-interface logic to force any 32-bit access, like that of the MOVL instruction, to align to an even address. This alignment is described in section 6.2 on page 6-31.

Execution

Syntax 1: **MOVL** **ACC,** *locLong*

If *locLong* = @XAR*i* (*i* = 6 or 7)
  ([XAR*i*] AND 003F FFFF$_{16}$) → ACC
  [PC] + 1 → PC
else
  [32-bit addressed location] → ACC
  [PC] + 1 → PC

Syntax 2: **MOVL** *locLong*, **ACC**

If *locLong* = @XAR*i* (*i* = 6 or 7)
  [ACC(21:0)] → XAR*i*
  [PC] + 1 → PC
else
  [ACC] → 32-bit addressed location
  [PC] + 1 → PC

Status Bits

Syntax1: **MOVL ACC,** *locLong*

OVM, SXM	Neither affects the operation.
C, OVC, V	None of these are affected by the operation.
N	After the move, if bit 31 of ACC is 1, N is set; otherwise, N is cleared.
Z	After the move, if ACC = 0, Z is set; otherwise, Z is cleared.

Syntax 2: **MOVL** *locLong*, **ACC**

OVM, SXM	Neither affects the operation.
C, N, V, Z	None of these are affected by the operation.

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Words

1

Example 1

```
MOVL   ACC, *--SP
; Decrement SP by 2 before read from data memory.
```

	Before instruction		After instruction	
SP	0070		SP	006e
ACC	7373 7373		ACC	8080 4040
	Data memory			Data memory
00006e	4040		00006e	4040
00006f	8080		00006f	8080
000070	6060		000070	6060
	C = X	N = X		C unchanged N = 1
	V = X	Z = X		V unchanged Z = 0

Example 2

```
MOVL   @XAR6, ACC

; Load XAR6 with ACC(21:0).
```

	Before instruction		After instruction
XAR6	00 7fff	XAR6	3c 0000
ACC	fffc 0000	ACC	fffc 0000

```
C = X       N = X          C unchanged  N unchanged
V = X       Z = X          V unchanged  Z unchanged
```

Example 3

```
MOVL   @6, ACC

; Store ACC to two 16–bit locations in data memory.
```

	Before instruction		After instruction
DP	0002	DP	0002
ACC	7000 3000	ACC	7000 3000
	Data memory		Data memory
000086	3434	000086	3000
000087	5656	000087	7000

```
C = X       N = X          C unchanged  N unchanged
V = X       Z = X          V unchanged  Z unchanged
```

Syntax **MOVP T,** *loc*

Operands

loc Reference to a 16-bit location, either a data-memory location or a 16-bit register. Use any of these types of operands:

- **@0:***6bit* PAGE0-direct-addressing operand. Data page is 0. Offset is specified by *6bit*, a 6-bit constant from 0 to 63. PAGE0 direct addressing mode is available only when PAGE0 = 1.
- @*6bit* DP-direct-addressing operand. Data page is specified by the DP. Offset is specified by *6bit*, a 6-bit constant from 0 to 63.
- @*reg* Register-addressing operand. For *reg*, specify one of these register names:

AH	AL	PL	PH	T	SP
AR0	AR1	AR2	AR3	AR4	AR5
AR6	AR7				

- ***–SP[***6bit***]*** PAGE0-stack-addressing operand. Address in data memory is specified as 0:(SP – *6bit*), where 0: indicates that the six MSBs are 0s and *6bit* is a 6-bit constant from 0 to 63. PAGE0 stack addressing mode is available only when PAGE0 = 0.
- ***ind*** Indirect-addressing operand. Select one of the following operands (*x* is a number from 0 to 5; *n* is 6 or 7; *z* is a number from 0 to 7; *3bit* is a 3-bit constant):

*SP++	*AR*x*	*XAR*n*	*ARPz
*––SP	*AR*x*++	*XAR*n*++	*
	*––AR*x*	*––XARn	*++
	*+AR*x*[AR0]	*+XARn[AR0]	*––
	*+AR*x*[AR1]	*+XARn[AR1]	*0++
	*+AR*x*[*3bit*]	*+XARn[*3bit*]	*0––
	*AR6%++		

T Multiplicand register

Opcode

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	1	1	0	See section 5.7.2 on page 5-23.							

Description

The content of the location referenced by *loc* is loaded to the T register. Also, the content of the P register, shifted according to the value of the PM bits in status register ST0, replaces the content of ACC.

If you want to store P to ACC but leave T unchanged, you can use the instruction MOVP T, @T.

Execution

[addressed location] → T
([P] shifted as per PM bits) → ACC
[PC] + 1 → PC

Status Bits

OVM, SXM	Neither affects the operation.
PM	The P value is shifted as defined by PM before it is stored to ACC. (P itself is not modified.)
C, OVC, V	None of these are affected by the operation.
N	After the move, if bit 31 of ACC is 1, N is set; otherwise, N is cleared.
Z	After the move, if ACC = 0, Z is set; otherwise, Z is cleared.

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Words

1

Example 1

```
MOVP T, @T

; Product shift mode is PM = 5 (shift right by 4, sign extend)
; Status bit changes associated with ACC operation:
;      ACC = ([P] >> 4)
```

	Before instruction	After instruction
ACC	9696 9696	0000 0002
P	0000 0020	0000 0020
T	0003	0003

```
C = X       N = X          C unchanged  N = 0
V = X       Z = X          V unchanged  Z = 0
```

Example 2

```
MOVP T, @0:10h

; PAGE0 direct addressing mode (PAGE0 = 1)
; Product shift mode is PM = 1 (no shift)
; Status bit changes associated with ACC operation:
;      ACC = [P]
```

	Before instruction	After instruction
ACC	2323 2323	0000 0000
P	0000 0000	0000 0000
T	1111	5000

Data memory		Data memory	
000010	5000	000010	5000

C = X	N = X
V = X	Z = X

C unchanged N = 0
V unchanged Z = 1

Example 3

```
MOVP T, *AR3

; Product shift mode is PM = 0 (shift left by 1)
; Status bit changes associated with ACC operation:
;      ACC = ([P] << 1)
```

	Before instruction	After instruction
AR3	0090	0090
ACC	7878 7878	8000 0000
P	4000 0000	4000 0000
T	5656	0004

Data memory		Data memory	
000090	0004	000090	0004

ARP = X	
C = X	N = X
V = X	Z = X

ARP = 3
C unchanged N = 1
V unchanged Z = 0

Syntax **MOVS T,** *loc*

Operands

loc Reference to a 16-bit location, either a data-memory location or a 16-bit register. Use any of these types of operands:

- **@0:***6bit* PAGE0-direct-addressing operand. Data page is 0. Offset is specified by *6bit*, a 6-bit constant from 0 to 63. PAGE0 direct addressing mode is available only when PAGE0 = 1.
- @*6bit* DP-direct-addressing operand. Data page is specified by the DP. Offset is specified by *6bit*, a 6-bit constant from 0 to 63.
- @*reg* Register-addressing operand. For *reg*, specify one of these register names:

AH	AL	PL	PH	T	SP
AR0	AR1	AR2	AR3	AR4	AR5
AR6	AR7				

- ***–SP[***6bit***]*** PAGE0-stack-addressing operand. Address in data memory is specified as 0:(SP – *6bit*), where 0: indicates that the six MSBs are 0s and *6bit* is a 6-bit constant from 0 to 63. PAGE0 stack addressing mode is available only when PAGE0 = 0.
- ***ind*** Indirect-addressing operand. Select one of the following operands (*x* is a number from 0 to 5; *n* is 6 or 7; *z* is a number from 0 to 7; *3bit* is a 3-bit constant):

*SP++	*AR*x*	*XAR*n*	*ARPz
*––SP	*AR*x*++	*XAR*n*++	*
	*––AR*x*	*––XARn	*++
	*+AR*x*[AR0]	*+XARn[AR0]	*––
	*+AR*x*[AR1]	*+XARn[AR1]	*0++
	*+AR*x*[*3bit*]	*+XARn[*3bit*]	*0––
	*AR6%++		

T Multiplicand register

Opcode

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	0	0	1	See section 5.7.2 on page 5-23.							

Description

The content of the location referenced by *loc* is loaded to the T register. Also, the content of the P register, shifted according to the value of the PM bits in status register ST0, is subtracted from the content of ACC.

If you want to subtract P from ACC but leave T unchanged, you can use the instruction MOVS T, @T.

Execution

[addressed location] → T
[ACC] – ([P] shifted as per PM bits) → ACC
[PC] + 1 → PC

Status Bits

OVM, OVC — If the operation causes ACC to overflow, ACC and OVC depend on OVM:

- OVM = 0 — ACC is not filled with a saturation value. OVC is affected as follows:

  If the overflow was in the negative direction, OVC is decremented by 1; if the overflow was in the positive direction, OVC is incremented by1.

- OVM = 1 — If the overflow was in the positive direction, ACC is filled with 7FFF $FFFF_{16}$. If the overflow was in the negative direction, ACC is filled with 8000 0000_{16}.

  OVC is not affected.

SXM — SXM does not affect the operation.

PM — The P value is shifted as defined by PM before it is subtracted from ACC. (P itself is not modified.)

C — If the subtraction of P from ACC generates a borrow, C is cleared; otherwise, C is set.

N — If bit 31 of the result in ACC is 1, N is set; otherwise, N is cleared.

V — If an overflow occurs, V is set; otherwise, V is not affected.

Z — If the result in ACC is 0, Z is set; otherwise, Z is cleared.

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Words

1

Example 1

```
MOVS T, *SP++

; Product shift mode is PM = 1 (no shift)
; Status bit changes associated with ACC operation:
;      ACC = [ACC] – [P]
```

	Before instruction	After instruction
SP	0080	0081
ACC	0000 0002	0000 0000
P	0000 0002	0000 0002
T	9898	0005
Data memory		
000080	0005	0005

```
C = X       N = X          C = 1       N = 0
V = X       Z = X          V unchanged Z = 1
```

Example 2

```
MOVS T, @4

; Product shift mode is PM = 1 (no shift)
; Status bit changes associated with ACC operation.
```

	Before instruction	After instruction
DP	0002	0002
ACC	6000 0000	e000 0000
P	8000 0000	8000 0000
T	3737	0001
Data memory		
000084	0001	0001

```
C = X       N = X          C = 0       N = 1
V = X       Z = X          V unchanged Z = 0
```

Example 3

```
MOVS   T, @AR1

; Product shift mode is PM = 3 (shift right by 2, sign extend)
; Status bit changes associated with ACC operation:
;      ACC = [ACC] – ([P] >> 2)= 8000 0000₁₆ – 0000 0002₁₆
```

	Before instruction	After instruction
P	0000 0008	0000 0008
T	4242	000e
AR1	000e	000e

```
;For OVM = 0: ACC overflows normally. OVC records overflow.
```

ACC	8000 0000	7fff fffe
	OVC = 0	OVC = –1
	C = X N = X	C = 1 N = 0
	V = X Z = X	V = 1 Z = 0

```
;For OVM = 1: ACC filled with saturation value
```

ACC	8000 0000	8000 0000
	OVC = X	OVC unchanged
	C = X N = X	C = 1 N = 1
	V = X Z = X	V = 1 Z = 0

Syntax **MOVU ACC,** *loc*

Operands

ACC Accumulator

loc Reference to a 16-bit location, either a data-memory location or a 16-bit register. Use any of these types of operands:

@0:*6bit* PAGE0-direct-addressing operand. Data page is 0. Offset is specified by *6bit*, a 6-bit constant from 0 to 63. PAGE0 direct addressing mode is available only when PAGE0 = 1.

@6bit DP-direct-addressing operand. Data page is specified by the DP. Offset is specified by *6bit*, a 6-bit constant from 0 to 63.

@reg Register-addressing operand. For *reg*, specify one of these register names:

AH	AL	PL	PH	T	SP
AR0	AR1	AR2	AR3	AR4	AR5
AR6	AR7				

–SP[6bit***]** PAGE0-stack-addressing operand. Address in data memory is specified as 0:(SP – *6bit*), where 0: indicates that the six MSBs are 0s and *6bit* is a 6-bit constant from 0 to 63. PAGE0 stack addressing mode is available only when PAGE0 = 0.

******ind* Indirect-addressing operand. Select one of the following operands (*x* is a number from 0 to 5; *n* is 6 or 7; *z* is a number from 0 to 7; *3bit* is a 3-bit constant):

*SP++	*AR*x*	*XAR*n*	*ARPz
*––SP	*AR*x*++	*XAR*n*++	*
	*––AR*x*	*––XARn	*++
	*+AR*x*[AR0]	*+XARn[AR0]	*––
	*+AR*x*[AR1]	*+XARn[AR1]	*0++
	*+AR*x*[*3bit*]	*+XARn[*3bit*]	*0––
	*AR6%++		

Opcode

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	1	0	See section 5.7.2 on page 5-23.							

Description The low half of the accumulator (AL) is loaded with the content of the location referenced by *loc*, and the high half (AH) is filled with 0s.

Execution

[addressed location] → AL
0000_{16} → AH
[PC] + 1 → PC

Status Bits

OVM, SXM	Neither affects the operation.
C, OVC, V	None of these are affected by the operation.
N	After the move, N is cleared because bit 31 of ACC is not 1.
Z	After the move, if ACC = 0, Z is set; otherwise, Z is cleared.

Repeat

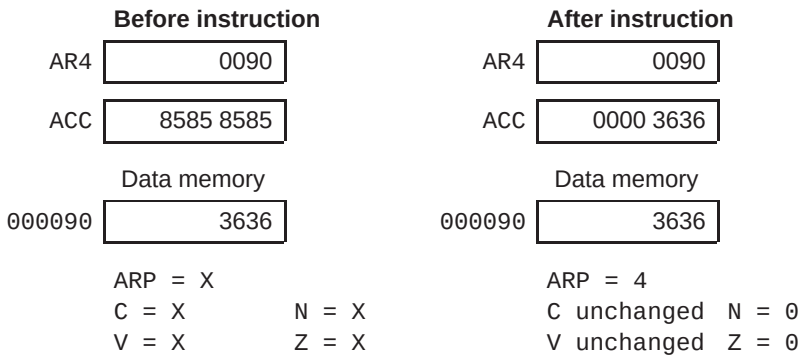
This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Words

1

Example

```
MOVU   ACC, *AR4
```



Syntax **MOVW DP,** #*16bit*

Operands *16bit* 16-bit number from 0000_{16} to $FFFF_{16}$ (0 to 65 535)

DP Data page pointer

Opcode

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	0	0	1	1	1	1	1
16bit															

Description The 16-bit constant is loaded into the data page pointer (DP).

Execution 16-bit constant → DP
[PC] + 2 → PC

Status Bits OVM, SXM Neither affects the operation.

C, N, V, Z None of these are affected by the operation.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Words 2

Example

```
MOVW   DP, #190h
```

Before instruction

```
DP   ae00

C = X        N = X
V = X        Z = X
```

After instruction

```
DP   0190

C unchanged  N unchanged
V unchanged  Z unchanged
```

Syntax

1: **MPY** **ACC,** *loc*, #*16BitSigned*
2: **MPY** **ACC, T,** *loc*
3: **MPY** **P, T,** *loc*

Operands

16BitSigned 16-bit signed number from –32 768 to 32 767.

ACC Accumulator

loc Reference to a 16-bit location, either a data-memory location or a 16-bit register. Use any of these types of operands:

- **@0:***6bit* PAGE0-direct-addressing operand. Data page is 0. Offset is specified by *6bit*, a 6-bit constant from 0 to 63. PAGE0 direct addressing mode is available only when PAGE0 = 1.
- @*6bit* DP-direct-addressing operand. Data page is specified by the DP. Offset is specified by *6bit*, a 6-bit constant from 0 to 63.
- @*reg* Register-addressing operand. For *reg*, specify one of these register names:

AH	AL	PL	PH	T	SP
AR0	AR1	AR2	AR3	AR4	AR5
AR6	AR7				

- ***–SP[***6bit***]*** PAGE0-stack-addressing operand. Address in data memory is specified as 0:(SP – *6bit*), where 0: indicates that the six MSBs are 0s and *6bit* is a 6-bit constant from 0 to 63. PAGE0 stack addressing mode is available only when PAGE0 = 0.
- *******ind* Indirect-addressing operand. Select one of the following operands (*x* is a number from 0 to 5; *n* is 6 or 7; *z* is a number from 0 to 7; *3bit* is a 3-bit constant):

*SP++	*AR*x*	*XAR*n*	*ARPz
*––SP	*AR*x*++	*XAR*n*++	*
	*––AR*x*	*––XARn	*++
	*+AR*x*[AR0]	*+XARn[AR0]	*––
	*+AR*x*[AR1]	*+XARn[AR1]	*0++
	*+AR*x*[*3bit*]	*+XARn[*3bit*]	*0––
	*AR6%++		

P Product register

T Multiplicand register

Opcode

Syntax 1: **MPY ACC,** *loc*, #*16BitSigned*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	0	1	0	0	See section 5.7.2 on page 5-23.							
16BitSigned															

Syntax 2: **MPY ACC, T,** *loc*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	0	1	0	See section 5.7.2 on page 5-23.							

Syntax 3: **MPY P, T,** *loc*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	0	0	1	1	See section 5.7.2 on page 5-23.							

Description

The general form of the MPY instruction is as follows:

MPY *destination*, *multiplicand1*, *multiplicand2*

multiplicand1 is multiplied by *multiplicand2*, and the result is stored to *destination* (ACC or P). The operation is as follows:

1) If syntax 1 is used, load the T register with the value referenced by *loc*; otherwise, leave T unchanged.
2) Multiply the T register value by the other multiplicand and store the result to the specified destination (ACC or P).

Notice that for syntaxes 2 and 3, you must load T with the desired value before executing the MPY instruction.

The MPY instruction performs signed multiplication; that is, both multiplicands are treated as signed numbers.

Execution

Syntax 1: **MPY ACC,** *loc*, #*16BitSigned*

[addressed location] $\rightarrow$ T
[T] $\times$ 16-bit signed constant $\rightarrow$ ACC
[PC] + 2 $\rightarrow$ PC

Syntax 2: **MPY ACC, T,** *loc*

[T] $\times$ [addressed location] $\rightarrow$ ACC
[PC] + 1 $\rightarrow$ PC

Syntax 3: **MPY** **P, T,** *loc*

[T] × [addressed location] → P
[PC] + 1 → PC

Status Bits

Syntaxes 1 and 2 (ACC holds the result):

OVM, SXM	Neither affects the operation.
C, OVC, V	None of these are affected by the operation.
N	If bit 31 of the result in ACC is 1, N is set; otherwise, N is cleared.
Z	If the result in ACC is 0, Z is set; otherwise, Z is cleared.

Syntax 3 (P holds the result):

OVM, SXM	Neither affects the operation.
C, N, V, Z	None of these are affected by the operation.

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Words

Syntax 1: 2

Syntaxes 2 and 3: 1

Example 1

```
MPY    ACC, *AR2, #–7

; T = value at 00 0050₁₆
; ACC = [T] x (–7) = 4 x (–7) = –28 = ffff ffe4₁₆
```

	Before instruction		After instruction
AR2	0050	AR2	0050
ACC	7878 7878	ACC	ffff ffe4
T	0000	T	0004
Data memory		Data memory	
000050	0004	000050	0004
ARP = X		ARP = 2	
C = X	N = X	C unchanged	N = 1
V = X	Z = X	V unchanged	Z = 0

Example 2

```
MPY    ACC, T, @AR5
; ACC = [T] x [AR5] = 2 x 0 = 0
```

	Before instruction	After instruction
ACC	3636 3636	0000 0000
T	0002	0002
AR5	0000	0000

```
C = X       N = X          C unchanged  N = 0
V = X       Z = X          V unchanged  Z = 1
```

Example 3

```
MPY    P, T, @7
; P = [T] × (value at 000087₁₆) = 7 × 3 = 21 = 0000 0015₁₆
```

	Before instruction	After instruction
DP	0002	0002
P	8989 8989	0000 0015
T	0007	0007
Data memory		
000087	0003	0003

```
C = X       N = X          C unchanged  N unchanged
V = X       Z = X          V unchanged  Z unchanged
```

Syntax

1: **MPYA** **P,** *loc*, *#16BitSigned*

2: **MPYA** **P, T,** *loc*

Operands

16BitSigned 16-bit signed number from –32 768 to 32 767.

loc Reference to a 16-bit location, either a data-memory location or a 16-bit register. Use any of these types of operands:

- **@0:***6bit* PAGE0-direct-addressing operand. Data page is 0. Offset is specified by *6bit*, a 6-bit constant from 0 to 63. PAGE0 direct addressing mode is available only when PAGE0 = 1.
- @*6bit* DP-direct-addressing operand. Data page is specified by the DP. Offset is specified by *6bit*, a 6-bit constant from 0 to 63.
- @*reg* Register-addressing operand. For *reg*, specify one of these register names:

AH	AL	PL	PH	T	SP
AR0	AR1	AR2	AR3	AR4	AR5
AR6	AR7				

- ***–SP[***6bit***]*** PAGE0-stack-addressing operand. Address in data memory is specified as 0:(SP – *6bit*), where 0: indicates that the six MSBs are 0s and *6bit* is a 6-bit constant from 0 to 63. PAGE0 stack addressing mode is available only when PAGE0 = 0.
- ***ind* Indirect-addressing operand. Select one of the following operands (*x* is a number from 0 to 5; *n* is 6 or 7; *z* is a number from 0 to 7; *3bit* is a 3-bit constant):

*SP++	*AR*x*	*XAR*n*	*ARPz
*––SP	*AR*x*++	*XAR*n*++	*
	*––AR*x*	*––XARn	*++
	*+AR*x*[AR0]	*+XARn[AR0]	*––
	*+AR*x*[AR1]	*+XARn[AR1]	*0++
	*+AR*x*[*3bit*]	*+XARn[*3bit*]	*0––
	*AR6%++		

P Product register

T Multiplicand register

Opcode

Syntax 1: **MPYA P,** *loc*, *#16BitSigned*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	1	0	1	See section 5.7.2 on page 5-23.							
16BitSigned															

Syntax 2: **MPYA P, T,** *loc*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	1	1	1	See section 5.7.2 on page 5-23.							

Description

The general form for the MPYA instruction is as follows:

MPYA P, *multiplicand1*, *multiplicand2*

multiplicand1 is multiplied by *multiplicand2*, and the result is stored to the P register. Prior to the multiplication, the P register value, shifted according to the value of the PM bits in status register ST0, is added to ACC. The operation is as follows:

1) Shift the P register value according to the PM bits and add the shifted P value to ACC.
2) If syntax 1 is used, load the T register with the value referenced by *loc*; if syntax 2 is used, leave T unchanged.
3) Multiply the T register value by the other multiplicand and store the result to the P register.

Notice that for syntax 2, you must load T with the desired value before executing the MPYA instruction.

The MPYA instruction performs signed multiplication; that is, both multiplicands are treated as signed numbers.

Execution

Syntax 1: **MPYA P,** *loc*, *#16BitSigned*

[ACC] + ([P] shifted as per PM bits) $\rightarrow$ ACC
[addressed location] $\rightarrow$ T
[T] $\times$ 16-bit signed constant $\rightarrow$ P
[PC] + 2 $\rightarrow$ PC

Syntax 2: **MPYA P, T,** *loc*

[ACC] + ([P] shifted as per PM bits) $\rightarrow$ ACC
[T] $\times$ [addressed location] $\rightarrow$ P
[PC] + 1 $\rightarrow$ PC

Status Bits

OVM, OVC — If the operation causes ACC to overflow, ACC and OVC depend on OVM:

- OVM = 0 — ACC is not filled with a saturation value. OVC is affected as follows:

  If the overflow was in the negative direction, OVC is decremented by 1; if the overflow was in the positive direction, OVC is incremented by 1.

- OVM = 1 — If the overflow was in the positive direction, ACC is filled with 7FFF $FFFF_{16}$. If the overflow was in the negative direction, ACC is filled with 8000 0000_{16}.

  OVC is not affected.

SXM — SXM does not affect the operation.

PM — The P value is shifted as defined by PM before it is added to ACC. (P itself is not modified.)

C — If the addition of P to ACC generates a carry, C is set; otherwise, C is cleared.

N — If bit 31 of the result in ACC is 1, N is set; otherwise, N is cleared.

V — If an overflow occurs in ACC, V is set; otherwise, V is not affected.

Z — If the result in ACC is 0, Z is set; otherwise, Z is cleared.

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Words

Syntax 1: 2

Syntax 2: 1

Example 1

```
MPYA   P, @5, #–10

; Product shift mode is PM = 3 (shift right by 2, sign extend)
; ACC = [ACC] + ([P] >> 2) = 7fff fffe16 + 0000 000316
; P    = (value at 00 00c516) × (–10) = 2 × (–10) = ffff ffec16
```

	Before instruction	After instruction
DP	0003	0003
P	0000 000c	ffff ffec
T	3636	0002
Data memory 0000c5	0002	0002

```
; For OVM = 0: ACC overflows normally. OVC records overflow.
```

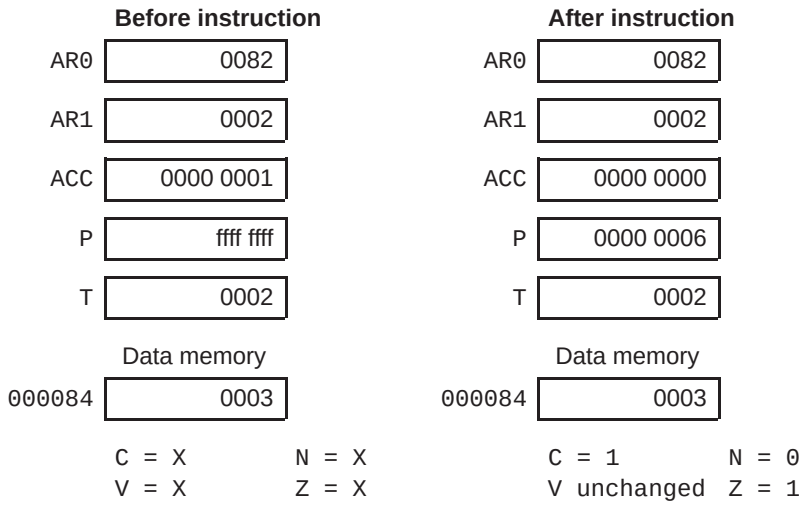
ACC	7fff fffe	8000 0001
	OVC = 0	OVC = 1
	C = X N = X	C = 0 N = 1
	V = X Z = X	V = 1 Z = 0

```
; For OVM = 1: ACC filled with saturation value
```

ACC	7fff fffe	7fff ffff
	OVC = X	OVC unchanged
	C = X N = X	C = 0 N = 0
	V = X Z = X	V = 1 Z = 0

Example 2

```
MPYA   P, T, *+AR0[AR1]

; Product shift mode is PM = 1 (no shift)
; Status bit changes associated with ACC operation:
;      ACC = [ACC] + [P] = 1 + (–1)  = 0
; P = [T] × (value at 00 0084₁₆) = 2 × 3 = 6
```



Syntax

1: **MPYB** **ACC, T,** #*8bit*

2: **MPYB** **P, T,** #*8bit*

Operands

8bit 8-bit number from 0 to 255

ACC Accumulator

P Product register

T Multiplicand register

Opcode

Syntax 1: **MPYB** **ACC, T,** #*8bit*

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

Syntax 2: **MPYB** **P, T,** #*8bit*

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

Description

The T register value is multiplied by the 8-bit unsigned constant, and the result is stored to the specified destination (ACC or P). The T register value is treated as a signed 16-bit number.

Execution

Syntax 1: **MPYB** **ACC, T,** #*8bit*

[T] $\times$ 8-bit unsigned constant $\rightarrow$ ACC
[PC] + 1 $\rightarrow$ PC

The value in the T register is treated as a signed 16-bit number.

Syntax 2: **MPYB** **P, T,** #*8bit*

[T] $\times$ 8-bit unsigned constant $\rightarrow$ P
[PC] + 1 $\rightarrow$ PC

The value in the T register is treated as a signed 16-bit number.

Status Bits

Syntax 1: **MPYB** **ACC, T,** #*8bit*

OVM, SXM	Neither affects the operation.
C, OVC, V	None of these are affected by the operation.
N	If bit 31 of the result in ACC is 1, N is set; otherwise, N is cleared.
Z	If the result in ACC is 0, Z is set; otherwise, Z is cleared.

Syntax 2: **MPYB** **P, T,** #*8bit*

OVM, SXM	Neither affects the operation.
C, N, V, Z	None of these are affected by the operation.

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Words

1

Example 1

```
MPYB   ACC, T, #3

; ACC = [T] × 3 = –2× 3 = –6 = ffff fffa16
```

Before instruction		After instruction	
ACC	5757 5757	ACC	ffff fffa
T	fffe	T	fffe
C = X	N = X	C unchanged	N = 1
V = X	Z = X	V unchanged	Z = 0

Example 2

```
MPYB   P, T, #4

; P = [T] × 4 = 2× 4 = 8
```

Before instruction		After instruction	
P	2828 2828	P	0000 0008
T	0002	T	0002
C = X	N = X	C unchanged	N unchanged
V = X	Z = X	V unchanged	Z unchanged

Syntax **MPYS P, T,** *loc*

Operands

loc Reference to a 16-bit location, either a data-memory location or a 16-bit register. Use any of these types of operands:

- **@0:***6bit* PAGE0-direct-addressing operand. Data page is 0. Offset is specified by *6bit*, a 6-bit constant from 0 to 63. PAGE0 direct addressing mode is available only when PAGE0 = 1.
- @*6bit* DP-direct-addressing operand. Data page is specified by the DP. Offset is specified by *6bit*, a 6-bit constant from 0 to 63.
- @*reg* Register-addressing operand. For *reg*, specify one of these register names:

AH	AL	PL	PH	T	SP
AR0	AR1	AR2	AR3	AR4	AR5
AR6	AR7				

- ***–SP[***6bit***]*** PAGE0-stack-addressing operand. Address in data memory is specified as 0:(SP – *6bit*), where 0: indicates that the six MSBs are 0s and *6bit* is a 6-bit constant from 0 to 63. PAGE0 stack addressing mode is available only when PAGE0 = 0.
- ***ind*** Indirect-addressing operand. Select one of the following operands (*x* is a number from 0 to 5; *n* is 6 or 7; *z* is a number from 0 to 7; *3bit* is a 3-bit constant):

*SP++	*AR*x*	*XAR*n*	*ARPz
*––SP	*AR*x*++	*XAR*n*++	*
	*––AR*x*	*––XARn	*++
	*+AR*x*[AR0]	*+XARn[AR0]	*––
	*+AR*x*[AR1]	*+XARn[AR1]	*0++
	*+AR*x*[*3bit*]	*+XARn[*3bit*]	*0––
	*AR6%++		

P Product register

T Multiplicand register

Opcode

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	0	1	1	See section 5.7.2 on page 5-23.							

Description

The following operation is carried out by the MPYS instruction:

1) Shift the P register value according to the PM bits in status register ST0, and subtract the shifted P value from ACC.
2) Multiply the T register value by the other multiplicand (referenced by *loc*), and store the result to the P register.

Notice that you must load T with the desired value before executing the MPYS instruction.

The MPYS instruction performs signed multiplication; that is, both multiplicands are treated as signed numbers.

Execution

[ACC] – ([P] shifted as per PM bits) → ACC
[T] × [addressed location] → P
[PC] + 1 → PC

Status Bits

OVM, OVC — If the operation causes ACC to overflow, ACC and OVC depend on OVM:

- OVM = 0 — ACC is not filled with a saturation value. OVC is affected as follows:

  If the overflow was in the negative direction, OVC is decremented by 1; if the overflow was in the positive direction, OVC is incremented by 1.

- OVM = 1 — If the overflow was in the positive direction, ACC is filled with 7FFF FFFF$_{16}$. If the overflow was in the negative direction, ACC is filled with 8000 0000$_{16}$.

  OVC is not affected.

SXM — SXM does not affect the operation.

PM — The P value is shifted as defined by PM before it is subtracted from ACC. (P itself is not modified.)

C — If the subtraction of P from ACC generates a borrow, C is cleared; otherwise, C is set.

N — If bit 31 of the result in ACC is 1, N is set; otherwise, N is cleared.

V — If an overflow occurs in ACC, V is set; otherwise, V is not affected.

Z — If the result in ACC is 0, Z is set; otherwise, Z is cleared.

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Words 1

Example 1

```
MPYS   P, T, @AL

; Product shift mode is PM = 0 (shift left by 1)
; Status bit changes associated with ACC operation:
;      ACC = [ACC] – ([P] << 1) = 8000 0001₁₆ – 0000 0004₁₆
; P = [T] × [AL] = 5 x 1
```

	Before instruction	After instruction
P	0000 0002	0000 0005
T	0005	0005

```
; For OVM = 0: ACC overflows normally. OVC records overflow.
```

ACC	8000 0001	7fff fffd
	OVC = 0	OVC = –1
	C = X N = X	C = 1 N = 0
	V = X Z = X	V = 1 Z = 0

```
; For OVM = 1: ACC filled with saturation value
```

ACC	8000 0001	8000 0000
	OVC = X	OVC unchanged
	C = X N = X	C = 1 N = 1
	V = X Z = X	V = 1 Z = 0

Example 2

```
MPYS   P, T, *AR3++

; Product shift mode is PM = 1 (no shift)
; Status bit changes associated with ACC operation:
;      [ACC] – [P] = 6000 0000₁₆ – 8000 0000₁₆
; P = [T] × [AL] = 5 × 2 = 0000 000a₁₆
```

	Before instruction	After instruction
AR3	0086	0087
ACC	6000 0000	e000 0000
P	8000 0000	0000 000a
T	0005	0005

Data memory		
000086	0002	0002

```
ARP = X                ARP = 3
C = X       N = X      C = 0       N = 1
V = X       Z = X      V unchanged Z = 0
```

Syntax

1: **MPYU ACC, T,** *loc*

2: **MPYU P, T,** *loc*

Operands

ACC Accumulator

loc Reference to a 16-bit location, either a data-memory location or a 16-bit register. Use any of these types of operands:

- **@0:***6bit* PAGE0-direct-addressing operand. Data page is 0. Offset is specified by *6bit*, a 6-bit constant from 0 to 63. PAGE0 direct addressing mode is available only when PAGE0 = 1.
- @*6bit* DP-direct-addressing operand. Data page is specified by the DP. Offset is specified by *6bit*, a 6-bit constant from 0 to 63.
- @*reg* Register-addressing operand. For *reg*, specify one of these register names:

AH	AL	PL	PH	T	SP
AR0	AR1	AR2	AR3	AR4	AR5
AR6	AR7				

- ***–SP[***6bit***]*** PAGE0-stack-addressing operand. Address in data memory is specified as 0:(SP – *6bit*), where 0: indicates that the six MSBs are 0s and *6bit* is a 6-bit constant from 0 to 63. PAGE0 stack addressing mode is available only when PAGE0 = 0.
- *******ind* Indirect-addressing operand. Select one of the following operands (*x* is a number from 0 to 5; *n* is 6 or 7; *z* is a number from 0 to 7; *3bit* is a 3-bit constant):

*SP++	*AR*x*	*XAR*n*	*ARPz
*––SP	*AR*x*++	*XAR*n*++	*
	*––AR*x*	*––XARn	*++
	*+AR*x*[AR0]	*+XARn[AR0]	*––
	*+AR*x*[AR1]	*+XARn[AR1]	*0++
	*+AR*x*[*3bit*]	*+XARn[*3bit*]	*0––
	*AR6%++		

P Product register

T Multiplicand register

Opcode

Syntax 1: **MPYU ACC, T,** *loc*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	0	1	1	0	See section 5.7.2 on page 5-23.							

Syntax 2: **MPYU P, T,** *loc*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	0	1	1	1	See section 5.7.2 on page 5-23.							

Description

The T register value is multiplied by the value referenced by *loc*, and the result is stored to the specified destination (ACC or P). Both multiplicands are treated as unsigned 16-bit numbers.

Execution

Syntax 1: **MPYU ACC, T,** *loc*

[T] × [addressed location] → ACC
[PC] + 1 → PC

Syntax 2: **MPYU P, T,** *loc*

[T] × [addressed location] → P
[PC] + 1 → PC

Status Bits

Syntax 1: **MPYU ACC, T,** *loc*

OVM, SXM	Neither affects the operation.
C, OVC, V	None of these are affected by the operation.
N	If bit 31 of the result in ACC is 1, N is set; otherwise, N is cleared.
Z	If the result in ACC is 0, Z is set; otherwise, Z is cleared.

Syntax 2: **MPYU P, T,** *loc*

OVM, SXM	Neither affects the operation.
C, N, V, Z	None of these are affected by the operation.

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Words

1

Example 1

```
MPYU   ACC, T, @10
```

; ACC = [T] × (value at 00 008a$_{16}$) = 65 534 × 8 = 0007 fff0$_{16}$

Before instruction		After instruction	
DP	0002	DP	0002
ACC	1818 1818	ACC	0007 fff0
T	fffe	T	fffe
Data memory		Data memory	
00008a	0008	00008a	0008

```
C = X        N = X          C unchanged  N = 0
V = X        Z = X          V unchanged  Z = 0
```

Example 2

```
MPYU   P, T, *XAR6++
```

; P = [T] × (value at 00 07ee$_{16}$) = 4 × 65 531 = 0003 ffec$_{16}$

Before instruction		After instruction	
XAR6	00 07ee	XAR6	00 07ef
P	4343 4343	P	0003 ffec
T	0004	T	0004
Data memory		Data memory	
0007ee	fffb	0007ee	fffb

```
ARP = X                           ARP = 6
C = X        N = X                C unchanged  N unchanged
V = X        Z = X                V unchanged  Z unchanged
```

Syntax

1: **MPYXU ACC, T,** *loc*

2: **MPYXU P, T,** *loc*

Operands

ACC Accumulator

loc Reference to a 16-bit location, either a data-memory location or a 16-bit register. Use any of these types of operands:

@0:*6bit* PAGE0-direct-addressing operand. Data page is 0. Offset is specified by *6bit*, a 6-bit constant from 0 to 63. PAGE0 direct addressing mode is available only when PAGE0 = 1.

@6bit DP-direct-addressing operand. Data page is specified by the DP. Offset is specified by *6bit*, a 6-bit constant from 0 to 63.

@reg Register-addressing operand. For *reg*, specify one of these register names:

AH	AL	PL	PH	T	SP
AR0	AR1	AR2	AR3	AR4	AR5
AR6	AR7				

–SP[6bit***]** PAGE0-stack-addressing operand. Address in data memory is specified as 0:(SP – *6bit*), where 0: indicates that the six MSBs are 0s and *6bit* is a 6-bit constant from 0 to 63. PAGE0 stack addressing mode is available only when PAGE0 = 0.

******ind* Indirect-addressing operand. Select one of the following operands (*x* is a number from 0 to 5; *n* is 6 or 7; *z* is a number from 0 to 7; *3bit* is a 3-bit constant):

*SP++	*AR*x*	*XAR*n*	*ARPz
*––SP	*AR*x*++	*XAR*n*++	*
	*––AR*x*	*––XARn	*++
	*+AR*x*[AR0]	*+XARn[AR0]	*––
	*+AR*x*[AR1]	*+XARn[AR1]	*0++
	*+AR*x*[*3bit*]	*+XARn[*3bit*]	*0––
	*AR6%++		

P Product register

T Multiplicand register

Opcode

Syntax 1: **MPYXU ACC, T,** *loc*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	0	0	0	0	See section 5.7.2 on page 5-23.							

Syntax 2: **MPYXU P, T,** *loc*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	0	0	1	0	See section 5.7.2 on page 5-23.							

Description

The T register value is multiplied by the value referenced by *loc*, and the result is stored to the specified destination (ACC or P). The T register value is treated as a signed number. The other multiplicand is treated as an unsigned number.

Execution

Syntax 1: **MPYXU ACC, T,** *loc*

[T] × [addressed location] → ACC
[PC] + 1 → PC

[T] is treated as signed; [addressed location] is treated as unsigned.

Syntax 2: **MPYXU P, T,** *loc*

[T] × [addressed location] → P
[PC] + 1 → PC

[T] is treated as signed; [addressed location] is treated as unsigned.

Status Bits

Syntax 1: **MPYXU ACC, T,** *loc*

OVM, SXM	Neither affects the operation.
C, OVC, V	None of these are affected by the operation.
N	If bit 31 of the result in ACC is 1, N is set; otherwise, N is cleared.
Z	If the result in ACC is 0, Z is set; otherwise, Z is cleared.

Syntax 2: **MPYXU P, T,** *loc*

OVM, SXM	Neither affects the operation.
C, N, V, Z	None of these are affected by the operation.

Repeat

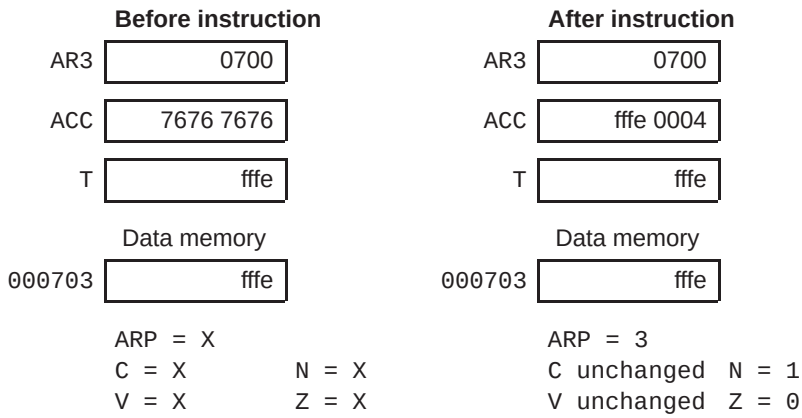
This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Words

1

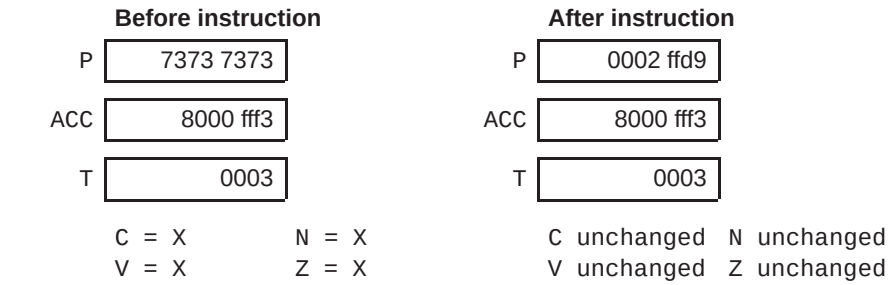
Example 1

```
MPYXU  ACC, T, *+AR3[3]

; T value signed. Other value unsigned.
; ACC = [T] × (value at 00 0703₁₆) = –2 × 65 534 = fffe 0004₁₆
```



Example 2

```
MPYXU  P, T, @AL

; T value signed. Other value unsigned.
; P = [T] × [AL] = 0003₁₆ × fff3₁₆ = 3 × 65 523 = 0002 ffd9₁₆
```



Syntax **NASP**

Operands None

Opcode

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	0	0	1	0	1	1	1

Description If the SPA bit is 1, the NASP instruction decrements the stack pointer (SP) by 1 and then clears SPA (in the decode 2 phase of the pipeline). This undoes a stack-pointer alignment performed earlier by the ASP instruction.

If the SPA bit is 0, the NASP instruction performs no operation.

Execution

If [SPA] = 1
    [SP] – 1 → SP
    0 → SPA
[PC] + 1 → PC

Status Bits

SPA	If SPA = 1 before the operation, SPA is cleared.
OVM, SXM	Neither affects the operation.
C, N, V, Z	None are affected by the operation.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Words 1

Example

```
NASP
; For SPA = 1
```

Before instruction		After instruction	
SP	0008	SP	0007
SPA = 1		SPA = 0	

```
; For SPA = 0
```

Before instruction		After instruction	
SP	0008	SP	0008
SPA = 0		SPA = 0 (unchanged)	

Syntax

1: **NEG ACC**

2: **NEG A***X*

Operands

ACC Accumulator

A*X* Use AH (the high word of the accumulator) or AL (the low word of the accumulator).

Opcode

Syntax 1: **NEG ACC**

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

Syntax 2: **NEG A***X*

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

Note: If AL is used in the instruction, AX = 0; if AH is used, AX = 1.

Description

The NEG instruction replaces the specified value (ACC, AH, or AL) with its negative. For example, –5 is replaced with 5. If the value is 0, it remains 0.

Execution

Syntax 1: **NEG ACC**

If ACC = $8000\ 0000_{16}$
    If OVM = 1
        $7FFF\ FFFF_{16} \rightarrow$ ACC
    else
        $8000\ 0000_{16} \rightarrow$ ACC
    $1 \rightarrow V$
else
    –[ACC] $\rightarrow$ ACC

If ACC = $0000\ 0000_{16}$
    $1 \rightarrow C$
else
    $0 \rightarrow C$

[PC] + 1 $\rightarrow$ PC

Syntax 2: **NEG** **A***X*

If AX = 8000_{16}
    $8000_{16} \rightarrow$ AX
    $1 \rightarrow$ V
else
    –[AX] $\rightarrow$ AX

If AX = 0000_{16}
    $1 \rightarrow$ C
else
    $0 \rightarrow$ C

[PC] + 1 $\rightarrow$ PC

Status Bits

Syntax 1: **NEG** **ACC**

OVM	If ACC is 8000 0000_{16} at the start of the operation, this is considered an overflow value, and the ACC value after the operation depends on OVM: OVM = 0 ACC is filled again with 8000 0000_{16}. OVM = 1 ACC is filled with its greatest positive number, 7FFF $FFFF_{16}$.
SXM	SXM does not affect the operation.
C	If ACC = 0, C is set; otherwise, C is cleared.
N	If bit 31 of the result in ACC is 1, N is set; otherwise, N is cleared.
OVC	OVC is not affected by the operation.
V	If ACC = 8000 0000_{16} at the start of the operation, this is considered an overflow value, and V is set. otherwise, V is not affected.
Z	If ACC = 0 after the operation, Z is set; otherwise, Z is cleared.

Syntax 2: **NEG** **A***X*

OVM, SXM	Neither affects the operation.
C	If AL/AH = 0, C is set; otherwise, C is cleared.
N	If bit 15 of the result in AL/AH is 1, N is set; otherwise, N is cleared.
V	If AL/AH = 8000_{16} at the start of the operation, it is considered an overflow value and V is set. Otherwise, V is not affected.
Z	If AL/AH = 0 after the operation, Z is set; otherwise, Z is cleared.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Words 1

Example 1

```
NEG    ACC              ; ACC = 0
```

Before instruction		After instruction	
ACC	0000 0000	ACC	0000 0000
C = X	N = X	C = 1	N = 0
V = X	Z = X	V unchanged	Z = 1

Example 2

```
NEG    ACC              ; ACC = 8000 0000₁₆
; For OVM = 0: ACC kept at its greatest negative value
```

Before instruction		After instruction	
ACC	8000 0000	ACC	8000 0000
OVC = X		OVC unchanged	
C = X	N = X	C = 0	N = 1
V = X	Z = X	V = 1	Z = 0

```
; For OVM = 1: ACC changed to its greatest positive value
```

Before instruction		After instruction	
ACC	8000 0000	ACC	7fff ffff
OVC = X		OVC unchanged	
C = X	N = X	C = 0	N = 0
V = X	Z = X	V = 1	Z = 0

Example 3

```
NEG    ACC              ; ACC not 0 and not 8000 0000₁₆
```

Before instruction		After instruction	
ACC 0000 0005		ACC ffff fffb	–5
C = X	N = X	C = 0	N = 1
V = X	Z = X	V unchanged	Z = 0

Example 4

```
NEG    AL               ; AL = 8000₁₆
```

Before instruction		After instruction	
ACC 3535 8000		ACC 3535 8000	
C = X	N = X	C = 0	N = 1
V = X	Z = X	V = 1	Z = 0

Example 5

```
NEG    AH               ; AH not 0 and not 8000₁₆
```

Before instruction		After instruction	
ACC fff3 aaaa		ACC 000d aaaa	
C = X	N = X	C = 0	N = 0
V = X	Z = X	V unchanged	Z = 0

Syntax

1: **NOP**

2: **NOP** ******ind*

Operands

******ind* Indirect-addressing operand. Select one of the following operands (*x* is a number from 0 to 5; *n* is 6 or 7; *z* is a number from 0 to 7; *3bit* is a 3-bit constant):

*SP++	*AR*x*	*XAR*n*	*ARPz
*––SP	*AR*x*++	*XAR*n*++	*
	*––AR*x*	*––XARn	*++
	*+AR*x*[AR0]	*+XARn[AR0]	*––
	*+AR*x*[AR1]	*+XARn[AR1]	*0++
	*+AR*x*[*3bit*]	*+XARn[*3bit*]	*0––
	*AR6%++		

Opcode

1: **NOP**

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	0	0	0	0	0	0	0

2: **NOP** ******ind*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	0	1	1	1	See section 5.7.2 on page 5-23.							

Description

If an operand is used and it specifies an increment or decrement, the stack pointer or the specified auxiliary register is changed. Otherwise, no operation is performed. Data memory is not read from or written to during the execution of the NOP instruction.

Execution

Syntax 1: **NOP**

No operation performed.
[PC] + 1 → PC

Syntax 2: **NOP** ******ind*

Modify the auxiliary register and/or ARP, or SP, as per the indirect-addressing operand.
[PC] + 1 → PC

Status Bits

OVM, SXM Neither affects the operation.

C, N, V, Z None of these are affected by the operation.

Repeat

This instruction is repeatable. See the description for the RPT instruction.

If you use indirect addressing, an auxiliary register or the stack pointer (SP) can be incremented or decremented during every repetition.

Once the repeat loop begins, it cannot be interrupted by any interrupt.

Words

1

Example 1

```
NOP    *ARP5

; Load ARP with 5. Do nothing else.
```

Before instruction		After instruction	
ARP = X		ARP = 5	
C = X	N = X	C unchanged	N unchanged
V = X	Z = X	V unchanged	Z unchanged

Example 2

```
RPT  #2   ||   NOP  *AR2++

; Increment AR2 for each NOP cycle.
```

Before instruction			After instruction		
AR2	0003		AR2	0006	
ARP = X			ARP = 2		
C = X	N = X		C unchanged	N unchanged	
V = X	Z = X		V unchanged	Z unchanged	

Syntax

1: **NORM ACC,** *aux***++**

2: **NORM ACC,** *aux***– –**

Operands

ACC Accumulator

aux Use one of the following auxiliary register names:
AR0 AR1 AR2 AR3 AR4 AR5
XAR6 XAR7

Opcode

Syntax 1: **NORM ACC,** *aux***++**

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

Syntax 2: **NORM ACC,** *aux***– –**

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

The following table shows the relationship between *aux* and auxNumber:

aux	**auxNumber**	***aux***	**auxNumber**
AR0	0	AR4	4
AR1	1	AR5	5
AR2	2	XAR6	6
AR3	3	XAR7	7

Description

The NORM instruction normalizes a signed number in ACC by finding the magnitude of the number. An exclusive-OR operation is performed on ACC bits 31 and 30. If the bits are the same, then the content of ACC is shifted to the left by 1 to eliminate the extra sign bit. In addition, the specified auxiliary register is incremented (++) or decremented (––) by 1. If you specify XAR6 or XAR7, the entire 22-bit value is affected. Multiple executions of the NORM instruction may be required to remove all extra sign bits.

If bits 31 and 30 are different, ACC and the auxiliary register are not modified, and the TC bit is set.

Execution

If ACC = $0000\ 0000_{16}$
    $1 \rightarrow$ TC
    [PC] + $1 \rightarrow$ PC
else
    If (ACC(31) XOR ACC(30)) = 0
        $0 \rightarrow$ TC
        [ACC] << $1 \rightarrow$ ACC
        If *aux***++**
            [specified auxiliary register] + $1 \rightarrow$ specified auxiliary register
        If *aux***– –**
            [specified auxiliary register] – $1 \rightarrow$ specified auxiliary register
    else
        $1 \rightarrow$ TC
    [PC] + $1 \rightarrow$ PC

Status Bits

OVM, SXM	Neither affects the operation.
C, OVC, V	None of these are affected by the operation.
TC	If the operation set TC, no normalization was needed (ACC did not need to be modified). If the operation cleared TC, bits 31 and 30 were the same and, as a result, ACC was shifted left by 1.
N	If bit 31 of the result in ACC is 1, N is set; otherwise, N is cleared.
Z	If ACC = 0 after the operation, Z is set; otherwise, Z is cleared.

Repeat

This instruction is repeatable. See the description for the RPT instruction.

When the NORM instruction is repeated, the repetitions do not automatically stop when the normalization is complete. Once a repeat loop starts, it continues until it is done. The value in the accumulator remains the same until the rest of the repetitions are done. If you only want the NORM instruction to execute until normalization is done, you can create a loop that checks the value of the TC bit. When TC = 1, normalization is complete.

Once the repeat loop begins, it cannot be interrupted by any interrupt.

Words

1

Example 1

```
NORM   ACC, XAR7--

; If two MSBs of ACC the same, modify ACC and auxiliary
; register, and clear TC.
```

Before instruction		After instruction	
ACC	2000 5000	ACC	4000 a000
XAR7	0f 0001	XAR7	0f 0000

```
ARP = X                    ARP unchanged
TC = X                     TC = 0
C = X       N = X          C unchanged  N = 0
V = X       Z = X          V unchanged  Z = 0
```

Example 2

```
NORM   ACC, AR3++

; If two MSBs of ACC not the same, do not modify ACC and
; auxiliary register, but set TC.
```

Before instruction		After instruction	
ACC	4000 7000	ACC	4000 7000
AR3	0001	AR3	0001

```
ARP = X                    ARP unchanged
TC = X                     TC = 1
C = X       N = X          C unchanged  N = 0
V = X       Z = X          V unchanged  Z = 0
```

Example 3

```
Loop   NORM   ACC, AR4++
       SB     Loop,NTC

; Loop until TC = 1. Use AR4 as counter.
```

Before loop		After exiting loop	
ACC	e000 0000	ACC	8000 0000
AR4	0000	AR4	0002

```
ARP = X                    ARP unchanged
TC = X                     TC = 1
C = X       N = X          C unchanged  N = 1
V = X       Z = X          V unchanged  Z = 0
```

Syntax

1: **NOT ACC**

2: **NOT A***X*

Operands

ACC Accumulator

A*X* Use AH (the high word of the accumulator) or AL (the low word of the accumulator).

Opcode

Syntax 1: **NOT ACC**

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

Syntax 2: **NOT A***X*

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

Note: If AL is used in the instruction, AX = 0; if AH is used, AX = 1.

Description

The content of the specified register (ACC, AH, or AL) is replaced with its complement. For example, $F0F0_{16}$ would be replaced with $0F0F_{16}$.

Execution

Syntax 1: **NOT ACC**

([ACC] XOR FFFF $FFFF_{16}$) $\rightarrow$ ACC
[PC] + 1 $\rightarrow$ PC

XOR = exclusive OR

Syntax 2: **NOT A***X*

([AX] XOR $FFFF_{16}$) $\rightarrow$ AX
[PC] + 1 $\rightarrow$ PC

XOR = exclusive OR

Status Bits

OVM, SXM	Neither affects the operation.
C, OVC, V	None of these are affected by the operation.
N	If the most significant bit of the result in ACC/AL/AH is 1, N is set; otherwise, N is cleared.
Z	If ACC/AL/AH = 0 after the operation, Z is set; otherwise, Z is cleared.

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Words

1

Example 1

```
NOT    ACC
```

Before instruction

ACC 0011 0011

```
C = X        N = X
V = X        Z = X
```

After instruction

ACC ffee ffee

```
C unchanged  N = 1
V unchanged  Z = 0
```

Example 2

```
NOT    AH
```

Before instruction

ACC ffff 8000

```
C = X        N = X
V = X        Z = X
```

After instruction

ACC 0000 8000

```
C unchanged  N = 0
V unchanged  Z = 1
```

Example 3

```
NOT    AL
```

Before instruction

ACC ffff 8000

```
C = X        N = X
V = X        Z = X
```

After instruction

ACC ffff 7fff

```
C unchanged  N = 0
V unchanged  Z = 0
```

Syntax

1: **OR** **A***X*, *loc*
2: **OR** *loc*, **A***X*
3: **OR** **IER,** #*16BitMask*
4: **OR** **IFR,** #*16BitMask*
5: **OR** *loc*, #*16BitMask*

Operands

16BitMask 16-bit mask value from 0000_{16} to $FFFF_{16}$

A*X* Use AH (the high word of the accumulator) or AL (the low word of the accumulator).

loc Reference to a 16-bit location, either a data-memory location or a 16-bit register. Use any of these types of operands:

@**0:***6bit* PAGE0-direct-addressing operand. Data page is 0. Offset is specified by *6bit*, a 6-bit constant from 0 to 63. PAGE0 direct addressing mode is available only when PAGE0 = 1.

@*6bit* DP-direct-addressing operand. Data page is specified by the DP. Offset is specified by *6bit*, a 6-bit constant from 0 to 63.

@*reg* Register-addressing operand. For *reg*, specify one of these register names:

AH	AL	PL	PH	T	SP
AR0	AR1	AR2	AR3	AR4	AR5
AR6	AR7				

–SP[6bit***]*** PAGE0-stack-addressing operand. Address in data memory is specified as 0:(SP – *6bit*), where 0: indicates that the six MSBs are 0s and *6bit* is a 6-bit constant from 0 to 63. PAGE0 stack addressing mode is available only when PAGE0 = 0.

ind Indirect-addressing operand. Select one of the following operands (*x* is a number from 0 to 5; *n* is 6 or 7; *z* is a number from 0 to 7; *3bit* is a 3-bit constant):

*SP++	*AR*x*	*XAR*n*	*ARPz
*––SP	*AR*x*++	*XAR*n*++	*
	*––AR*x*	*––XARn	*++
	*+AR*x*[AR0]	*+XARn[AR0]	*––
	*+AR*x*[AR1]	*+XARn[AR1]	*0++
	*+AR*x*[*3bit*]	*+XARn[*3bit*]	*0––
	*AR6%++		

IER Interrupt enable register

IFR Interrupt flag register

Opcode

Syntax 1: **OR** **A***X*, *loc*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	1	0	1	AX	See section 5.7.2 on page 5-23.							

Note: If AL is used in the instruction, AX = 0; if AH is used, AX = 1.

Syntax 2: **OR** *loc*, **A***X*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	1	0	0	AX	See section 5.7.2 on page 5-23.							

Note: If AL is used in the instruction, AX = 0; if AH is used, AX = 1.

Syntax 3: **OR** **IER,** #*16BitMask*

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	0	1	0	0	0	1	1
16BitMask															

Syntax 4: **OR** **IFR,** #*16BitMask*

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	0	1	0	0	1	1	1
16BitMask															

Syntax 5: **OR** *loc*, #*16BitMask*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	0	1	0	See section 5.7.2 on page 5-23.							
16BitMask															

Description

The general form for the OR instruction is as follows:

OR *operand1*, *operand2*

The ALU calculates the OR of the values referenced by *operand1* and *operand2*. The result is stored to the location of *operand1*.

Execution

Syntax 1: **OR** **A***X*, *loc*

[AX] OR [addressed location] → AX
[PC] + 1 → PC

Syntax 2: **OR** *loc*, **A***X*

[addressed location] OR [AX] → addressed location
[PC] + 1 → PC

Syntax 3: **OR** **IER,** #*16BitMask*

[IER] OR 16-bit mask value → IER
[PC] + 2 → PC

Syntax 4: **OR** **IFR,** #*16BitMask*

[IFR] OR 16-bit mask value → IFR
[PC] + 2 → PC

Syntax 5: **OR** *loc*, #*16BitMask*

[addressed location] OR 16-bit mask value → addressed location
[PC] + 2 → PC

Status Bits

Syntaxes 1, 2, and 5:

OVM, SXM	Neither affects the operation.
C, V	Neither is affected by the operation.
N	If bit 15 of the result is 1, N is set; otherwise, N is cleared.
Z	If the result is 0, Z is set; otherwise, Z is cleared.

Syntaxes 3 and 4 (IER and IFR receive the results):

OVM, SXM	Neither affects the operation.
C, N, V, Z	None of these are affected by the operation.

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Words

Syntaxes 1 and 2: 1

Syntaxes 3, 4, and 5: 2

Example 1

```
OR     AL, *+XAR7[AR0]
; AL = [AL] OR (value at 00 0084_16)
```

Before instruction		After instruction	
XAR7	00 0082	XAR7	00 0082
AR0	0002	AR0	0002
ACC	5656 4000	ACC	5656 7000
Data memory		Data memory	
000084	3000	000084	3000

```
ARP = X                    ARP = 7
C = X       N = X          C unchanged  N = 0
V = X       Z = X          V unchanged  Z = 0
```

Example 2

```
OR     @4, #0000000000001010b     ; Binary mask used
```

Before instruction		After instruction	
DP	0002	DP	0002
Data memory		Data memory	
000084	8000	000084	800a

```
C = X       N = X          C unchanged  N = 1
V = X       Z = X          V unchanged  Z = 0
```

Example 3

```
OR     IER, #0c000h               ; Hexadecimal mask used
```

Before instruction		After instruction	
IER	0001	IER	c001

```
C = X       N = X          C unchanged  N unchanged
V = X       Z = X          V unchanged  Z unchanged
```

Syntax **ORB** **A***X*, #*8BitMask*

Operands

8BitMask	8-bit mask value from 00_{16} to FF_{16}
A*X*	Use AH (the high word of the accumulator) or AL (the low word of the accumulator).

Opcode

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

Note: If AL is used in the instruction, AX = 0; if AH is used, AX = 1.

Description The ALU calculates the OR of the specified accumulator half (AH or AL) and the 8-bit mask value. The result is stored to the specified accumulator half.

Execution

[AX] OR (8-bit mask value AND $00FF_{16}$) → AX
[PC] + 1 → PC

Status Bits

OVM, SXM	Neither affects the operation.
C, V	Neither is affected by the operation.
N	If bit 15 of the result is 1, N is set; otherwise, N is cleared.
Z	If the result is 0, Z is set; otherwise, Z is cleared.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Words 1

Example

```
ORB     AL, #0c0h          ; Hexadecimal mask used
```

Before instruction

ACC: eeee 0001

```
C = X     N = X
V = X     Z = X
```

After instruction

ACC: eeee 00c1

```
C unchanged  N = 0
V unchanged  Z = 0
```

Syntax

1: **POP** *loc*
2: **POP** **DBGIER**
3: **POP** **DP**
4: **POP** **ST0**
5: **POP** **ST1**
6: **POP** **T:ST0**
7: **POP** **DP:ST1**
8: **POP** **PH:PL**
9: **POP** **AR1:AR0**
10: **POP** **AR3:AR2**
11: **POP** **AR5:AR4**
12: **POP** **XAR***n*

Operands

AR0–AR5 Auxiliary registers 0 through 5

DBGIER Debug interrupt enable register

DP Data page pointer

loc Reference to a 16-bit location, either a data-memory location or a 16-bit register. Use any of these types of operands:

- **@0:***6bit* PAGE0-direct-addressing operand. Data page is 0. Offset is specified by *6bit*, a 6-bit constant from 0 to 63. PAGE0 direct addressing mode is available only when PAGE0 = 1.
- *@6bit* DP-direct-addressing operand. Data page is specified by the DP. Offset is specified by *6bit*, a 6-bit constant from 0 to 63.
- *@reg* Register-addressing operand. For *reg*, specify one of these register names:

AH	AL	PL	PH	T	SP
AR0	AR1	AR2	AR3	AR4	AR5
AR6	AR7				

- ***–SP[***6bit***]*** PAGE0-stack-addressing operand. Address in data memory is specified as 0:(SP – *6bit*), where 0: indicates that the six MSBs are 0s and *6bit* is a 6-bit constant from 0 to 63. PAGE0 stack addressing mode is available only when PAGE0 = 0.

ind Indirect-addressing operand. Select one of the following operands (*x* is a number from 0 to 5; *n* is 6 or 7; *z* is a number from 0 to 7; *3bit* is a 3-bit constant):

*SP++	*AR*x*	*XAR*n*	*ARPz
*––SP	*AR*x*++	*XAR*n*++	*
	*––AR*x*	*––XARn	*++
	*+AR*x*[AR0]	*+XARn[AR0]	*––
	*+AR*x*[AR1]	*+XARn[AR1]	*0++
	*+AR*x*[*3bit*]	*+XARn[*3bit*]	*0––
	*AR6%++		

PH High word of the product register (P)

PL Low word of the product register(P)

ST0, ST1 Status registers

T Multiplicand register

XAR*n* Use XAR6 or XAR7.

Opcode

Syntax 1: **POP** *loc*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	1	0	1	0	See section 5.7.2 on page 5-23.							

Syntax 2: **POP DBGIER**

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	0	0	1	0	0	1	0

Syntax 3: **POP DP**

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	0	0	0	0	0	1	1

Syntax 4: **POP ST0**

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	0	0	1	0	0	1	1

Syntax 5: **POP ST1**

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	0	0	0	0	0	0	0

Syntax 6: **POP T:ST0**

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	0	0	1	0	1	0	1

Syntax 7: **POP DP:ST1**

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	0	0	0	0	0	0	1

Syntax 8: **POP PH:PL**

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	0	0	1	0	0	0	1

Syntax 9: **POP AR1:AR0**

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	0	0	0	0	1	1	1

Syntax 10: **POP AR3:AR2**

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	0	0	0	0	1	0	1

Syntax 11: **POP AR5:AR4**

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	0	0	0	0	1	1	0

Syntax 12: **POP** **XAR***n*

For XAR6:

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

Note: This is the opcode for the following equivalent instruction: MOV XAR6, *SP– –.

For XAR7:

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

Note: This is the opcode for the following equivalent instruction: MOV XAR7, *SP– –.

Description

Syntaxes 1–5 each pop a 16-bit value off the stack. The stack pointer (SP) is decremented by 1. Then the value at that address is popped off the stack and loaded into the data-memory location or register specified in the instruction.

Syntaxes 6–11 each pop a 32-bit value off the stack. The general form for these syntaxes is as follows:

POP *Register2*:*Register1*

The following operation is performed:

1) Decrement SP by 2.
2) Read a 32-bit value from the address in SP and the next.
3) Load *Register1* with the value referenced by SP, and load *Register2* with the value from the next address. For example, the POP AR3:AR2 instruction loads AR2 with the value referenced by SP and then loads AR3 with the value from the next address.

Syntax 12 pops a 22-bit value off the stack and loads it to XAR6 or XAR7.

1) Decrement SP by 2.
2) Read the 32-bit value at the addresses referenced by SP and SP + 1.
3) Load XARn(15:0) with the value referenced by SP, and load XARn(21:16) with the six LSBs at the next address. For example, if the consecutive stack values are $FFFF_{16}$ and $003A_{16}$, the POP XAR6 instruction loads 3A $FFFF_{16}$ to XAR6.

The ’27x core expects memory wrappers or peripheral-interface logic to force 32-bit accesses, like those of syntaxes 6–11, to align to even addresses. This alignment is described in section 6.2 on page 6-31.

Execution

Syntax 1: **POP** *loc*

[SP] – 1 → SP
[address referenced by SP] → addressed location
[PC] + 1 → PC

Syntax 2: **POP DBGIER**

[SP] – 1 → SP
[address referenced by SP] → DBGIER
[PC] + 1 → PC

Syntax 3: **POP DP**

[SP] – 1 → SP
[address referenced by SP] → DP
[PC] + 1 → PC

Syntax 4: **POP ST0**

[SP] – 1 → SP
[address referenced by SP] → ST0
[PC] + 1 → PC

Syntax 5: **POP ST1**

[SP] – 1 → SP
[address referenced by SP] → ST1
[PC] + 1 → PC

Syntax 6: **POP T:ST0**

[SP] – 2 → SP
[address referenced by SP] → ST0
[address referenced by (SP + 1)] → T
[PC] + 1 → PC

Syntax 7: **POP DP:ST1**

[SP] – 2 → SP
[address referenced by SP] → ST1
[address referenced by (SP + 1)] → DP
[PC] + 1 → PC

Syntax 8: **POP PH:PL**

[SP] – 2 → SP
[address referenced by SP] → PL
[address referenced by (SP + 1)] → PH
[PC] + 1 → PC

Syntax 9: **POP AR1:AR0**

[SP] – 2 → SP
[address referenced by SP] → AR0
[address referenced by (SP + 1)] → AR1
[PC] + 1 → PC

Syntax 10: **POP AR3:AR2**

[SP] – 2 → SP
[address referenced by SP] → AR2
[address referenced by (SP + 1)] → AR3
[PC] + 1 → PC

Syntax 11: **POP AR5:AR4**

[SP] – 2 → SP
[address referenced by SP] → AR4
[address referenced by (SP + 1)] → AR5
[PC] + 1 → PC

Syntax 12: **POP XAR***n*

[SP] – 2 → SP
[address referenced by SP] → XARn(15:0)
[6 LSBs at address referenced by (SP + 1)] → XARn(21:16)
[PC] + 1 → PC

Status Bits

Syntax 1: **POP** *loc*

OVM, SXM	Neither affects the operation.
C, V	C and V are not affected by the operation.
N	If AH or AL is loaded, N is affected as follows: If bit 15 of AH/AL is 1, N is set; otherwise, N is cleared. If neither AH nor AL is loaded, N is not affected.
Z	If AH or AL is loaded, Z is affected as follows: If AH/AL is 0, Z is set; otherwise, Z is cleared. If neither AH nor AL is loaded, Z is not affected.

Syntaxes 2, 3, and 8–12:

OVM, SXM	Neither affects the operation.
C, N, V, Z	None of these are affected by the operation.

Syntax 4: **POP ST0**

Neither OVM nor SXM affects the operation. The value popped off the stack replaces the following bit values of ST0: OVC, PM, V, N, Z, C, TC, OVM, and SXM.

Syntax 5: **POP ST1**

Neither OVM nor SXM affects the operation. The value popped off the stack replaces the following bit values of ST1: ARP, EALLOW, SPA, VMAP, PAGE0, DBGM, and INTM. The following bits are *not* replaced: IDLESTAT and LOOP.

Syntax 6: **POP T:ST0**

Neither OVM nor SXM affects the operation. The first 16 bits popped off the stack replace the following bit values of ST0: OVC, PM, V, N, Z, C, TC, OVM, and SXM.

Syntax 7: **POP DP:ST1**

Neither OVM nor SXM affects the operation. The first 16 bits popped off the stack replace the following bit values of ST1: ARP, EALLOW, SPA, VMAP, PAGE0, DBGM, and INTM. The following bits are *not* replaced: IDLESTAT and LOOP.

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Words

1

Example 1

```
POP    @T

; Status bits not affected. Decrement SP by 1 before read.
```

	Before instruction	After instruction
SP	0005	0004
T	7676	0007
Data memory		
000004	0007	0007

```
C = X       N = X          C unchanged  N unchanged
V = X       Z = X          V unchanged  Z unchanged
```

Example 2

```
POP    @AL

; Status bits affected. Decrement SP by 1 before read.
```

	Before instruction	After instruction
SP	0005	0004
ACC	3939 3939	3939 ffff
Data memory		
000004	ffff	ffff

```
C = X       N = X          C unchanged  N = 1
V = X       Z = X          V unchanged  Z = 0
```

Example 3

```
POP    PH:PL

; Status bits not affected. Decrement SP by 2 before read.
; Read aligned to even address.
```

	Before instruction	After instruction
SP	0005	0003
P	4242 4242	bbbb eeee
Data memory		
000002	eeee	eeee
000003	bbbb	bbbb

```
C = X       N = X          C unchanged  N unchanged
V = X       Z = X          V unchanged  Z unchanged
```

Example 4

```
POP    AR3:AR2
; Status bits not affected. Decrement SP by 2 before read.
```

	Before instruction	After instruction
SP	0004	0002
AR2	5959	cccc
AR3	8282	aaaa

Data memory	Before instruction	After instruction
000002	cccc	cccc
000003	aaaa	aaaa

```
C = X        N = X          C unchanged  N unchanged
V = X        Z = X          V unchanged  Z unchanged
```

Syntax **PREAD** *loc*, ***XAR7**

Operands

loc Reference to a 16-bit location, either a data-memory location or a 16-bit register. Use any of these types of operands:

- **@0:***6bit* PAGE0-direct-addressing operand. Data page is 0. Offset is specified by *6bit*, a 6-bit constant from 0 to 63. PAGE0 direct addressing mode is available only when PAGE0 = 1.
- *@6bit* DP-direct-addressing operand. Data page is specified by the DP. Offset is specified by *6bit*, a 6-bit constant from 0 to 63.
- *@reg* Register-addressing operand. For *reg*, specify one of these register names:

AH	AL	PL	PH	T	SP
AR0	AR1	AR2	AR3	AR4	AR5
AR6	AR7				

- ***–SP[***6bit***]** PAGE0-stack-addressing operand. Address in data memory is specified as 0:(SP – *6bit*), where 0: indicates that the six MSBs are 0s and *6bit* is a 6-bit constant from 0 to 63. PAGE0 stack addressing mode is available only when PAGE0 = 0.
- *****ind* Indirect-addressing operand. Select one of the following operands (*x* is a number from 0 to 5; *n* is 6 or 7; *z* is a number from 0 to 7; *3bit* is a 3-bit constant):

*SP++	*AR*x*	*XAR*n*	*ARPz
*––SP	*AR*x*++	*XAR*n*++	*
	*––AR*x*	*––XARn	*++
	*+AR*x*[AR0]	*+XARn[AR0]	*––
	*+AR*x*[AR1]	*+XARn[AR1]	*0++
	*+AR*x*[*3bit*]	*+XARn[*3bit*]	*0––
	*AR6%++		

XAR7 Extended auxiliary register XAR7

Opcode

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	0	See section 5.7.2 on page 5-23.							

Description XAR7 holds the address of the program-memory value to be read. The program-memory value is loaded to the data-memory location or register referenced by *loc*.

Execution [program-memory location addressed by XAR7] → location addressed by *loc*
[PC] + 1 → PC

Status Bits

OVM, SXM	Neither affects the operation.
C, V	C and V are not affected by the operation.
N	If AH or AL is loaded, N is affected as follows: If bit 15 of AH/AL is 1, N is set; otherwise, N is cleared. If neither AH nor AL is loaded, N is not affected.
Z	If AH or AL is loaded, Z is affected as follows: If AH/AL is 0, Z is set; otherwise, Z is cleared. If neither AH nor AL is loaded, Z is not affected.

Repeat

This instruction is repeatable. See the description for the RPT instruction.

At the start of the first execution, the value in XAR7 is loaded to the fetch counter (FC). When the PREAD instruction is repeated, the FC is incremented by 1 during each repetition. Thus, a new program-memory value is read during each repetition. To have new data-memory locations loaded each time, use indirect addressing to reference data memory. Otherwise, the same destination in data memory will be overwritten during each repetition.

Once the repeat loop begins, it cannot be interrupted by any interrupt.

Words

1

Example 1

```
PREAD  @5, *XAR7

; Not loading AH or AL – status bits not affected.
```

	Before instruction	After instruction
DP	0002	0002
XAR7	00017c	00017c
Data memory 000085	4545	0002
Program memory 00017c	0002	0002

```
ARP = X                 ARP unchanged
C = X      N = X        C unchanged   N unchanged
V = X      Z = X        V unchanged   Z unchanged
```

Example 2

```
PREAD  @AH, *XAR7
```

	Before instruction	After instruction
XAR7	00017c	00017c
ACC	8181 8007	0002 8007

Program memory

	Before instruction	After instruction
00017c	0002	0002

```
ARP = X                    ARP unchanged
C = X       N = X          C unchanged   N = 0
V = X       Z = X          V unchanged   Z = 0
```

Example 3

```
RPT #2 || PREAD   *AR2++, *XAR7
```

	Before instruction	After instruction
AR2	0083	0086
XAR7	00017b	00017b

Data memory

	Before instruction	After instruction
000083	7676	0a0a
000084	6565	0b0b
000085	2727	0c0c

Program memory

	Before instruction	After instruction
00017b	0a0a	0a0a
00017c	0b0b	0b0b
00017d	0c0c	0c0c

```
ARP = X                    ARP = 2
C = X       N = X          C unchanged   N = unchanged
V = X       Z = X          V unchanged   Z = unchanged
```

Syntax

1: **PUSH** *loc*
2: **PUSH DBGIER**
3: **PUSH DP**
4: **PUSH IFR**
5: **PUSH ST0**
6: **PUSH ST1**
7: **PUSH T:ST0**
8: **PUSH DP:ST1**
9: **PUSH PH:PL**
10: **PUSH AR1:AR0**
11: **PUSH AR3:AR2**
12: **PUSH AR5:AR4**
13: **PUSH XAR***n*

Operands

AR0–AR5 Auxiliary registers 0 through 5

DBGIER Debug interrupt enable register

DP Data page pointer

IFR Interrupt flag register

loc Reference to a 16-bit location, either a data-memory location or a 16-bit register. Use any of these types of operands:

- **@0:***6bit* PAGE0-direct-addressing operand. Data page is 0. Offset is specified by *6bit*, a 6-bit constant from 0 to 63. PAGE0 direct addressing mode is available only when PAGE0 = 1.
- *@6bit* DP-direct-addressing operand. Data page is specified by the DP. Offset is specified by *6bit*, a 6-bit constant from 0 to 63.
- *@reg* Register-addressing operand. For *reg*, specify one of these register names:

AH	AL	PL	PH	T	SP
AR0	AR1	AR2	AR3	AR4	AR5
AR6	AR7				

- ***–SP[***6bit***]*** PAGE0-stack-addressing operand. Address in data memory is specified as 0:(SP – *6bit*), where 0: indicates that the six MSBs are 0s and *6bit* is a 6-bit constant from 0 to 63. PAGE0 stack addressing mode is available only when PAGE0 = 0.

ind Indirect-addressing operand. Select one of the following operands (*x* is a number from 0 to 5; *n* is 6 or 7; *z* is a number from 0 to 7; *3bit* is a 3-bit constant):

*SP++	*AR*x*	*XAR*n*	*ARPz
*––SP	*AR*x*++	*XAR*n*++	*
	*––AR*x*	*––XARn	*++
	*+AR*x*[AR0]	*+XARn[AR0]	*––
	*+AR*x*[AR1]	*+XARn[AR1]	*0++
	*+AR*x*[*3bit*]	*+XARn[*3bit*]	*0––
	*AR6%++		

PH High word of the product register (P)

PL Low word of the product register(P)

ST0, ST1 Status registers

T Multiplicand register

XAR*n* Use XAR6 or XAR7.

Opcode

Syntax 1: **PUSH** *loc*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	0	1	0	See section 5.7.2 on page 5-23.							

Syntax 2: **PUSH DBGIER**

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	0	0	0	1	1	1	0

Syntax 3: **PUSH DP**

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	0	0	0	1	0	1	1

Syntax 4: **PUSH IFR**

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	0	0	0	1	0	1	0

Syntax 5: **PUSH ST0**

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	0	0	1	1	0	0	0

Syntax 6: **PUSH ST1**

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	0	0	0	1	0	0	0

Syntax 7: **PUSH T:ST0**

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	0	0	1	1	0	0	1

Syntax 8: **PUSH DP:ST1**

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	0	0	0	1	0	0	1

Syntax 9: **PUSH PH:PL**

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	0	0	1	1	1	0	1

Syntax 10: **PUSH AR1:AR0**

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	0	0	0	1	1	0	1

Syntax 11: **PUSH AR3:AR2**

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	0	0	0	1	1	1	1

Syntax 12: **PUSH AR5:AR4**

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	0	0	0	1	1	0	0

Syntax 13: **PUSH XAR***n*

For XAR6:

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

Note: This is the opcode for the following equivalent instruction: MOV *SP++, XAR6.

For XAR7:

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

Note: This is the opcode for the following equivalent instruction: MOV *SP++, XAR7.

Description

Syntaxes 1–6 each push a 16-bit value onto the stack. The register or data-memory value is saved to the address pointed to by the stack pointer (SP). Then SP is incremented by 1.

Syntaxes 7–12 each push a 32-bit register pair onto the stack. The general form for these syntaxes is as follows:

PUSH *Register2*:*Register1*

The following operation is performed:

1) Read the specified register pair using one 32-bit read operation.
2) Save *Register1* to the address referenced by SP, and save *Register2* to the next address. For example, the PUSH AR3:AR2 instruction saves AR2 to the address referenced by SP and then saves AR3 to the next address.
3) Increment SP by 2.

Syntax 13 pushes a 22-bit value (XAR6 or XAR7) onto the stack as a 32-bit value, as follows:

1) Read XARn (n = 6 or 7) using one 32-bit read operation.
2) Save XARn(15:0) to the address referenced by SP, and save XARn(21:16) with 10 leading 0s to the next address. For example, if XAR6 = 3A $FFFF_{16}$, the PUSH XAR6 instruction saves $FFFF_{16}$ to the address referenced by SP and then saves $003A_{16}$ to the next address.
3) Increment SP by 2.

The ’27x core expects memory wrappers or peripheral-interface logic to force 32-bit accesses, like those of syntaxes 7–13, to align to even addresses. This alignment is described in section 6.2 on page 6-31.

Execution

Syntax 1: **PUSH** *loc*

[addressed location] → address referenced by SP
[SP] + 1 → SP
[PC] + 1 → PC

Syntax 2: **PUSH** **DBGIER**

[DBGIER] → address referenced by SP
[SP] + 1 → SP
[PC] + 1 → PC

Syntax 3: **PUSH** **DP**

[DP] → address referenced by SP
[SP] + 1 → SP
[PC] + 1 → PC

Syntax 4: **PUSH** **IFR**

[IFR] → address referenced by SP
[SP] + 1 → SP
[PC] + 1 → PC

Syntax 5: **PUSH** **ST0**

[ST0] → address referenced by SP
[SP] + 1 → SP
[PC] + 1 → PC

Syntax 6: **PUSH** **ST1**

[ST1] → address referenced by SP
[SP] + 1 → SP
[PC] + 1 → PC

Syntax 7: **PUSH** **T:ST0**

[ST0] → address referenced by SP
[T] → address referenced by (SP + 1)
[SP] + 2 → SP
[PC] + 1 → PC

Syntax 8: **PUSH** **DP:ST1**

[ST1] → address referenced by SP
[DP] → address referenced by (SP + 1)
[SP] + 2 → SP
[PC] + 1 → PC

Syntax 9: **PUSH PH:PL**

[PL] → address referenced by SP
[PH] → address referenced by (SP + 1)
[SP] + 2 → SP
[PC] + 1 → PC

Syntax 10: **PUSH AR1:AR0**

[AR0] → address referenced by SP
[AR1] → address referenced by (SP + 1)
[SP] + 2 → SP
[PC] + 1 → PC

Syntax 11: **PUSH AR3:AR2**

[AR2] → address referenced by SP
[AR3] → address referenced by (SP + 1)
[SP] + 2 → SP
[PC] + 1 → PC

Syntax 12: **PUSH AR5:AR4**

[AR4] → address referenced by SP
[AR5] → address referenced by (SP + 1)
[SP] + 2 → SP
[PC] + 1 → PC

Syntax 13: **PUSH XAR*n***

[XARn(15:0)] → address referenced by SP
[XARn(21:16)] with 10 leading 0s → address referenced by (SP + 1)
[SP] + 2 → SP
[PC] + 1 → PC

Status Bits

OVM, SXM	Neither affects the operation.
C, N, V, Z	None of these are affected by the operation.

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Words

1

Example 1

```
PUSH   @T

; Increment SP by 1 after write.
```

Before instruction		After instruction	
SP	0005	SP	0006
T	0001	T	0001
Data memory		Data memory	
000005	2323	000005	0001

```
C = X       N = X          C unchanged  N unchanged
V = X       Z = X          V unchanged  Z unchanged
```

Example 2

```
PUSH   IFR

; Increment SP by 1 after write.
```

Before instruction		After instruction	
SP	0005	SP	0006
IFR	0101	IFR	0101
Data memory		Data memory	
000005	ffff	000005	0101

```
C = X       N = X          C unchanged  N unchanged
V = X       Z = X          V unchanged  Z unchanged
```

Example 3

```
PUSH   PH:PL

; Write aligned to even address.
; Increment SP by 2 after write.
```

Before instruction		After instruction	
SP	0005	SP	0007
P	4000 2000	P	4000 2000
Data memory		Data memory	
000004	eeee	000004	2000
000005	bbbb	000005	4000

```
C = X       N = X          C unchanged  N unchanged
V = X       Z = X          V unchanged  Z unchanged
```

Example 4

```
PUSH   AR3:AR2

; Increment SP by 2 after write.
```

	Before instruction	After instruction
SP	0004	0006
AR2	7000	7000
AR3	9000	9000

Data memory	Before instruction	After instruction
000004	cccc	7000
000005	aaaa	9000

Before instruction		After instruction	
C = X	N = X	C unchanged	N unchanged
V = X	Z = X	V unchanged	Z unchanged

Syntax

PWRITE ***XAR7,** *loc*

Operands

loc Reference to a 16-bit location, either a data-memory location or a 16-bit register. Use any of these types of operands:

- **@0:***6bit* PAGE0-direct-addressing operand. Data page is 0. Offset is specified by *6bit*, a 6-bit constant from 0 to 63. PAGE0 direct addressing mode is available only when PAGE0 = 1.
- @*6bit* DP-direct-addressing operand. Data page is specified by the DP. Offset is specified by *6bit*, a 6-bit constant from 0 to 63.
- @*reg* Register-addressing operand. For *reg*, specify one of these register names:

AH	AL	PL	PH	T	SP
AR0	AR1	AR2	AR3	AR4	AR5
AR6	AR7				

- ***–SP[***6bit***]** PAGE0-stack-addressing operand. Address in data memory is specified as 0:(SP – *6bit*), where 0: indicates that the six MSBs are 0s and *6bit* is a 6-bit constant from 0 to 63. PAGE0 stack addressing mode is available only when PAGE0 = 0.
- *****ind Indirect-addressing operand. Select one of the following operands (*x* is a number from 0 to 5; *n* is 6 or 7; *z* is a number from 0 to 7; *3bit* is a 3-bit constant):

*SP++	*AR*x*	*XAR*n*	*ARPz
*––SP	*AR*x*++	*XAR*n*++	*
	*––AR*x*	*––XARn	*++
	*+AR*x*[AR0]	*+XARn[AR0]	*––
	*+AR*x*[AR1]	*+XARn[AR1]	*0++
	*+AR*x*[*3bit*]	*+XARn[*3bit*]	*0––
	*AR6%++		

XAR7 Extended auxiliary register XAR7

Opcode

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	1	0	See section 5.7.2 on page 5-23.							

Description

XAR7 holds the address of the program-memory location; *loc* references a data-memory location or register. The value in the data-memory location or the register is loaded to the program-memory location.

Execution

[location addressed by *loc*] → program-memory location addressed by XAR7
[PC] + 1 → PC

Status Bits

OVM, SXM Neither affects the operation.

C, N, V, Z None of these are affected by the operation.

Repeat

This instruction is repeatable. See the description for the RPT instruction.

At the start of the first execution, the value in XAR7 is loaded to the fetch counter (FC). When the PWRITE instruction is repeated, the FC is incremented by 1 during each repetition. Thus, a new program-memory location is loaded during each repetition. To have new data-memory locations read each time, use indirect addressing to reference data memory. Otherwise, the same value will be loaded to program memory during each repetition.

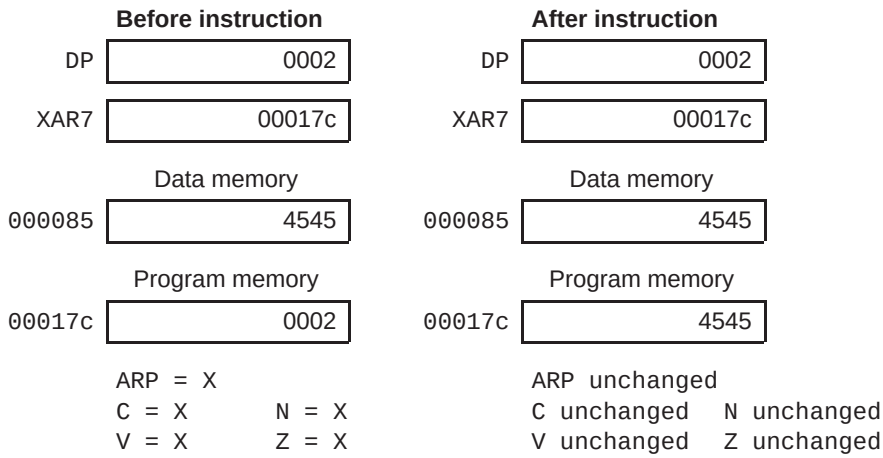
Once the repeat loop begins, it cannot be interrupted by any interrupt.

Words

1

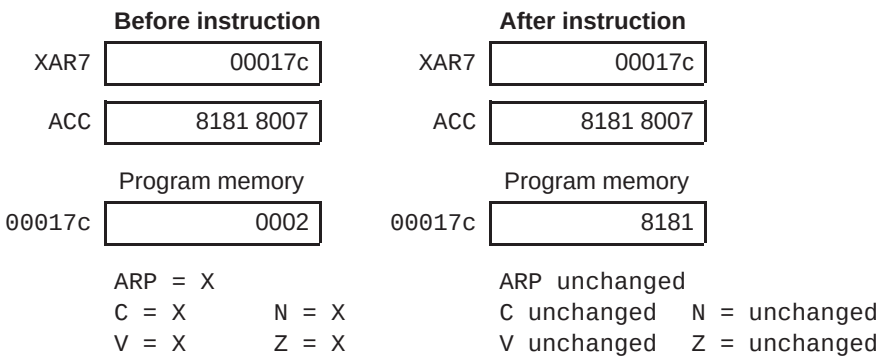
Example 1

```
PWRITE     *XAR7, @5
```



Example 2

```
PWRITE     *XAR7, @AH
```



Example 3

```
RPT #2 || PWRITE  *XAR7, *AR2++
```

Before instruction

AR2	0083
XAR7	00017b

Data memory

000083	7676
000084	6565
000085	2727

Program memory

00017b	0a0a
00017c	0b0b
00017d	0c0c

```
ARP = X
C = X       N = X
V = X       Z = X
```

After instruction

AR2	0086
XAR7	00017b

Data memory

000083	7676
000084	6565
000085	2727

Program memory

00017b	7676
00017c	6565
00017d	2727

```
ARP = 2
C unchanged   N = unchanged
V unchanged   Z = unchanged
```

Syntax **RET**

Operands None

Opcode

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	0	0	1	0	1	0	0

Description The RET instruction is used to return from a function or subroutine. It transfers the return address from the stack to the PC. Bits 21–16 are restored first, followed by bits 15–0.

If you would like interrupts $\overline{\text{INT1}}$–$\overline{\text{INT14}}$, DLOGINT, and RTOSINT enabled upon return, use the RETE instruction instead. If you want the CPU to automatically restore register values before returning from an interrupt service routine, use the IRET instruction.

Execution

[SP] – 1 → SP
[address referenced by SP] → PC(21:16)
[SP] – 1 → SP
[address referenced by SP] → PC(15:0)

Status Bits

OVM, SXM Neither affects the operation.

C, N, V, Z None of these are affected by the operation.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Words 1

Syntax **RETE**

Operands None

Opcode

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	0	0	1	0	0	0	0

Description The RETE instruction is used to return from a function or subroutine. It performs the same function as the RET instruction but also enables interrupts $\overline{\text{INT1}}$–$\overline{\text{INT14}}$, DLOGINT, and RTOSINT. The operation is as follows:

1) Transfer the return address from the stack to the PC. Bits 21–16 are restored first, followed by bits 15–0.

2) Enable the interrupts by clearing INTM.

If you do *not* want the interrupts $\overline{\text{INT1}}$–$\overline{\text{INT14}}$, DLOGINT, and RTOSINT enabled upon return, use the RET instruction instead. If you want the CPU to automatically restore register values before returning from an interrupt service routine, use the IRET instruction.

Execution

[SP] – 1 → SP
[address referenced by SP] → PC(21:16)
[SP] – 1 → SP
[address referenced by SP] → PC(15:0)
0 → INTM

Status Bits

OVM, SXM	Neither affects the operation.
C, N, V, Z	None of these are affected by the operation.
INTM	INTM is cleared.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Words 1

Syntax **ROL ACC**

Operands **ACC** Accumulator

Opcode

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

Description The ROL operation can be seen as the rotation of a 33-bit value that is the concatenation of the carry bit (C) and ACC. This is the operation:

1) Store the value in C to a temporary holding place.
2) Copy bit 31 of ACC to C.
3) Shift the content of ACC left by 1.
4) Transfer the temporarily stored C value to bit 0 of ACC.

Execution

$[C] \rightarrow temp$
$[ACC(31)] \rightarrow C$
$[ACC] << 1 \rightarrow ACC$
$[temp] \rightarrow ACC(0)$
$[PC] + 1 \rightarrow PC$

Status Bits

OVM, SXM	Neither affects the operation.
V, OVC	Neither of these is affected by the operation.
C	The value in C is transferred to bit 0 of ACC. The value in bit 31 of ACC is transferred to C.
N	If bit 31 of the result in ACC is 1, N is set; otherwise, N is cleared.
Z	If the result in ACC is 0, Z is set; otherwise, Z is cleared.

Repeat This instruction is repeatable. See the description for the RPT instruction.

Once the repeat loop begins, it cannot be interrupted by any interrupt.

Words 1

Example

```
ROL     ACC
```

Before instruction

ACC 4000 0000

C = 1 N = X
V = X Z = X

After instruction

ACC 8000 0001

C = 0 N = 1
V unchanged Z = 0

Syntax **ROR ACC**

Operands **ACC** Accumulator

Opcode

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

Description The ROR operation can be seen as the rotation of a 33-bit value that is the concatenation of the carry bit (C) and ACC. This is the operation:

1) Store the value in C to a temporary holding place.
2) Copy bit 0 of ACC to C.
3) Shift the content of ACC right by 1.
4) Transfer the temporarily stored C value to bit 31 of ACC.

Execution

$[C] \rightarrow temp$
$[ACC(0)] \rightarrow C$
$[ACC] >> 1 \rightarrow ACC$
$[temp] \rightarrow ACC(31)$
$[PC] + 1 \rightarrow PC$

Status Bits

OVM, SXM	Neither affects the operation.
V, OVC	Neither of these is affected by the operation.
C	The value in bit 0 of ACC is transferred to C. The value in C is transferred to bit 31 of ACC.
N	If bit 31 of the result in ACC is 1, N is set; otherwise, N is cleared.
Z	If the result in ACC is 0, Z is set; otherwise, Z is cleared.

Repeat This instruction is repeatable. See the description for the RPT instruction.

Once the repeat loop begins, it cannot be interrupted by any interrupt.

Words 1

Example

```
ROR     ACC
```

Before instruction

ACC 8000 0000

```
C = 1       N = X
V = X       Z = X
```

After instruction

ACC c000 0000

```
C = 0       N = 1
V unchanged Z = 0
```

Syntax

1: **RPT** *loc*

2: **RPT** *#8bit*

Operands

8bit 8-bit number from 0 to 255

loc Reference to a 16-bit location, either a data-memory location or a 16-bit register. Use any of these types of operands:

@0:*6bit* PAGE0-direct-addressing operand. Data page is 0. Offset is specified by *6bit*, a 6-bit constant from 0 to 63. PAGE0 direct addressing mode is available only when PAGE0 = 1.

@*6bit* DP-direct-addressing operand. Data page is specified by the DP. Offset is specified by *6bit*, a 6-bit constant from 0 to 63.

@*reg* Register-addressing operand. For *reg*, specify one of these register names:

AH	AL	PL	PH	T	SP
AR0	AR1	AR2	AR3	AR4	AR5
AR6	AR7				

–SP[6bit***]** PAGE0-stack-addressing operand. Address in data memory is specified as 0:(SP – *6bit*), where 0: indicates that the six MSBs are 0s and *6bit* is a 6-bit constant from 0 to 63. PAGE0 stack addressing mode is available only when PAGE0 = 0.

*****ind* Indirect-addressing operand. Select one of the following operands (*x* is a number from 0 to 5; *n* is 6 or 7; *z* is a number from 0 to 7; *3bit* is a 3-bit constant):

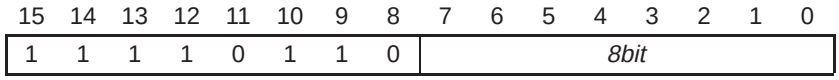
*SP++	*AR*x*	*XAR*n*	*ARPz
*––SP	*AR*x*++	*XAR*n*++	*
	*––AR*x*	*––XARn	*++
	*+AR*x*[AR0]	*+XARn[AR0]	*––
	*+AR*x*[AR1]	*+XARn[AR1]	*0++
	*+AR*x*[*3bit*]	*+XARn[*3bit*]	*0––
	*AR6%++		

Opcode

Syntax 1: **RPT** *loc*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	1	1	See section 5.7.2 on page 5-23.							

Syntax 2: **RPT** *#8bit*



Description

An internal repeat counter (RPTC) is loaded with a value N. The source for N is either the addressed location or an 8-bit unsigned constant. After the instruction following RPT is executed once, it is repeated N times; that is, the instruction following RPT executes N + 1 times. Because the RPTC cannot be saved during a context switch, repeat loops are regarded as multi-cycle instructions and are not interruptible.

The following instructions are repeatable. Only four of the syntaxes of the MOV instruction (those listed here) are repeatable. When a repeatable instruction is repeated, the repeat operation cannot be interrupted by any interrupt.

- **MAC P,** *loc*, **0:***pmem*
- **MOV *(0:***16bit***),** *loc*
- **MOV** *loc*, **#0**
- **MOV** *loc*, *#16bit*
- **MOV** *loc*, ***(0:***16bit***)**
- **NOP** {**ind*}
- **NORM ACC,** *aux***++**
- **NORM ACC,** *aux***– –**
- **PREAD** *loc*, ***XAR7**
- **PWRITE *XAR7,** *loc*
- **ROL ACC**
- **ROR ACC**
- **SUBCU ACC,** *loc*

Instructions that are not repeatable reset the RPTC to 0. The RPTC is also reset to 0 by a device reset.

Execution

Syntax 1: **RPT** *loc*

[addressed location] → RPTC
[PC] + 1 → PC

Syntax 2: **RPT** *#8bit*

8-bit unsigned constant → RPTC
[PC] + 1 → PC

Status Bits

OVM, SXM Neither affects the operation.

C, N, V, Z None of these are affected by the operation.

Repeat

This instruction is not repeatable.

Words

1

Example

```
RPT #1   ||   MOV *(0:0086h), @5
; Copy value at 00 0085h to the next two locations.
```

Before instruction

DP	0002

Data memory

000085	0005
000086	7979
000087	4747

C = X N = X
V = X Z = X

After instruction

DP	0002

Data memory

000085	0005
000086	0005
000087	0005

C unchanged N unchanged
V unchanged Z unchanged

Syntax **SAT ACC**

Operands **ACC** Accumulator

Opcode

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

Description The operation of the SAT instruction depends on the value of the overflow counter (OVC) in status register ST0. When overflow mode is off (OVM in status register ST0 is 0), overflows in ACC do not cause ACC to saturate. Instead, OVC is incremented by 1 for positive overflows and is decremented by 1 for negative overflows. You can use the SAT instruction to saturate ACC such that it reflects the net overflow represented in OVC:

- **OVC = 0 (net overflow is zero).** ACC is not saturated; it is not modified at all. OVC is not modified. C and V are cleared.
- **OVC > 0 (net overflow is positive).** ACC is filled with its greatest positive value, 7FFF FFFF$_{16}$. V is set to 1 to indicate an overflow condition. OVC and C are cleared.
- **OVC < 0 (net overflow is negative).** ACC is filled with its greatest negative value, 8000 0000$_{16}$. V is set to 1 to indicate an overflow condition. OVC and C are cleared.

Execution

If OVC = 0
$0 \rightarrow C$
$0 \rightarrow V$

If OVC > 0
7FFF FFFF$_{16} \rightarrow$ ACC
$0 \rightarrow$ OVC
$0 \rightarrow C$
$1 \rightarrow V$

If OVC < 0
8000 0000$_{16} \rightarrow$ ACC
$0 \rightarrow$ OVC
$0 \rightarrow C$
$1 \rightarrow V$

[PC] + 1 $\rightarrow$ PC

Status Bits

OVM, SXM	Neither affects the operation.
C	C is cleared.
N	If OVC = 0 at the start of the operation, N is affected according to the existing value in ACC. If bit 31 of ACC is 1, N is set; otherwise, N is cleared. If OVC > 0 at the start of the operation, N is cleared when ACC is filled with 7FFF $FFFF_{16}$. If OVC < 0 at the start of the operation, N is set when ACC is filled with 8000 0000_{16}.
OVC	If OVC is not 0 at the start of the operation, OVC is cleared; otherwise, OVC is not affected.
V	If OVC is not 0 at the start of the operation, V is set; otherwise, V is cleared.
Z	If OVC = 0 at the start of the operation, Z is affected according to the existing value in ACC. If ACC = 0, Z is set; otherwise, Z is cleared. If OVC is not 0 at the start of the operation, Z is cleared when ACC is filled with 7FFF $FFFF_{16}$ or 8000 0000_{16}.

Repeat

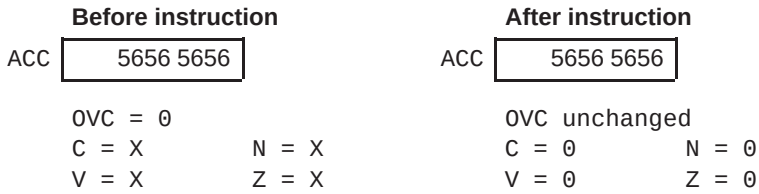
This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Words

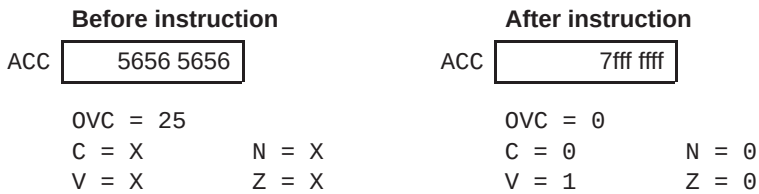
1

Example

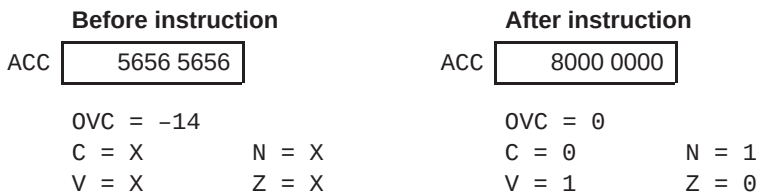
```
SAT     ACC

; For OVC = 0: ACC not changed
```




```
; For OVC > 0: ACC filled with greatest positive value
```




```
; For OVC < 0: ACC filled with greatest negative value
```



Syntax **SB** *8BitOffset*, *cond*

Operands

8BitOffset 8-bit signed offset from –128 to 127

cond Condition to be tested. See Table 6–18.

Opcode

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	See Table 6–18.				*8BitOffset*							

Description

If the specified condition is true, the 8-bit signed offset is added to the current PC value (the address of the SB instruction). This forces program control to the new address, (PC + *8BitOffset*). The 8-bit constant is sign extended to 22 bits before the addition. As shown by the following example, the offset is with respect to the address of the SB instruction.

```
label:      .
            .
            .
            SB   offset,cond   ; <- PC
                               ; offset = label – PC
```

If the specified condition is not true, the PC is incremented by 1 to force program control to the instruction that follows the SB instruction.

Table 6–18 shows the conditions that can be tested.

Table 6–18. Conditions and Their Corresponding Opcode Segments and Flag Tests

cond	Condition	Condition Code in Opcode	Flag Test Performed
NEQ	Not equal to 0	0000	Z = 0
EQ	Equal to 0	0001	Z = 1
GT	Greater than 0	0010	Z = 0 AND N = 0
GEQ	Greater than or equal to 0	0011	N = 0
LT	Less than 0	0100	N = 1
LEQ	Less than or equal to 0	0101	Z = 1 OR N = 1
HI	Higher	0110	C = 1 AND Z = 0
HIS or C	Higher or same or C = 1	0111	C = 1
LO or NC	Lower or C = 0	1000	C = 0
LOS	Lower or same	1001	C = 0 OR Z = 1
NOV	No overflow	1010	V = 0
OV	Overflow	1011	V = 1
NTC	TC = 0	1100	TC = 0
TC	TC = 1	1101	TC = 1
(reserved)	–	1110	–
UNC	Unconditional	1111	None

Execution

If specified condition is true
    [PC] + 8-bit signed offset → PC
else
    [PC] + 1 → PC

Status Bits

OVM, SXM	Neither affects the operation.
C, N, Z	None of these are affected by the operation.
V	If the condition of V is tested (*cond* is OV or NOV), V is cleared.

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Words

1

Example 1

```
SB     –5, GT          ; Test for greater–than condition.
                       ; If N = 0 and Z = 0, branch backward
                       ; 5 addresses.
```

Before instruction		After instruction	
C = X	N = X	C unchanged	N unchanged
V = X	Z = X	V unchanged	Z unchanged

Example 2

```
SB     Continue, NOV   ; Test for no–overflow condition.
                       ; If V = 0, branch to the address
                       ; labeled Continue.
```

Before instruction		After instruction	
C = X	N = X	C unchanged	N unchanged
V = X	Z = X	V = 0	Z unchanged

Syntax **SBBU ACC,** *loc*

Operands

ACC Accumulator

loc Reference to a 16-bit location, either a data-memory location or a 16-bit register. Use any of these types of operands:

@**0:***6bit* PAGE0-direct-addressing operand. Data page is 0. Offset is specified by *6bit*, a 6-bit constant from 0 to 63. PAGE0 direct addressing mode is available only when PAGE0 = 1.

@*6bit* DP-direct-addressing operand. Data page is specified by the DP. Offset is specified by *6bit*, a 6-bit constant from 0 to 63.

@*reg* Register-addressing operand. For *reg*, specify one of these register names:

AH	AL	PL	PH	T	SP
AR0	AR1	AR2	AR3	AR4	AR5
AR6	AR7				

–SP[6bit***]*** PAGE0-stack-addressing operand. Address in data memory is specified as 0:(SP – *6bit*), where 0: indicates that the six MSBs are 0s and *6bit* is a 6-bit constant from 0 to 63. PAGE0 stack addressing mode is available only when PAGE0 = 0.

ind Indirect-addressing operand. Select one of the following operands (*x* is a number from 0 to 5; *n* is 6 or 7; *z* is a number from 0 to 7; *3bit* is a 3-bit constant):

*SP++	*AR*x*	*XAR*n*	*ARPz
*––SP	*AR*x*++	*XAR*n*++	*
	*––AR*x*	*––XARn	*++
	*+AR*x*[AR0]	*+XARn[AR0]	*––
	*+AR*x*[AR1]	*+XARn[AR1]	*0++
	*+AR*x*[*3bit*]	*+XARn[*3bit*]	*0––
	*AR6%++		

Opcode

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	0	1	See section 5.7.2 on page 5-23.							

Description

The logical inversion of the value in the carry bit (C) is added to the content of the addressed location. Then the sum is subtracted from ACC with sign extension suppressed, and the result is placed in ACC. The content of the addressed location is treated as an unsigned number. The carry bit is affected by the subtraction (see the category below called *Status Bits*).

Execution

[ACC] – ([addressed location] + logical inverse of [C]) $\rightarrow$ ACC
[PC] + 1 $\rightarrow$ PC

Status Bits

OVM, OVC — If the operation causes ACC to overflow, ACC and OVC depend on OVM:

- OVM = 0 — ACC is not filled with a saturation value. OVC is affected as follows:

  If the overflow was in the negative direction, OVC is decremented by 1; if the overflow was in the positive direction, OVC is incremented by 1.

- OVM = 1 — If the overflow was in the positive direction, ACC is filled with 7FFF $FFFF_{16}$. If the overflow was in the negative direction, ACC is filled with 8000 0000_{16}.

  OVC is not affected.

SXM — SXM does not affect the operation.

C — If the subtraction generates a borrow, C is cleared; otherwise, C is set.

N — If bit 31 of the result in ACC is 1, N is set; otherwise, N is cleared.

V — If an overflow occurs, V is set; otherwise, V is not affected.

Z — If the result in ACC is 0, Z is set; otherwise, Z is cleared.

Repeat

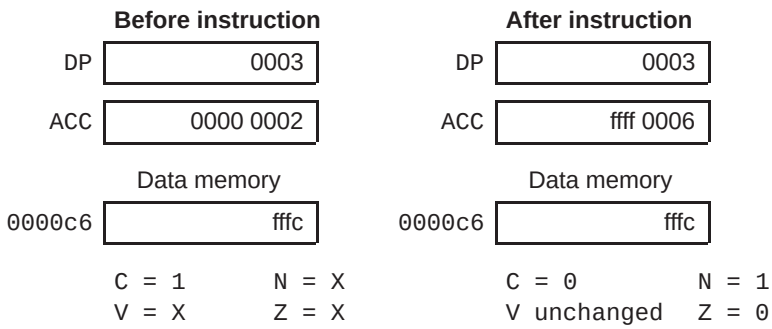
This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Words

1

Example 1

```
SBBU   ACC, @6

; Data value treated as unsigned.
; ACC  = [ACC] – (value at 00 00c6₁₆ + [C̄]) = 2 – (65 532 + 0)
;      = –65 530 = ffff 0006₁₆
```




```
C = 1      N = X          C = 0        N = 1
V = X      Z = X          V unchanged  Z = 0
```

Example 2

```
SBBU   ACC, @AR2

; AR2 value treated as unsigned.
; ACC = [ACC] – ([AR2] + [C̄]) = 8000 0001₁₆ – (2₁₆ + 1₁₆)
```

```
; For OVM = 0: ACC overflows normally. OVC records overflow.
```

```
OVC = 0                   OVC = –1
C = 0      N = X          C = 1        N = 0
V = X      Z = X          V = 1        Z = 0

; For OVM = 1: ACC filled with saturation value
```

```
OVC = X                   OVC unchanged
C = 0      N = X          C = 1        N = 1
V = X      Z = X          V = 1        Z = 0
```

Syntax **SBRK** *#8bit*

Operands *8bit* 8-bit number from 0 to 255

Opcode

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

Description The 8-bit unsigned constant is subtracted from the content of the *current auxiliary register*. The current auxiliary register is defined as the auxiliary register that is pointed to by the auxiliary register pointer (ARP). For example, if ARP = 2, the current auxiliary register is AR2; if ARP = 7, the current auxiliary register is XAR7.

Execution [current auxiliary register] – 8-bit constant → current auxiliary register
[PC] + 1 → PC

Status Bits

ARP	Points to the current auxiliary register.
OVM, SXM	Neither affects the operation.
C, N, V, Z	None of these are affected by the operation.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Words 1

Example

```
SBRK    #5

; ARP = 6 (XAR6 is the current auxiliary register)
```

	Before instruction	**After instruction**
XAR6	00 000a	00 0005

```
ARP = 6                 ARP = 6
C = X      N = X        C unchanged   N unchanged
V = X      Z = X        V unchanged   Z unchanged
```

Syntax

1: **SETC** *BitName1* { , *BitName2* } ... { ,*BitName8* }

2: **SETC** *8BitMask*

Operands

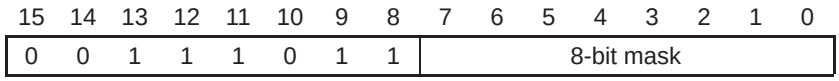
8BitMask 8-bit mask value from 00_{16} to FF_{16}

BitName Choose one of the following names of status bits in status registers ST0 and ST1. More than one of these names can be used at once, separated by commas.

SXM	Sign-extension mode bit (bit 0 of ST0)
OVM	Overflow mode bit (bit 1 of ST0)
TC	Test/control flag bit (bit 2 of ST0)
C	Carry bit (bit 3 of ST0)
INTM	Interrupt global mask bit (bit 0 of ST1)
DBGM	Debug enable mask bit (bit 1 of ST1)
PAGE0	PAGE0 addressing modes configuration bit (bit 2 of ST1)
VMAP	Vector map bit (bit 3 of ST1)

Opcode

If you use syntax 2, the 8-bit mask value you specify is embedded in the 8 low-order bits of the opcode. If you use syntax 1, the CPU derives the corresponding 8-bit mask and then embeds it in the 8 low-order bits of the opcode. Both syntaxes use the following opcode:



Description

The specified status bit or bits are set. You can specify from one to eight of the allowable status bits using syntax 1 or syntax 2. Here is an example of syntax 1 (the order of the operands does not matter):

```
SETC PAGE0, DBGM, INTM, TC, SXM
```

This sets status bits PAGE0, DBGM, INTM, TC, and SXM. To perform the same operation with syntax 2, you would use:

```
SETC 01110101b
```

The b indicates that, in this case, the operand is a binary number. This operand is a mask value that relates to the status bits in this way:

Mask bit position	7	6	5	4	3	2	1	0
Mask value	0	1	1	1	0	1	0	1
Status bit	VMAP	PAGE0	DBGM	INTM	C	TC	OVM	SXM

Where a mask bit is 1, the corresponding status bit is set; where a mask bit is 0, the corresponding bit is not affected. As summarized in Table 6–19:

- Some of the status bits are changed in the execute phase of the pipeline, and others are changed in the decode 2 phase of the pipeline. The SETC INTM instruction disables interrupts at the end of the decode 2 phase. If an interrupt occurs before the decode 2 phase of the SETC INTM instruction, the SETC INTM and the next two instruction words in the pipeline are flushed from the pipeline. On returning from the interrupt, all three of the instructions are fetched again when the PC is restored.
- Any SETC instruction that sets DBGM and/or INTM takes an additional cycle to execute.

Table 6–19. Status Bits as Affected by the SETC Instruction

Mask Bit Position	Status Bit	Value Changed In	Cycles
0	SXM	Execute phase	1
1	OVM	Execute phase	1
2	TC	Execute phase	1
3	C	Execute phase	1
4	INTM	Decode 2 phase	2
5	DBGM	Decode 2 phase	2
6	PAGE0	Decode 2 phase	1
7	VMAP	Decode 2 phase	1

Execution

Set specified status bit(s)
[PC] + 1 → PC

Status Bits

The operation is not affected by OVM or SXM. Any (allowable) status bit that is specified in the instruction will be set. No others are affected.

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Words

1

Example 1

```
SETC   SXM                    ; Set SXM (SXM = 1)
```

Example 2

There are two ways to set VMAP, PAGE0, and OVM with SETC:

```
SETC   VMAP, PAGE0, OVM       ; Bit names used
SETC   #11000010b             ; Binary code used
```

Syntax

1: **SFR ACC,** *shift*

2: **SFR ACC, T**

Operands

ACC Accumulator

shift Number from 1 to 16

T Multiplicand register

Opcode

Syntax 1: **SFR ACC,** *shift*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	0	1	0	0	(*shift* – 1)			

Syntax 2: **SFR ACC, T**

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

Description

The content of ACC is shifted right by an amount specified by *shift* or by the four least significant bits of the T register, T(3:0):

- ***shift.*** Specify a constant from 1 to 16 as the second operand of the instruction. The last bit to be shifted out of ACC is stored in the carry bit (C).

- **T(3:0).** Specify T as the second operand of the instruction. The four least significant bits of T enable a shift from 0 to 15. If the T register specifies a shift of 0, C is cleared; otherwise, C is filled with the last bit to be shifted out of ACC.

During the shift operation, the way the high-order bits are filled depends on the value of the SXM bit in status register ST0. If SXM = 0, the high-order bits are zero-filled (a logical shift is performed); if SXM = 1, the value is sign extended (an arithmetic shift is performed).

To perform a logical right shift on AL or AH, use the LSR instruction. To perform an arithmetic right shift on AL or AH, use the ASR instruction.

Execution

Syntax 1: **SFR** **ACC,** *shift*

[ACC(shift – 1)] → C
If SXM = 0
    [ACC] shifted right (high-order bits zero filled) → ACC
If SXM = 1
    [ACC] shifted right (sign extended) → ACC
[PC] +1 → PC

Syntax 2: **SFR** **ACC, T**

shift = T(3:0)
If shift = 0
    0 → C
    [ACC] → ACC
else
    [ACC(shift – 1)] → C
    If SXM = 0
        [ACC] shifted right (high-order bits zero filled) → ACC
    If SXM = 1
        [ACC] shifted right (sign extended) → ACC
[PC] + 1 → PC

Status Bits

Syntax 1: **SFR** **ACC,** *shift*

OVM	OVM does not affect the operation.
SXM	SXM affects the shift as follows:
	SXM = 0 During the shift, the high-order bits are zero filled.
	SXM = 1 During the shift, the value is sign extended.
C	The last bit to be shifted out of ACC is stored in C.
N	If bit 31 of the result in ACC is 1, N is set; otherwise, N is cleared.
V, OVC	Neither is affected by the operation.
Z	If the result in ACC is 0, Z is set; otherwise, Z is cleared.

Syntax 2: **SFR ACC, T**

OVM	OVM does not affect the operation.
SXM	SXM affects the shift as follows: SXM = 0 During the shift, the high-order bits are zero filled. SXM = 1 During the shift, the value is sign extended.
C	If the T register specifies a shift of 0, C is cleared; otherwise, the last bit to be shifted out of ACC is stored in C.
N	If bit 31 of the result in ACC is 1, N is set; otherwise, N is cleared. Even if the T register specifies a shift of 0, the value of ACC is tested for the negative condition and N is affected.
V, OVC	Neither is affected by the operation.
Z	If the result is in ACC 0, Z is set; otherwise, Z is cleared. Even if the T register specifies a shift of 0, the value of ACC is tested for the zero condition and Z is affected.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Words 1

Example 1

```
SFR    ACC, 8

; For SXM = 0: ACC not sign extended during shifting
```

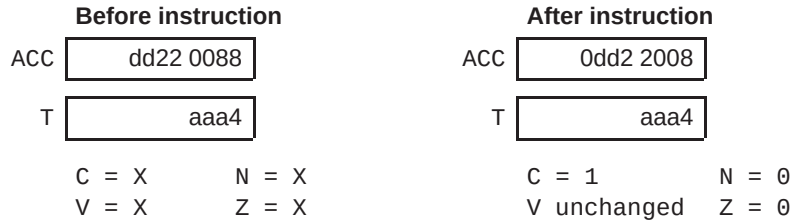
Before instruction		After instruction	
ACC	dd22 0088	ACC	00dd 2200
C = X	N = X	C = 1	N = 0
V = X	Z = X	V unchanged	Z = 0

```
; For SXM = 1: ACC sign extended during shifting
```

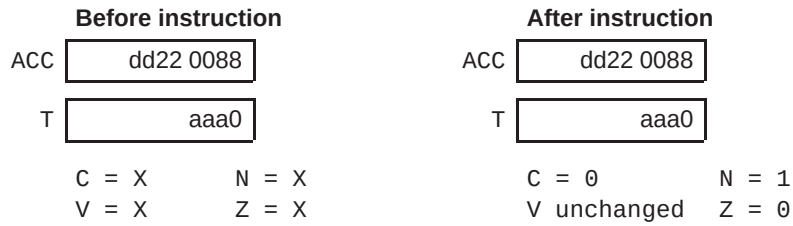
Before instruction		After instruction	
ACC	dd22 0088	ACC	ffdd 2200
C = X	N = X	C = 1	N = 1
V = X	Z = X	V unchanged	Z = 0

Example 2

```
SFR    ACC, T          ; T(3:0) holds shift count
                       ; Sign extension mode is OFF (SXM = 0)

; For T(3:0) > 0
```




```
; For T(3:0) = 0
```



Syntax

SPM *ShiftMode*

Operands

ShiftMode Number from –6 to 1, indicating the desired shift mode. Negative numbers indicate right shifts; positive numbers indicate left shifts.

Opcode

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

The following table shows the relationship between the operand *ShiftMode* and the three LSBs of the opcode. This 3-bit value gets loaded into the PM bits of status register ST0.

ShiftMode	**PM**	**Shift Type**
1	000	Left shift by 1 (default after reset)
0	001	No shift
–1	010	Right shift by 1
–2	011	Right shift by 2
–3	100	Right shift by 3
–4	101	Right shift by 4
–5	110	Right shift by 5
–6	111	Right shift by 6

Description

The three low-order bits of the instruction word are copied into the PM (product shift mode) bits of status register ST0. No shift operation is carried out by the SPM instruction.

The product shift mode determines how the following instructions shift the P register value before using it. For example, the SPM –5 instruction sets PM to 110. This product shift mode forces the CMP instruction to shift the P register value right by 5 bits before comparing it with ACC.

- **ADD ACC, P**
- **CMP ACC, P**
- **MAC P,** *loc*, **0:***pmem*
- **MOV** *loc*, **P**
- **MOVA T,** *loc*
- **MOVH** *loc*, **P**
- **MOVP T,** *loc*
- **MOVS T,** *loc*
- **MPYA P,** *loc*, *#16BitSigned*
- **MPYA P, T,** *loc*
- **MPYS P, T,** *loc*
- **SUB ACC, P**

For all these instructions, the shifting of the P value is done the same way. During a left shift, the high order bits are lost and the low-order bits are zero filled. During a right shift, the high-order bits are filled with copies of the sign bit, and the low-order bits are lost.

Execution

[PM opcode field] → ST0(9:7)

ST0(9:7) = PM bits of status register ST0.

Status Bits

OVM, SXM	Neither affects the operation.
PM	PM is loaded with the 3-bit value specified in the opcode of the SPM instruction.
C, N, V, Z	None of these are affected by the operation.

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Words

1

Example

```
SPM    –3              ; Set product shift mode to PM = 4
                       ; (shift right by 3)
```

Syntax

1: **SUB** **A***X*, *loc*
2: **SUB** *loc*, **A***X*
3: **SUB** **ACC,** *loc* {**<<** *shift3*}
4: **SUB** **ACC,** #*16BitSU* {**<<** *shift2*}
5: **SUB** **ACC, P**

Operands

16BitSU If SXM = 0, *16BitSU* is an unsigned 16-bit number from 0 to 65 535. If SXM = 1, *16BitSU* is a signed 16-bit number from –32 768 to 32 767.

ACC Accumulator

A*X* Use AH (the high word of the accumulator) or AL (the low word of the accumulator).

loc Reference to a 16-bit location, either a data-memory location or a 16-bit register. Use any of these types of operands:

- @**0:***6bit* PAGE0-direct-addressing operand. Data page is 0. Offset is specified by *6bit*, a 6-bit constant from 0 to 63. PAGE0 direct addressing mode is available only when PAGE0 = 1.
- @*6bit* DP-direct-addressing operand. Data page is specified by the DP. Offset is specified by *6bit*, a 6-bit constant from 0 to 63.
- @*reg* Register-addressing operand. For *reg*, specify one of these register names:

AH	AL	PL	PH	T	SP
AR0	AR1	AR2	AR3	AR4	AR5
AR6	AR7				

- ***–SP[***6bit**]** PAGE0-stack-addressing operand. Address in data memory is specified as 0:(SP – *6bit*), where 0: indicates that the six MSBs are 0s and *6bit* is a 6-bit constant from 0 to 63. PAGE0 stack addressing mode is available only when PAGE0 = 0.

***ind** Indirect-addressing operand. Select one of the following operands (*x* is a number from 0 to 5; *n* is 6 or 7; *z* is a number from 0 to 7; *3bit* is a 3-bit constant):

*SP++	*AR*x*	*XAR*n*	*ARPz
*––SP	*AR*x*++	*XAR*n*++	*
	*––AR*x*	*––XARn	*++
	*+AR*x*[AR0]	*+XARn[AR0]	*––
	*+AR*x*[AR1]	*+XARn[AR1]	*0++
	*+AR*x*[*3bit*]	*+XARn[*3bit*]	*0––
	*AR6%++		

P Product register

shift2 Number from 0 to 15

shift3 Number from 0 to 16

Opcode

Syntax 1: **SUB** **A***X*, *loc*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	1	1	1	AX	See section 5.7.2 on page 5-23.							

Note: If AL is used in the instruction, AX = 0; if AH is used, AX = 1.

Syntax 2: **SUB** *loc*, **A***X*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	0	1	0	AX	See section 5.7.2 on page 5-23.							

Note: If AL is used in the instruction, AX = 0; if AH is used, AX = 1.

Syntax 3: **SUB** **ACC,** *loc* {**<<** *shift3*}

For *shift3* < 16

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	*shift3*				See section 5.7.2 on page 5-23.							

For *shift3* = 16

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	See section 5.7.2 on page 5-23.							

Syntax 4: **SUB ACC,** #*16BitSU* {**<<** *shift2*}

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	0	0	0	0	*shift2*			
16BitSU															

Syntax 5: **SUB ACC, P**

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

Note: This is the opcode for the following equivalent instruction: MOVS T, @T.

Description

The general form for the SUB instruction is as follows:

SUB *location1***,** *operand2*

The value referenced or supplied by *operand2* is subtracted from the value at *location1*, and the result is stored to *location1*.

In syntaxes 3 and 4, the SXM bit affects *operand2*. If SXM = 0, *operand2* is treated as unsigned. If SXM = 1, *operand2* is treated as signed. In syntaxes 3 and 4, a left shift can also be specified:

SUB *location1***,** *operand2* **<<** *shift*

The value given by *operand2* is left shifted before being subtracted from the value at *location1*. During the shift, low-order bits are zero filled. If SXM = 1, the value is signed extended; if SXM = 0, high-order bits are zero filled.

Execution

Syntax 1: **SUB A***X***,** *loc*

[AX] – [addressed location] → AX*;*
[PC] +1 → PC

Syntax 2: **SUB** *loc***, A***X*

[addressed location] – AX → addressed location
[PC] + 1 → PC

AX and the addressed value are treated as a signed numbers.
The six most significant bits of XAR6 and XAR7 are not affected by loads to AR6 and AR7, respectively.

Syntax 3: **SUB** **ACC,** *loc* {**<<** *shift3*}

If SXM = 0
    [addressed location] is unsigned
If SXM = 1
    [addressed location] is signed
[ACC] – [addressed location] → ACC
[PC] + 1 → PC

If a shift is specified, the addressed value is shifted before the subtraction. During the shift, low-order bits are zero filled. If SXM = 1, the value is sign extended; if SXM = 0, high-order bits are zero filled.

Syntax 4: **SUB** **ACC,** #*16BitSU* {**<<** *shift2*}

If SXM = 0
    [ACC] – 16-bit unsigned constant → ACC
If SXM = 1
    [ACC] – 16-bit signed constant → ACC
[PC] + 2 → PC

If a shift is specified, the 16-bit constant is shifted before the subtraction. During the shift, low-order bits are zero filled. If SXM = 1, the value is sign extended; if SXM = 0, high-order bits are zero filled.

Syntax 5: **SUB** **ACC, P**

[ACC] – ([P] shifted as per PM bits) → ACC
[PC] + 1 → PC

Status Bits

Syntaxes 1 and 2:

OVM, SXM	Neither affects the operation.
C	If the subtraction generates a borrow, C is cleared; otherwise, C is set.
N	If bit 15 of the result is 1, N is set; otherwise, N is cleared.
V	If an overflow occurs, V is set; otherwise, V is not affected.
Z	If the result is 0, Z is set; otherwise, Z is cleared.

Syntaxes 3, 4, and 5 (ACC is destination):

OVM, OVC — If the operation causes ACC to overflow, ACC and OVC depend on OVM:

- OVM = 0 — ACC is not filled with a saturation value. OVC is affected as follows:

  If the overflow was in the negative direction, OVC is decremented by 1; if the overflow was in the positive direction, OVC is incremented by 1.

- OVM = 1 — If the overflow was in the positive direction, ACC is filled with 7FFF $FFFF_{16}$. If the overflow was in the negative direction, ACC is filled with 8000 0000_{16}.

  OVC is not affected.

SXM — For syntaxes 3 and 4, SXM affects the 16-bit source operand as follows:

- SXM = 0 — The 16-bit source operand is treated as an unsigned number. The value is not sign extended during the operation.
- SXM = 1 — The 16-bit source operand is treated as a signed number. The value is sign extended during the operation.

PM — For syntax 5, the P value is shifted as defined by PM before it is subtracted from ACC. (P itself is not modified.)

C — If the subtraction generates a borrow, C is cleared; otherwise, C is set.
Exception: If a shift of 16 is used, the SUB instruction can clear C but cannot set C.

N — If bit 31 of the result in ACC is 1, N is set; otherwise, N is cleared.

V — If an overflow occurs, V is set; otherwise, V is not affected.

Z — If the result in ACC is 0, Z is set; otherwise, Z is cleared.

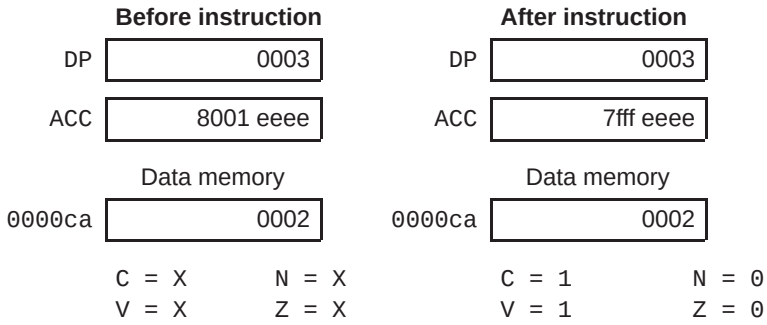
Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Words

Syntaxes1, 2, 3, and 5:	1
Syntax 4:	2

Example 1

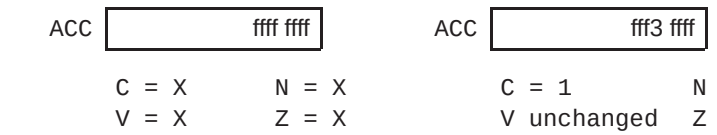
```
SUB    AH, @10

; The subtraction causes an overflow in AH.
```



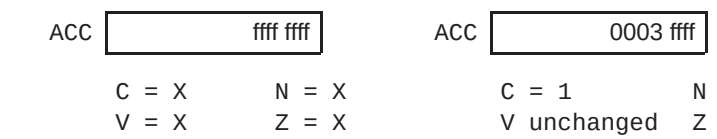
Example 2

```
SUB    ACC, @AR5 << 4
```

```
; For SXM = 0: AR5 value not sign extended during operation
; ACC = [ACC] – ([AR5] << 4) = ffff ffff16 – 000c 000016
```

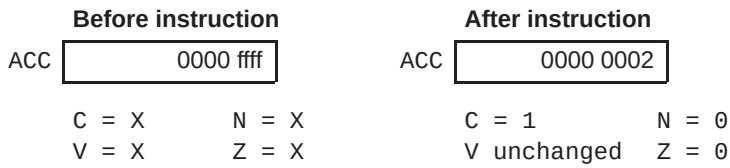



```
; For SXM = 1: AR5 value sign extended during operation
; ACC = [ACC] – ([AR5] << 4) = ffff ffff16 – fffc 000016
```

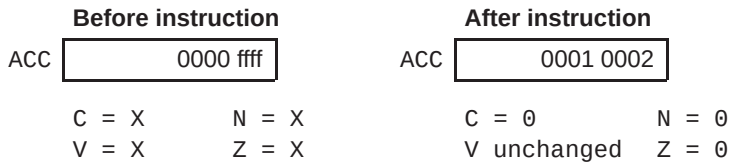


Example 3

```
SUB    ACC, #0fffdh

; For SXM = 0: Constant treated as unsigned
; ACC = [ACC] – 0000 fffd16 =  0000 ffff16 – 0000 fffd16
```



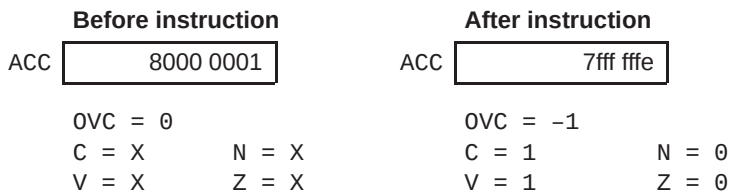

```
; For SXM = 1: Constant treated as signed
; ACC  = [ACC] – ffff fffd16  = 65 535 – (–3)
       = 65 538 = 0001 0002₁₆
```



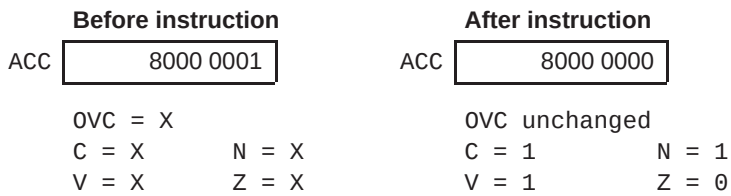
Example 4

```
SUB    ACC, @PL
```

```
; For OVM = 0: ACC overflows normally. OVC records overflow.
```




```
; For OVM = 1: ACC filled with saturation value
```



Syntax

1: **SUBB** **ACC,** #*8bit*

2: **SUBB** *aux*, #*7bit*

3: **SUBB** **SP,** #*7bit*

Operands

7bit 7-bit number from 0 to 127

8bit 8-bit number from 0 to 255

ACC Accumulator

aux Use one of the following auxiliary register names:
AR0 AR1 AR2 AR3 AR4 AR5
XAR6 XAR7

SP Stack pointer

Opcode

Syntax 1: **SUBB** **ACC,** #*8bit*

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

Syntax 2: **SUBB** *aux*, #*7bit*

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

The following table shows the relationship between *aux* and auxNumber:

aux	**auxNumber**	***aux***	**auxNumber**
AR0	0	AR4	4
AR1	1	AR5	5
AR2	2	XAR6	6
AR3	3	XAR7	7

Syntax 3: **SUBB** **SP,** #*7bit*

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

Description

The general form of the SUBB instruction is

SUBB *location1*, *short value*

The short (7- or 8-bit) value is subtracted from the value in *location1*, and the result is stored in *location1*. When a value is subtracted from the SP or one of the auxiliary registers, the subtraction is performed by the address register arithmetic unit (ARAU). When a value is subtracted from ACC, the subtraction is performed by the arithmetic logic unit (ALU).

Execution

Syntax 1: **SUBB** **ACC,** #*8bit*

[ACC] – 8-bit unsigned constant → ACC
[PC] + 1 → PC

Syntax 2: **SUBB** *aux*, #*7bit*

[auxiliary register] – 7-bit unsigned constant → auxiliary register
[PC] + 1 → PC

Syntax 3: **SUBB** **SP,** #*7bit*

[SP] – 7-bit unsigned constant → SP
[PC] + 1 → PC

Status Bits

Syntax 1: **SUBB** **ACC,** #*8bit*

OVM, OVC — If the operation causes ACC to overflow, ACC and OVC depend on OVM:

- OVM = 0 — ACC is not filled with a saturation value. OVC is affected as follows:

  If the overflow was in the negative direction, OVC is decremented by 1; if the overflow was in the positive direction, OVC is incremented by 1.

- OVM = 1 — If the overflow was in the positive direction, ACC is filled with 7FFF FFFF$_{16}$. If the overflow was in the negative direction, ACC is filled with 8000 0000$_{16}$.

  OVC is not affected.

SXM — SXM does not affect the operation.

C — If the subtraction generates a borrow, C is cleared; otherwise, C is set.

N	If bit 31 of the result in ACC is 1, N is set; otherwise, N is cleared.
V	If an overflow occurs, V is set; otherwise, V is not affected.
Z	If the result in ACC is 0, Z is set; otherwise, Z is cleared.

Syntaxes 2 and 3 (auxiliary register or stack pointer holds result):

OVM, SXM	Neither affects the operation.
C, N, V, Z	None of these are affected by the operation.

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Words

1

Example 1

```
SUBB   SP, #2
```

	Before instruction	After instruction
SP	0003	0001

```
C = X      N = X          C unchanged   N unchanged
V = X      Z = X          V unchanged   Z unchanged
```

Example 2

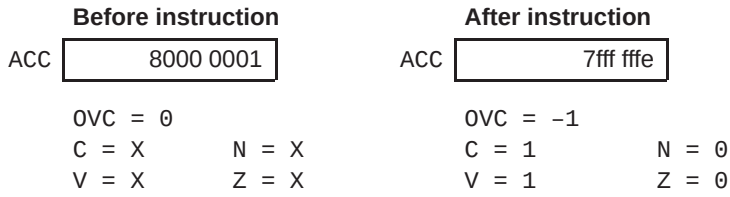
```
SUBB   ACC, #0f7h

; ACC = [ACC] – 247 = –247
```

	Before instruction	After instruction
ACC	0000 0000	ffff ff09

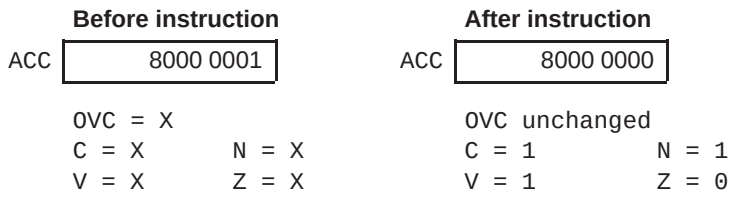
```
C = X      N = X          C = 0         N = 1
V = X      Z = X          V unchanged   Z = 0
```

Example 3

```
SUBB   ACC, #3

; For OVM = 0: ACC overflows normally. OVC records overflow.
```




```
; For OVM = 1: ACC filled with saturation value
```



Syntax **SUBCU ACC,** *loc*

Operands

ACC Accumulator

loc Reference to a 16-bit location, either a data-memory location or a 16-bit register. Use any of these types of operands:

- **@0:***6bit* PAGE0-direct-addressing operand. Data page is 0. Offset is specified by *6bit*, a 6-bit constant from 0 to 63. PAGE0 direct addressing mode is available only when PAGE0 = 1.
- @*6bit* DP-direct-addressing operand. Data page is specified by the DP. Offset is specified by *6bit*, a 6-bit constant from 0 to 63.
- @*reg* Register-addressing operand. For *reg*, specify one of these register names:

AH	AL	PL	PH	T	SP
AR0	AR1	AR2	AR3	AR4	AR5
AR6	AR7				

- ***–SP[***6bit***]*** PAGE0-stack-addressing operand. Address in data memory is specified as 0:(SP – *6bit*), where 0: indicates that the six MSBs are 0s and *6bit* is a 6-bit constant from 0 to 63. PAGE0 stack addressing mode is available only when PAGE0 = 0.
- ***ind*** Indirect-addressing operand. Select one of the following operands (*x* is a number from 0 to 5; *n* is 6 or 7; *z* is a number from 0 to 7; *3bit* is a 3-bit constant):

*SP++	*AR*x*	*XAR*n*	*ARPz
*––SP	*AR*x*++	*XAR*n*++	*
	*––AR*x*	*––XARn	*++
	*+AR*x*[AR0]	*+XARn[AR0]	*––
	*+AR*x*[AR1]	*+XARn[AR1]	*0++
	*+AR*x*[*3bit*]	*+XARn[*3bit*]	*0––
	*AR6%++		

Opcode

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	1	1	See section 5.7.2 on page 5-23.							

Description

The SUBC instruction performs conditional subtraction, which can be used for division as follows:

1) Place a positive 16-bit dividend in AL and clear AH (this can be done with the MOVU instruction).

2) Place a 16-bit positive divisor in data memory or in a register.

3) Execute SUBCU 16 times. After completion of the last SUBCU, the quotient of the division is in AL, and the remainder is in AH.

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 SUBCU is reduced from 16 by that number. One leading 0 is always significant.

SUBCU operations performed as stated above are not affected by the sign-extension mode bit (SXM). For negative accumulator and/or data-memory/register values, SUBCU cannot be used for division.

Execution

ACC – ([addressed location] << 15) → ALU output
Modify C, OVC, and V based on ALU output.
If ALU output ≥ 0
    ([ALU output] << 1) + 1 → ACC
    Set Z and N based on result in ACC.
else
    [ACC] << 1 → ACC
    Set Z and N based on result in ACC.
[PC] + 1 → PC

Status Bits

OVM, SXM	Neither affects the operation.
C	If the subtraction in the first part of the operation generates a borrow, C is cleared; otherwise, C is set.
OVC	If the first part of the operation results in an overflow at the ALU output, OVC is affected as follows: If the overflow was in the negative direction, OVC is decremented by 1; if the overflow was in the positive direction, OVC is incremented by 1.
V	If an overflow occurs in the first part of the operation, V is set; otherwise, V is not affected.
N	If bit 31 of the final result in ACC is 1, N is set; otherwise, N is cleared.
Z	If the final result in ACC is 0, Z is set; otherwise, Z is cleared.

Repeat

This instruction is repeatable. See the description for the RPT instruction.

When repeating SUBCU, do not use a form of indirect addressing that references a new data-memory value during each repetition. The repetition of SUBCU does not perform the division properly unless the same dividend is subtracted from ACC throughout the repeat loop.

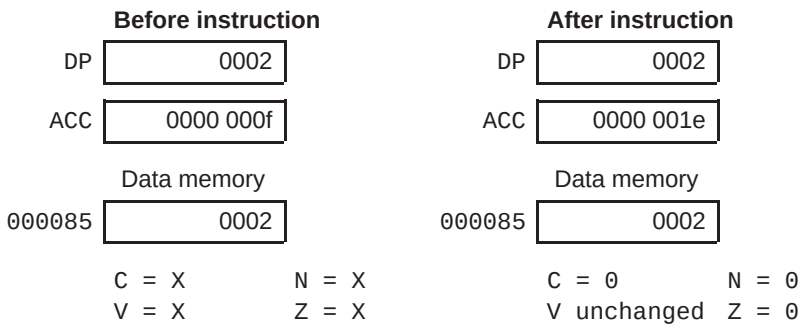
Once the repeat loop begins, it cannot be interrupted by any interrupt.

Words

1

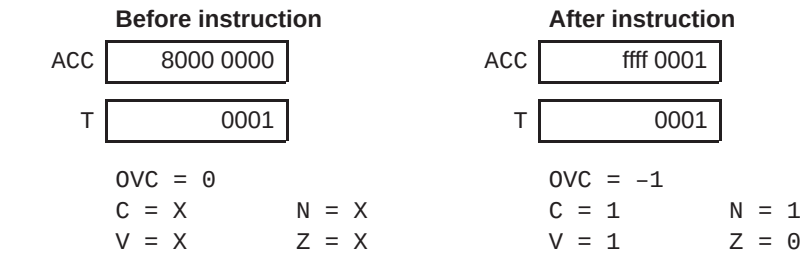
Example 1

```
SUBCU  ACC, @5
```



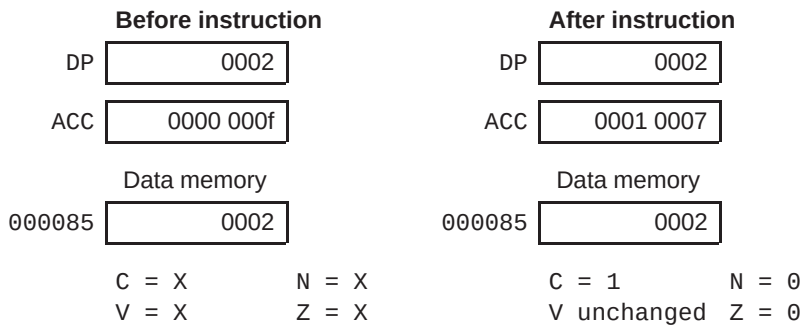
Example 2

```
SUBCU  ACC, @T

; Generates overflow at ALU output. OVC affected.
; ALU output  = [ACC] – ([T] << 15) = 8000 0000₁₆ – 0000 8000₁₆
;             = 7fff 8000₁₆ > 0
; Because ALU output > 0: ACC = (ALU output <<1) + 1
                                = ffff 0000₁₆ + 1
```



Example 3

```
RPT #15 || SUBCU  ACC, @5

; Execute SUBCU 16 times to perform 16–bit division.
; 15 divided by 2 gives 7 and a remainder of 1.
```



Syntax

SUBL ACC, *locLong*

Operands

ACC Accumulator

locLong Reference to a 32-bit data-memory location or one of the 22-bit auxiliary registers (XAR6 or XAR7). Use any of these types of operands:

- **@0:***6bit* PAGE0-direct-addressing operand. Data page is 0. Offset is specified by *6bit*, a 6-bit constant from 0 to 63. PAGE0 direct addressing mode is available only when PAGE0 = 1.
- @*6bit* DP-direct-addressing operand. Data page is specified by the DP. Offset is specified by *6bit*, a 6-bit constant from 0 to 63.
- @**XAR***n* Register-addressing operand. For **XAR***n*, specify XAR6 or XAR7.
- ***–SP[***6bit***]*** PAGE0-stack-addressing operand. Address in data memory is specified as 0:(SP – *6bit*), where 0: indicates that the six MSBs are 0s and *6bit* is a 6-bit constant from 0 to 63. PAGE0 stack addressing mode is available only when PAGE0 = 0.
- ***ind*** Indirect-addressing operand. Select one of the following operands (*x* is a number from 0 to 5; *n* is 6 or 7; *z* is a number from 0 to 7; *3bit* is a 3-bit constant):

*SP++	*AR*x*	*XAR*n*	*ARPz
*––SP	*AR*x*++	*XAR*n*++	*
	*––AR*x*	*––XARn	*++
	*+AR*x*[AR0]	*+XARn[AR0]	*––
	*+AR*x*[AR1]	*+XARn[AR1]	*0++
	*+AR*x*[*3bit*]	*+XARn[*3bit*]	*0––
	*AR6%++		

Opcode

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	1	1	See section 5.7.2 on page 5-23.							

Description

The content of the location addressed by *locLong* is subtracted from the content of ACC, and the result is placed in ACC. The addressed location is either a data-memory location holding a 32-bit value or one of the extended auxiliary registers (XAR6 or XAR7) holding a 22-bit value. To specify XAR6, use the register-addressing operand @XAR6; to specify XAR7, use @XAR7. If @XAR6 or @XAR7 is used, the corresponding auxiliary-register value is not sign

extended; it is treated as an unsigned number. If a register-addressing operand other than @XAR6 or @XAR7 is used, an illegal-instruction trap (TRAP #19 instruction) is generated.

The '27x core expects memory wrappers or peripheral-interface logic to force any 32-bit access, like that of the SUBL instruction, to align to an even address. This alignment is described in section 6.2 on page 6-31.

Execution

If *locLong* is @XAR6 or @XAR7
    [ACC] – unsigned [auxiliary register] → ACC
    [PC] + 1 → PC
else
    [ACC] – [32-bit addressed data-memory location] → ACC
    [PC] + 1 → PC

Status Bits

OVM, OVC	If the operation causes ACC to overflow, ACC and OVC depend on OVM:	
	OVM = 0	ACC is not filled with a saturation value. OVC is affected as follows: If the overflow was in the negative direction, OVC is decremented by 1; if the overflow was in the positive direction, OVC is incremented by 1.
	OVM = 1	If the overflow was in the positive direction, ACC is filled with 7FFF $FFFF_{16}$. If the overflow was in the negative direction, ACC is filled with 8000 0000_{16}. OVC is not affected.
SXM	SXM does not affect the operation.	
C	If the subtraction generates a borrow, C is cleared; otherwise, C is set.	
N	If bit 31 of the result in ACC is 1, N is set; otherwise, N is cleared.	
V	If an overflow occurs, V is set; otherwise, V is not affected.	
Z	If the result in ACC is 0, Z is set; otherwise, Z is cleared.	

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Words

1

Example 1

```
SUBL   ACC, @XAR6

; XAR6 treated as unsigned
; ACC = [ACC] – [XAR6] = 4 – 65 534 = –65 530 = ffff 0006₁₆
```

	Before instruction	After instruction
ACC	0000 0004	ffff 0006
XAR6	00 fffe	00 fffe

Before instruction		After instruction	
C = X	N = X	C = 0	N = 1
V = X	Z = X	V unchanged	Z = 0

Example 2

```
SUBL   ACC, @10

; 32-bit data value read from two consecutive 16-bit locations
; and treated as signed
; ACC = [ACC] – ffff fffc₁₆ = 4 – (–4)
```

	Before instruction	After instruction
DP	0003	0003
ACC	0000 0004	0000 0008

Data memory	Before instruction	Data memory	After instruction
0000ca	fffc	0000ca	fffc
0000cb	ffff	0000cb	ffff

Before instruction		After instruction	
C = X	N = X	C = 0	N = 0
V = X	Z = X	V unchanged	Z = 0

Example 3

```
SUBL   ACC, @7

; Addressed 32–bit value treated as signed
; ACC = [ACC] – 32-bit value = 8000 0000₁₆ – 0000 0002₁₆
```

Before instruction		After instruction	
DP	0003	DP	0003

Data memory		Data memory	
0000c6	0002	0000c6	0002
0000c7	0000	0000c7	0000

```
; For OVM = 0: ACC overflows normally. OVC records overflow.
```

Before instruction		After instruction	
ACC	8000 0000	ACC	7fff fffe

```
OVC = 0                    OVC = –1
C = X      N = X           C = 1        N = 0
V = X      Z = X           V = 1        Z = 0
```

```
; For OVM = 1: ACC filled with saturation value
```

Before instruction		After instruction	
ACC	8000 0000	ACC	8000 0000

```
OVC = X                    OVC unchanged
C = X      N = X           C = 1        N = 1
V = X      Z = X           V = 1        Z = 0
```

Syntax

SUBU ACC, *loc*

Operands

ACC Accumulator

loc Reference to a 16-bit location, either a data-memory location or a 16-bit register. Use any of these types of operands:

@0:*6bit* PAGE0-direct-addressing operand. Data page is 0. Offset is specified by *6bit*, a 6-bit constant from 0 to 63. PAGE0 direct addressing mode is available only when PAGE0 = 1.

@*6bit* DP-direct-addressing operand. Data page is specified by the DP. Offset is specified by *6bit*, a 6-bit constant from 0 to 63.

@*reg* Register-addressing operand. For *reg*, specify one of these register names:

AH	AL	PL	PH	T	SP
AR0	AR1	AR2	AR3	AR4	AR5
AR6	AR7				

–SP[6bit***]** PAGE0-stack-addressing operand. Address in data memory is specified as 0:(SP – *6bit*), where 0: indicates that the six MSBs are 0s and *6bit* is a 6-bit constant from 0 to 63. PAGE0 stack addressing mode is available only when PAGE0 = 0.

*****ind Indirect-addressing operand. Select one of the following operands (*x* is a number from 0 to 5; *n* is 6 or 7; *z* is a number from 0 to 7; *3bit* is a 3-bit constant):

*SP++	*AR*x*	*XAR*n*	*ARPz
*––SP	*AR*x*++	*XAR*n*++	*
	*––AR*x*	*––XARn	*++
	*+AR*x*[AR0]	*+XARn[AR0]	*––
	*+AR*x*[AR1]	*+XARn[AR1]	*0++
	*+AR*x*[*3bit*]	*+XARn[*3bit*]	*0––
	*AR6%++		

Opcode

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	1	See section 5.7.2 on page 5-23.							

Description

The value addressed by *loc* is subtracted from the content of ACC, and the result is placed in ACC. Sign extension is suppressed. The addressed value is treated as an unsigned number.

Execution

[ACC] – unsigned [addressed location] → ACC
[PC] + 1 → PC

Status Bits

OVM, OVC — If the operation causes ACC to overflow, ACC and OVC depend on OVM:

- OVM = 0 — ACC is not filled with a saturation value. OVC is affected as follows:

  If the overflow was in the negative direction, OVC is decremented by 1; if the overflow was in the positive direction, OVC is incremented by 1.

- OVM = 1 — If the overflow was in the positive direction, ACC is filled with 7FFF $FFFF_{16}$. If the overflow was in the negative direction, ACC is filled with 8000 0000_{16}.

  OVC is not affected.

SXM — SXM does not affect the operation.

C — If the subtraction generates a borrow, C is cleared; otherwise, C is set.

N — If bit 31 of the result in ACC is 1, N is set; otherwise, N is cleared.

V — If an overflow occurs, V is set; otherwise, V is not affected.

Z — If the result in ACC is 0, Z is set; otherwise, Z is cleared.

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Words

1

Example 1

```
SUBU   ACC, @AR6
```

; ACC = [ACC] – [AR6] = 4 – 65 534 = –65 530 = ffff 0006_{16}

	Before instruction	After instruction
ACC	0000 0004	ffff 0006
XAR6	3a fffe	3a fffe
	C = X N = X	C = 0 N = 1
	V = X Z = X	V unchanged Z = 0

Example 2

```
SUBU   ACC, @10
```

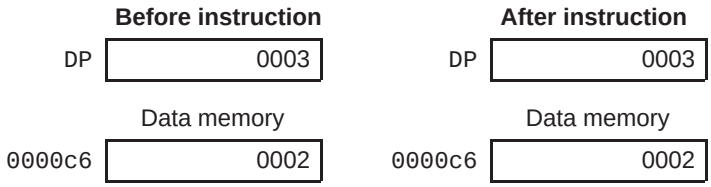
; ACC = [ACC] – $fffc_{16}$ = 1 – 65 532 = –65 531 = ffff 0005_{16}

	Before instruction	After instruction
DP	0003	0003
ACC.	0000 0001	ffff 0005
	Data memory	Data memory
0000ca	fffc	fffc
	C = X N = X	C = 0 N = 1
	V = X Z = X	V unchanged Z = 0

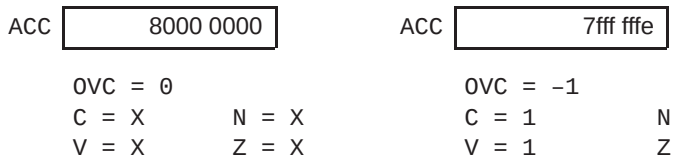
Example 3

```
SUBU   ACC, @6
```

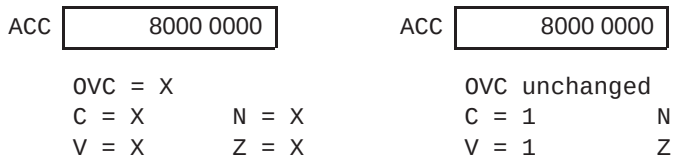
; ACC = [ACC] – value at 00 $00c6_{16}$ = 8000 0000_{16} – 0000 0002_{16}




```
; For OVM = 0: ACC overflows normally. OVC records overflow.
```




```
; For OVM = 1: ACC filled with saturation value
```



Syntax **SXTB** **A***X*

Operands **A***X* Use AH (the high word of the accumulator) or AL (the low word of the accumulator).

Opcode

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

Note: If AL is used in the instruction, AX = 0; if AH is used, AX = 1.

Description The least significant byte of the specified accumulator half (AH or AL) is sign extended into the most significant byte. For example, suppose you execute SXTB AH. If bit 7 of AH is 0, bits 15 through 8 are replaced with 0s; if bit 7 is 1, bits 15 through 8 are replaced with 1s.

Execution

If AX(7) = 0
    AX AND $00FF_{16} \rightarrow$ AX
else
    AX OR $FF00_{16} \rightarrow$ AX
[PC] + 1 $\rightarrow$ PC

Status Bits

OVM, SXM	Neither affects the operation.
C, V	Neither of these is affected by the operation.
N	If bit 15 of the result in AL/AH is 1, N is set; otherwise, N is cleared.
Z	If the result in AL/AH is 0, Z is set; otherwise, Z is cleared.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Words 1

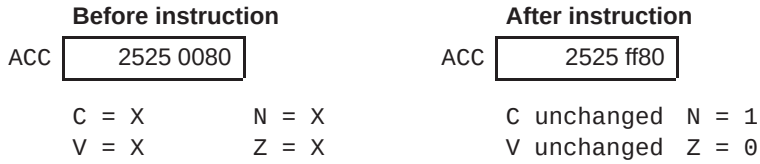
Example 1

```
SXTB   AH   ; Sign extend the LSByte of AH into the MSByte.
```

Before instruction		After instruction	
ACC	ff30 2525	ACC	0030 2525
C = X	N = X	C unchanged	N = 0
V = X	Z = X	V unchanged	Z = 0

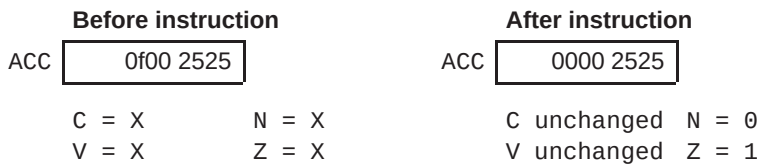
Example 2

```
SXTB   AL   ; Sign extend the LSByte of AL into the MSByte
```



Example 3

```
SXTB   AH   ; Sign extend the LSByte of AH into the MSByte.
```



Syntax **TBIT** *loc*, *#BitNumber*

Operands

BitNumber Number from 0 to 15

loc Reference to a 16-bit location, either a data-memory location or a 16-bit register. Use any of these types of operands:

@0:*6bit* PAGE0-direct-addressing operand. Data page is 0. Offset is specified by *6bit*, a 6-bit constant from 0 to 63. PAGE0 direct addressing mode is available only when PAGE0 = 1.

@*6bit* DP-direct-addressing operand. Data page is specified by the DP. Offset is specified by *6bit*, a 6-bit constant from 0 to 63.

@*reg* Register-addressing operand. For *reg*, specify one of these register names:

AH	AL	PL	PH	T	SP
AR0	AR1	AR2	AR3	AR4	AR5
AR6	AR7				

–SP[6bit***]** PAGE0-stack-addressing operand. Address in data memory is specified as 0:(SP – *6bit*), where 0: indicates that the six MSBs are 0s and *6bit* is a 6-bit constant from 0 to 63. PAGE0 stack addressing mode is available only when PAGE0 = 0.

*****_ind_ Indirect-addressing operand. Select one of the following operands (*x* is a number from 0 to 5; *n* is 6 or 7; *z* is a number from 0 to 7; *3bit* is a 3-bit constant):

*SP++	*AR*x*	*XAR*n*	*ARPz
*––SP	*AR*x*++	*XAR*n*++	*
	*––AR*x*	*––XARn	*++
	*+AR*x*[AR0]	*+XARn[AR0]	*––
	*+AR*x*[AR1]	*+XARn[AR1]	*0++
	*+AR*x*[*3bit*]	*+XARn[*3bit*]	*0––
	*AR6%++		

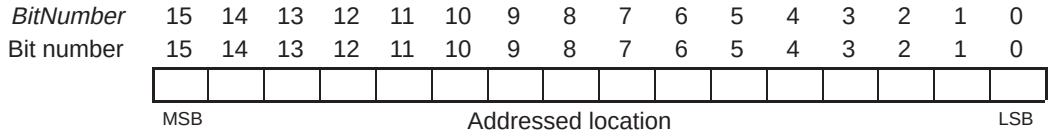
Opcode

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	*BitNumber*				See section 5.7.2 on page 5-23.							

Description

The TBIT instruction tests a specified bit in a data-memory location or in a register and then modifies the TC bit in status register ST0. If the tested bit is 0, TC is cleared. If the tested bit is 1, TC is set.

As the following figure shows, the value you specify for *BitNumber* directly corresponds to the actual bit number. For example, if *BitNumber* = 0, you access bit 0 (the least significant bit) of the addressed location; if *BitNumber* = 15, you access bit 15 (the most significant bit).



Execution

If specified bit of [addressed location] is 0
    TC = 0
else
    TC = 1
[PC] + 1 → PC

Status Bits

OVM, SXM	Neither affects the operation.
C, N, V, Z	None of these are affected by the operation.
TC	If the tested bit is 1, TC is set; if the tested bit is 0 , TC is cleared.

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Words

1

Example 1

```
TBIT   @T, #15        ; Test bit 15 of the T register.
```

Before instruction

T 8000

```
TC = X
C = X       N = X
V = X       Z = X
```

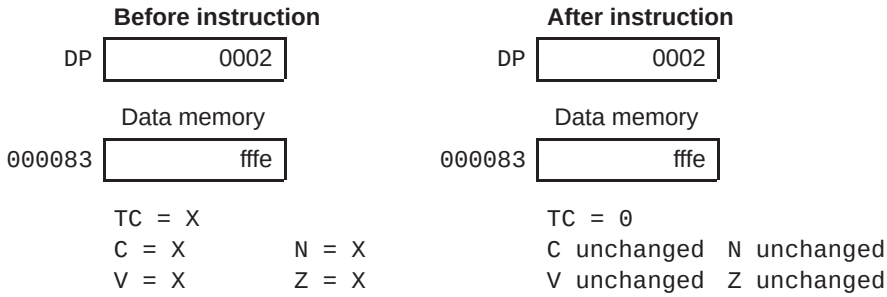
After instruction

T 8000

```
TC = 1
C unchanged  N unchanged
V unchanged  Z unchanged
```

Example 2

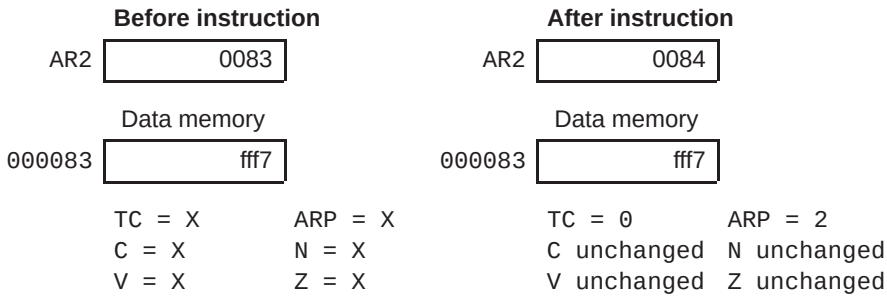
```
TBIT   @3, #0          ; Test bit 0 of addressed location.
```




```
TC = X                          TC = 0
C = X        N = X              C unchanged  N unchanged
V = X        Z = X              V unchanged  Z unchanged
```

Example 3

```
TBIT   *AR2++, #3      ; Test bit 3 of addressed location.
```




```
TC = X       ARP = X            TC = 0       ARP = 2
C = X        N = X              C unchanged  N unchanged
V = X        Z = X              V unchanged  Z unchanged
```

Syntax **TEST ACC**

Operands **ACC** Accumulator

Opcode

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

Description ACC is compared to zero (ACC – 0) and the Z and N flags in status register ST0 are modified accordingly. You determine the accumulator's relationship to zero by testing these flags. Here are some of the possible conditions: If Z = 1, ACC is 0. If N = 1, ACC is less than zero. If Z = 0 and N = 0, ACC is greater than zero.

One use for the TEST instruction is to modify Z and N prior to a conditional branch instruction (B or SB) that is dependent on Z and/or N.

Execution

[ACC] – 0 → ALU output
Modify N and Z according to result.
[PC] + 1 → PC

Status Bits

OVM, SXM	Neither affects the operation.
C, OVC, V	None of these are affected by the operation.
N	If bit 31 of the result in ACC is 1, N is set; otherwise, N is cleared.
Z	If the result in ACC is 0, Z is set; otherwise, Z is cleared.

Repeat This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Words 1

Example 1

```
TEST   ACC          ; ACC = 0
```

Before instruction		After instruction	
ACC	0000 0000	ACC	0000 0000
C = X	N = X	C unchanged	N = 0
V = X	Z = X	V unchanged	Z = 1

Example 2

```
TEST   ACC          ; ACC < 0
```

Before instruction

```
C = X        N = X
V = X        Z = X
```

After instruction

```
C unchanged  N = 1
V unchanged  Z = 0
```

Syntax **TRAP** #*VectorNumber*

Operands The following table indicates which interrupt vector is associated with a chosen value for *VectorNumber*.

VectorNumber	**Interrupt Vector**	***VectorNumber***	**Interrupt Vector**
0	RESET	16	RTOSINT
1	INT1	17	Reserved
2	INT2	18	NMI
3	INT3	19	ILLEGAL
4	INT4	20	USER1
5	INT5	21	USER2
6	INT6	22	USER3
7	INT7	23	USER4
8	INT8	24	USER5
9	INT9	25	USER6
10	INT10	26	USER7
11	INT11	27	USER8
12	INT12	28	USER9
13	INT13	29	USER10
14	INT14	30	USER11
15	DLOGINT	31	USER12

Opcode

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

Description

The TRAP instruction transfers program control to the interrupt service routine that corresponds to the vector specified in the instruction. It does not affect the interrupt flag register (IFR) or the interrupt enable register (IER), regardless of whether the chosen interrupt has corresponding bits in these registers. The TRAP instruction is not affected by the interrupt global mask bit (INTM) in status register ST1. It is also not affected by enable bits in the IER or the debug interrupt enable register (DBGIER). Once the TRAP instruction reaches the decode 2 phase of the pipeline, hardware interrupts cannot be serviced until the TRAP instruction is done executing (until the interrupt service routine begins).

Part of the operation involves saving pairs of 16-bit CPU registers. Each pair of registers is saved in a single 32-bit operation. The register forming the low word of the pair is saved first (to an even address); the register forming the high word of the pair is saved next (to the following odd address). For example, the first value saved is the concatenation of the T register and status register ST0 (T:ST0). ST0 is saved first, then T.

Note:

The TRAP #0 instruction does not initiate a full reset. It only forces execution of the interrupt service routine that corresponds to the RESET interrupt vector.

Execution

Empty the pipeline.
Increment and temporarily store PC:
    [PC] + 1 → PC
    [PC] → temp
Fetch specified vector.
Increment stack pointer:
    [SP] + 1 → SP
Save register values:
    [T:ST0] → addresses referenced by SP
    [SP] + 2 → SP
    [AH:AL] → addresses referenced by SP
    [SP] + 2 → SP
    [PH:PL] → addresses referenced by SP
    [SP] + 2 → SP
    [AR1:AR0] → addresses referenced by SP
    [SP] + 2 → SP
    [DP:ST1] → addresses referenced by SP
    [SP] + 2 → SP
    [DBGSTAT:IER] → addresses referenced by SP
    [SP] + 2 → SP

Disable interrupts $\overline{\text{INT1}}$–$\overline{\text{INT14}}$, DLOGINT, and RTOSINT:
    1 → INTM
Disable debug events:
    1 → DBGM
Disable access to emulation registers:
    0 → EALLOW
Clear LOOP and IDLESTAT flags:
    0 → LOOP
    0 → IDLESTAT
Load PC with fetched vector.
    vector → PC

Status Bits

OVM, SXM	Neither affects the operation.
C, N, V, Z	None of these are affected by the operation.
DBGM, INTM	Both are set by the operation. This disables debug events (DBGM = 1) and disables interrupts $\overline{\text{INT1}}$–$\overline{\text{INT14}}$, DLOGINT, and RTOSINT (INTM = 1).
EALLOW, LOOP, IDLESTAT	All three are cleared by the operation. Clearing EALLOW disables access to emulation registers.

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Words

1

Example

```
TRAP   #3   ; Execute service routine for INT3.
            ; IER and IFR not affected.
```

Before instruction

IER	0004
IFR	0004
SP	00efff

Data memory

00efff	0000
00f000	0000
00f001	0000
00f002	0000
00f003	0000
00f004	0000
00f005	0000
00f006	0000
00f007	0000
00f008	0000
00f009	0000
00f00a	0000
00f00b	0000
00f00c	0000
00f00d	0000
00f00e	0000

```
INTM = X   DBGM = X
LOOP = X
EALLOW = X
IDLESTAT = X
```

After instruction

IER	0004
IFR	0004
SP	f00e

Data memory

00efff	0000	
00f000	[ST0]	
00f001	[T]	
00f002	[AL]	
00f003	[AH]	
00f004	[PL]	
00f005	[PH]	
00f006	[AR0]	
00f007	[AR1]	
00f008	[ST1]	
00f009	[DP]	
00f00a	[IER]	
00f00b	[DBGSTAT]	
00f00c	return address	(low half)
00f00d	return address	(high half)
00f00e	0000	

```
INTM = 1     DBGM = 1
LOOP = 0
EALLOW = 0
IDLESTAT = 0
```

Syntax

1: **XOR** **A***X*, *loc*
2: **XOR** *loc*, **A***X*
3: **XOR** *loc*, #*16BitMask*

Operands

16BitMask 16-bit mask value from 0000_{16} to $FFFF_{16}$

A*X* Use AH (the high word of the accumulator) or AL (the low word of the accumulator).

loc Reference to a 16-bit location, either a data-memory location or a 16-bit register. Use any of these types of operands:

@0:*6bit* PAGE0-direct-addressing operand. Data page is 0. Offset is specified by *6bit*, a 6-bit constant from 0 to 63. PAGE0 direct addressing mode is available only when PAGE0 = 1.

@6bit DP-direct-addressing operand. Data page is specified by the DP. Offset is specified by *6bit*, a 6-bit constant from 0 to 63.

@reg Register-addressing operand. For *reg*, specify one of these register names:

AH	AL	PL	PH	T	SP
AR0	AR1	AR2	AR3	AR4	AR5
AR6	AR7				

–SP[6bit***]** PAGE0-stack-addressing operand. Address in data memory is specified as 0:(SP – *6bit*), where 0: indicates that the six MSBs are 0s and *6bit* is a 6-bit constant from 0 to 63. PAGE0 stack addressing mode is available only when PAGE0 = 0.

*****ind* Indirect-addressing operand. Select one of the following operands (*x* is a number from 0 to 5; *n* is 6 or 7; *z* is a number from 0 to 7; *3bit* is a 3-bit constant):

*SP++	*AR*x*	*XAR*n*	*ARPz
*––SP	*AR*x*++	*XAR*n*++	*
	*––AR*x*	*––XARn	*++
	*+AR*x*[AR0]	*+XARn[AR0]	*––
	*+AR*x*[AR1]	*+XARn[AR1]	*0++
	*+AR*x*[*3bit*]	*+XARn[*3bit*]	*0––
	*AR6%++		

Opcode

Syntax 1: **XOR** **A***X*, *loc*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	0	0	0	AX	See section 5.7.2 on page 5-23.							

Note: If AL is used in the instruction, AX = 0; if AH is used, AX = 1.

Syntax 2: **XOR** *loc*, **A***X*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	AX	See section 5.7.2 on page 5-23.							

Note: If AL is used in the instruction, AX = 0; if AH is used, AX = 1.

Syntax 3: **XOR** *loc*, *#16BitMask*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	0	0	See section 5.7.2 on page 5-23.							
16BitMask															

Description

The general form for the XOR instruction is as follows:

XOR *operand1*, *operand2*

The ALU calculates the exclusive OR of the values referenced by *operand1* and *operand2*. The result is stored to the location of *operand1*.

Execution

Syntax 1: **XOR** **A***X*, *loc*

[AX] XOR [addressed location] → AX
[PC] + 1 → PC

Syntax 2: **XOR** *loc*, **A***X*

[AX] XOR [addressed location] → addressed location
[PC] + 1 → PC

Syntax 3: **XOR** *loc*, *#16BitMask*

[addressed location] XOR 16-bit mask value → addressed location
[PC] + 2 → PC

Status Bits

OVM, SXM	Neither affects the operation.
C, V	Neither is affected by the operation.
N	If bit 15 of the result is 1, N is set; otherwise, N is cleared.
Z	If the result is 0, Z is set; otherwise, Z is cleared.

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

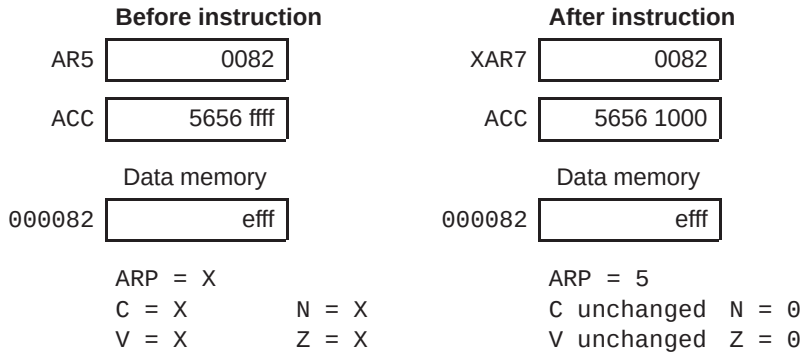
Words

Syntaxes 1 and 2: 1

Syntax 3: 2

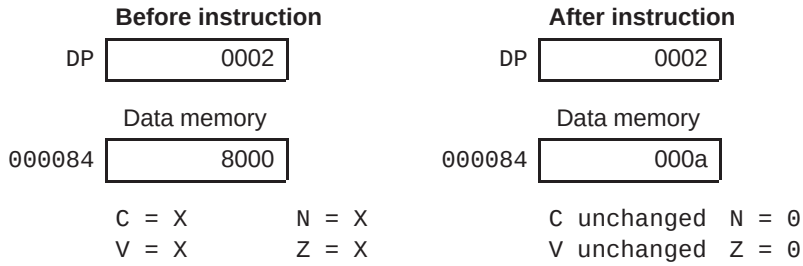
Example 1

```
XOR    AL, *AR5
```



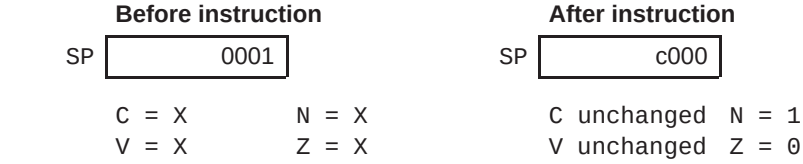
Example 2

```
XOR    @4, #1000000000001010b     ; Binary mask used.
```



Example 3

```
XOR    SP, #0c001h                ; Hexadecimal mask used.
```



Syntax

XORB A*X***, #***8BitMask*

Operands

8BitMask 8-bit mask value from 00_{16} to FF_{16}

A*X* Use AH (the high word of the accumulator) or AL (the low word of the accumulator).

Opcode

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

Note: If AL is used in the instruction, AX = 0; if AH is used, AX = 1.

Description

The ALU calculates the exclusive OR of the specified accumulator half (AH or AL) and the 8-bit mask value. The result is stored to the specified accumulator half.

Execution

[AX] XOR (8-bit mask value AND $00FF_{16}$) $\rightarrow$ AX
[PC] + 1 $\rightarrow$ PC

Status Bits

OVM, SXM	Neither affects the operation.
C, V	Neither is affected by the operation.
N	If bit 15 of the result in AL/AH is 1, N is set; otherwise, N is cleared.
Z	If the result in AL/AH is 0, Z is set; otherwise, Z is cleared.

Repeat

This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Words

1

Example

```
XORB   AL, #0c0h                    ; Hexadecimal mask used.
```

Before instruction

ACC | eeee 00c1 |

```
C = X        N = X
V = X        Z = X
```

After instruction

ACC | eeee 0001 |

```
C unchanged  N = 0
V unchanged  Z = 0
```

Chapter 7

Emulation Features

The core in the ’27x contains hardware extensions for advanced emulation features that can assist you in the development of your application system (software and hardware). This chapter describes the emulation features that are available on all ’27x devices using only the JTAG port (with TI extensions).

For more information about instructions shown in examples in this chapter, see Chapter 6, *Assembly Language Instructions*.

Topic **Page**

7.1 Overview of Emulation Features 7-2
7.2 Debug Interface 7-3
7.3 Debug Terminology 7-6
7.4 Execution Control Modes 7-7
7.5 Aborting Interrupts With the ABORTI Instruction 7-14
7.6 DT-DMA Mechanism 7-15
7.7 Analysis Breakpoints, Watchpoints, and Counter(s) 7-18
7.8 Data Logging 7-21
7.9 Sharing Analysis Resources 7-28
7.10 Diagnostics and Recovery 7-29

7.1 Overview of Emulation Features

The core's hardware extensions for advanced emulation features provide simple, inexpensive, and speed-independent access to the core for sophisticated debugging and economical system development, without requiring the costly cabling and access to processor pins required by traditional emulator systems. It provides this access without intruding on system resources.

The on-chip development interface provides:

- ❏ Minimally intrusive access to internal and external memory
- ❏ Minimally intrusive access to CPU and peripheral registers
- ❏ Control of the execution of background code while continuing to service time-critical interrupts
  - ■ Break on a software breakpoint instruction (instruction replacement)
  - ■ Break on a specified program or data access without requiring instruction replacement (accomplished using bus comparators)
  - ■ Break on external attention request from debug host or additional hardware
  - ■ Break after the execution of a single instruction (single-stepping)
  - ■ Control over the execution of code from device power up
- ❏ Nonintrusive determination of device status
  - ■ Detection of a system reset, emulation/test-logic reset, or power-down occurrence
  - ■ Detection of the absence of a system clock or memory-ready signal
  - ■ Determination of whether global interrupts are enabled
  - ■ Determination of why debug accesses might be blocked
- ❏ Rapid transfer of memory contents between the device and a host (data logging)
- ❏ A cycle counter for performance benchmarking. With a 100-MHz cycle clock, the counter can benchmark actions up to 3 hours in duration.

7.2 Debug Interface

The target-level TI debug interface uses the five standard IEEE 1149.1 (JTAG) signals ($\overline{\text{TRST}}$, TCK, TMS, TDI, and TDO) and the two TI extensions (EMU0 and EMU1). Figure 7–1 shows the 14-pin JTAG header that is used to interface the target to a scan controller, and Table 7–1 (page 7-4) defines the pins.

As shown in the figure, the header requires more than the five JTAG signals and the TI extensions. It also requires a test clock return signal (TCK_RET), the target supply (V_{CC}) and ground (GND). TCK_RET is a test clock out of the scan controller and into the target system. The target system uses TCK_RET if it does not supply its own test clock (in which case TCK would simply not be used). In many target systems, TCK_RET is simply connected to TCK and used as the test clock.

Figure 7–1. JTAG Header to Interface a Target to the Scan Controller

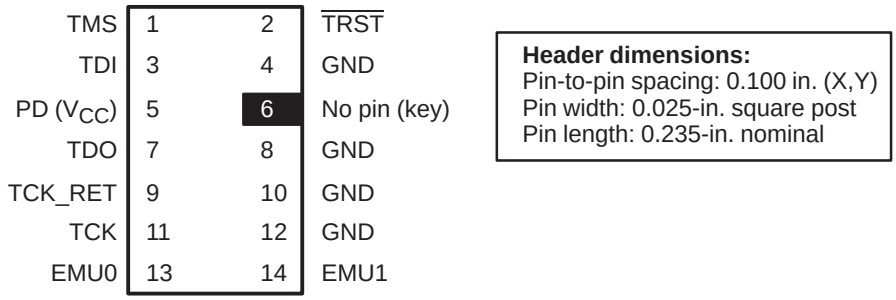


Table 7–1. 14-Pin Header Signal Descriptions

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 should be tied to V_{CC} in the target system.	I	O
TCK	Test clock. TCK is a 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. Can 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.)

The state of the $\overline{\text{TRST}}$, EMU0, and EMU1 signals at device power up determine the operating mode of the device. The operating mode takes effect as soon as the device has sufficient power to operate. Should the $\overline{\text{TRST}}$ signal rise, the EMU0 and EMU1 signals are sampled on its rising edge and the\at operating mode is latched. Some of these modes are reserved for test purposes, but those that can be of use in a target system are detailed in Table 7–2. A target system is not required to support any mode other than normal mode.

Design considerations for using an XDS510 emulation scan controller to control the debug interface are discussed in Appendix C.

Table 7–2. Selecting Device Operating Modes By Using $\overline{TRST}$, EMU0, and EMU1

$\overline{TRST}$	EMU1	EMU0	Device Operating Mode	JTAG Cable Active?
Low	Low	Low	*Slave mode.* Disables the CPU and memory portions of the '27x. Another processor treats the '27x as a peripheral.	No
Low	Low	High	Reserved for testing	No
Low	High	Low	*Wait-in-reset mode*. Prolongs the device's reset until released by external means. This allows a '27x to power up in reset, provided external hardware holds EMU0 low only while power-up reset is active.	Yes
Low	High	High	*Normal mode with emulation disabled*. This is the setting that should be used on target systems when a scan controller (such as the XDS510) is not attached. $\overline{TRST}$ will be pulled down and EMU1 and EMU0 pulled up within the '27x; this is the default mode.	No
High	Low or High	Low or High	*Normal mode with emulation enabled*. This is the setting to use on target systems when a scan controller is attached (the scan controller will control $\overline{TRST}$). $\overline{TRST}$ should not be high during device power-up.	Yes

7.3 Debug Terminology

The following definitions will help you to understand the information in the rest of this chapter:

- **Background code.** The body of code that can be halted during debugging because it is not time-critical.
- **Foreground code.** The code of time-critical interrupt service routines, which are executed even when background code is halted.
- **Debug-halt state.** The state in which the device does not execute background code.
- **Time-critical interrupt.** An interrupt that must be serviced even when background code is halted. For example, a time-critical interrupt might service a motor controller or a high-speed timer.
- **Debug event.** An action, such as the decoding of a software breakpoint instruction, the occurrence of an analysis breakpoint/watchpoint, or a request from a host processor that can result in special debug behavior, such as halting the device or pulsing one of the signals EMU0 or EMU1.
- **Break event.** A debug event that causes the device to enter the debug-halt state.

7.4 Execution Control Modes

The ’27x supports two debug execution control modes:

- Stop mode
- Real-time mode

Stop mode provides complete control of program execution, allowing for the disabling of all interrupts. Real-time mode allows time-critical interrupt service routines to be performed while execution of other code is halted. Both execution modes can suspend program execution at break events, such as occurrences of software breakpoint instructions or specified program-space or data-space accesses.

7.4.1 Stop Mode

Stop mode causes break events, such as software breakpoints and analysis watchpoints, to suspend program execution at the next interrupt boundary (which is usually identical to the next instruction boundary). When execution is suspended, all interrupts (including $\overline{\text{NMI}}$ and $\overline{\text{RS}}$) are ignored until the CPU receives a directive to run code again. In stop mode, the CPU can operate in the following execution states:

- **Debug-halt state.** This state is entered through a break event, such as the decoding of a software breakpoint instruction or the occurrence of an analysis breakpoint/watchpoint. This state can also be entered by a request from the host processor. In the stop mode debug-halt state, the CPU is halted. You can place the device into one of the other two states by giving the appropriate command to the debugger.

  The CPU cannot service any interrupts, including $\overline{\text{NMI}}$ and $\overline{\text{RS}}$ (reset). When multiple instances of the same interrupt occurs without the first instance being serviced, the later instances are lost.

- **Single-instruction state.** This state is entered when you tell the debugger to execute a single instruction by using a RUN 1 command or a STEP 1 command. The CPU executes the single instruction pointed to by the PC and then returns to the debug-halt state (it executes from one interrupt boundary to the next). The CPU is only in the single-instruction state until that single instruction is done.

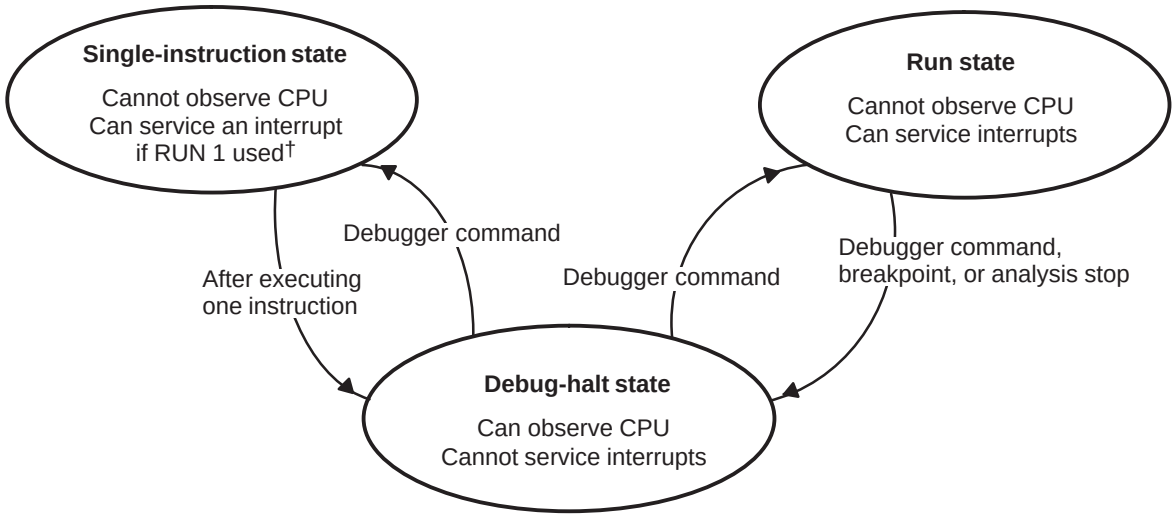
  If an interrupt occurs in this state, the command used to enter this state determines whether that interrupt can be serviced. If a RUN 1 command was used, the CPU can service the interrupt. If a STEP 1 command was used, the CPU cannot, even if the interrupt is $\overline{\text{NMI}}$ or $\overline{\text{RS}}$.

- **Run state.** This state is entered when you use a run command from the debugger interface. The CPU executes instructions until a debugger command or a debug event returns the CPU to the debug-halt state.

The CPU can service all interrupts in this state. When an interrupt occurs simultaneously with a debug event, the debug event has priority; however, if interrupt processing began before the debug event occurred, the debug event cannot be processed until the interrupt service routine begins.

Figure 7–2 illustrates the relationship among the three states. Notice that the ’27x cannot pass directly between the single-instruction and run states. Notice also that the CPU can be observed only in the debug-halt state. In practical terms, this means the contents of CPU registers and memory are not updated in the debugger display in the single-instruction state or the run state. Maskable interrupts occurring in any state are latched in the interrupt flag register (IFR).

Figure 7–2. Stop Mode Execution States



† If you use a RUN 1 command to execute a single instruction, an interrupt can be serviced in the single-instruction state. If you use a STEP 1 command for the same purpose, an interrupt cannot be serviced.

7.4.2 Real-Time Mode

Real-time mode provides for the debugging of code that interacts with interrupts that must not be disabled. Real-time mode allows you to suspend background code at break events while continuing to execute time-critical interrupt service routines (also referred to as foreground code). In real-time mode, the CPU can operate in the following execution states:

- **Debug-halt state.** This state is entered through a break event such as the decoding of a software breakpoint instruction or the occurrence of an analysis breakpoint/watchpoint. This state can also be enter by a request from the host processor. You can place the device into one of the other two states by giving the appropriate command to the debugger.

  In this state, only time-critical interrupts can be serviced. No other code can be executed. Maskable interrupts are considered time-critical if they are enabled in the debug interrupt enable register (DBGIER). If they are also enabled in the interrupt enable register (IER), they are serviced. The interrupt global mask bit (INTM) is ignored. $\overline{\text{NMI}}$ and $\overline{\text{RS}}$ are also considered time-critical, and are always serviced once requested. It is possible for multiple interrupts to occur and be serviced while the device is in the debug-halt state.

  Suspending execution adds only one cycle to interrupt latency. When the ’27x returns from a time-critical ISR, it reenters the debug-halt state.

  If a CPU reset occurs (initiated by $\overline{\text{RS}}$), the device runs the corresponding interrupt service routine until that routine clears the debug enable mask bit (DBGM) in status register ST1. When a reset occurs, DBGM is set, disabling debug events. To reenable debug events, the interrupt service routine must clear DBGM. Only then will the outstanding emulation-suspend condition be recognized.

Note:

Should a time-critical interrupt occur in real-time mode at the precise moment that the debugger receives a RUN command, the time-critical interrupt will be taken and serviced in its entirety before the CPU changes states.

- **Single-instruction state.** This state is entered when you you tell the debugger to execute a single instruction by using a RUN 1 command or a STEP 1 command. The CPU executes the single instruction pointed to by the PC and then returns to the debug-halt state (it executes from one interrupt boundary to the next).

  If an interrupt occurs in this state, the command used to enter this state determines whether that interrupt can be serviced. If a RUN 1 command was

used, the CPU can service the interrupt. If a STEP 1 command was used, the CPU cannot, even if the interrupt is $\overline{\text{NMI}}$ or $\overline{\text{RS}}$. In real-time mode, if the DBGM bit is 1 (debug events are disabled), a RUN 1 or STEP 1 command forces continuous execution of instructions until DBGM is cleared.

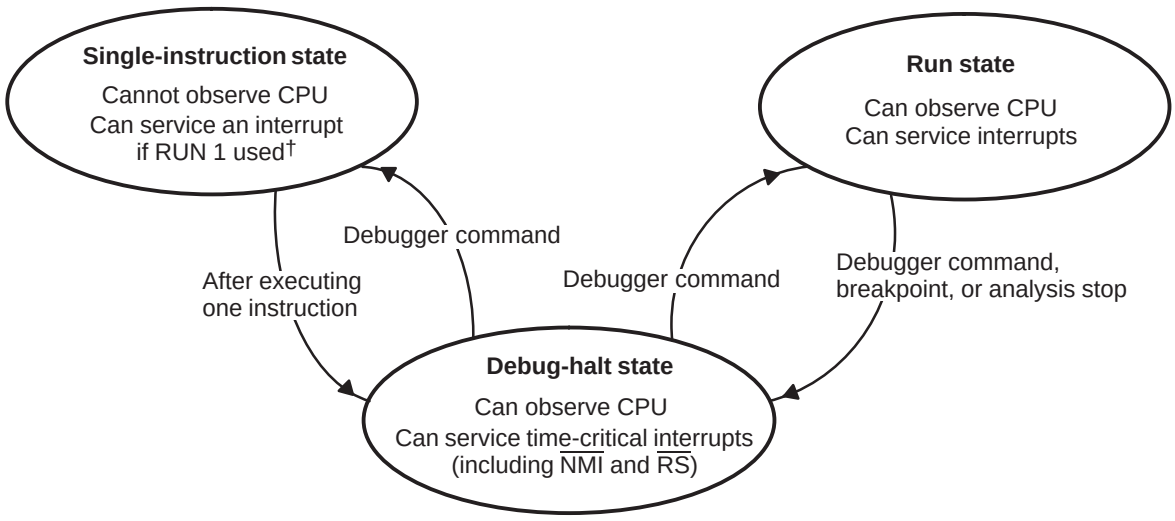
Note: If you single-step an instruction in real–time emulation mode and that instruction sets DBGM, the CPU continues to execute instructions until DBGM is cleared. If you want to single-step through a non-time-critical interrupt service routine (ISR), you must initiate a CLRC DBGM instruction at the beginning of the ISR. Once you clear DBGM, you can single-step or place breakpoints.

- **Run state.** This state is entered when you use a run command from the debugger interface. The CPU executes instructions until a debugger command or a debug event returns the CPU to the debug-halt state.

  The CPU can service all interrupts in this state. When an interrupt occurs simultaneously with a debug event, the debug event has priority; however, if interrupt processing began before the debug event occurred, the debug event cannot be processed until the interrupt service routine begins.

Figure 7–3 illustrates the relationship among the three states. Notice that the ’27x cannot pass directly between the single-instruction and run states. Notice also that the CPU can be observed in the debug-halt state and in the run state. In the single-instruction state, the contents of CPU registers and memory are not updated in the debugger display. In the debug-halt and run states, register and memory values are updated unless DBGM = 1. Maskable interrupts occurring in any state are latched in the interrupt flag register (IFR).

Figure 7–3. Real-time Mode Execution States



† If you use a RUN 1 command to execute a single instruction, an interrupt can be serviced in the single-instruction state. If you use a STEP 1 command for the same purpose, an interrupt cannot be serviced.

Caution about breakpoints within time-critical interrupt service routines

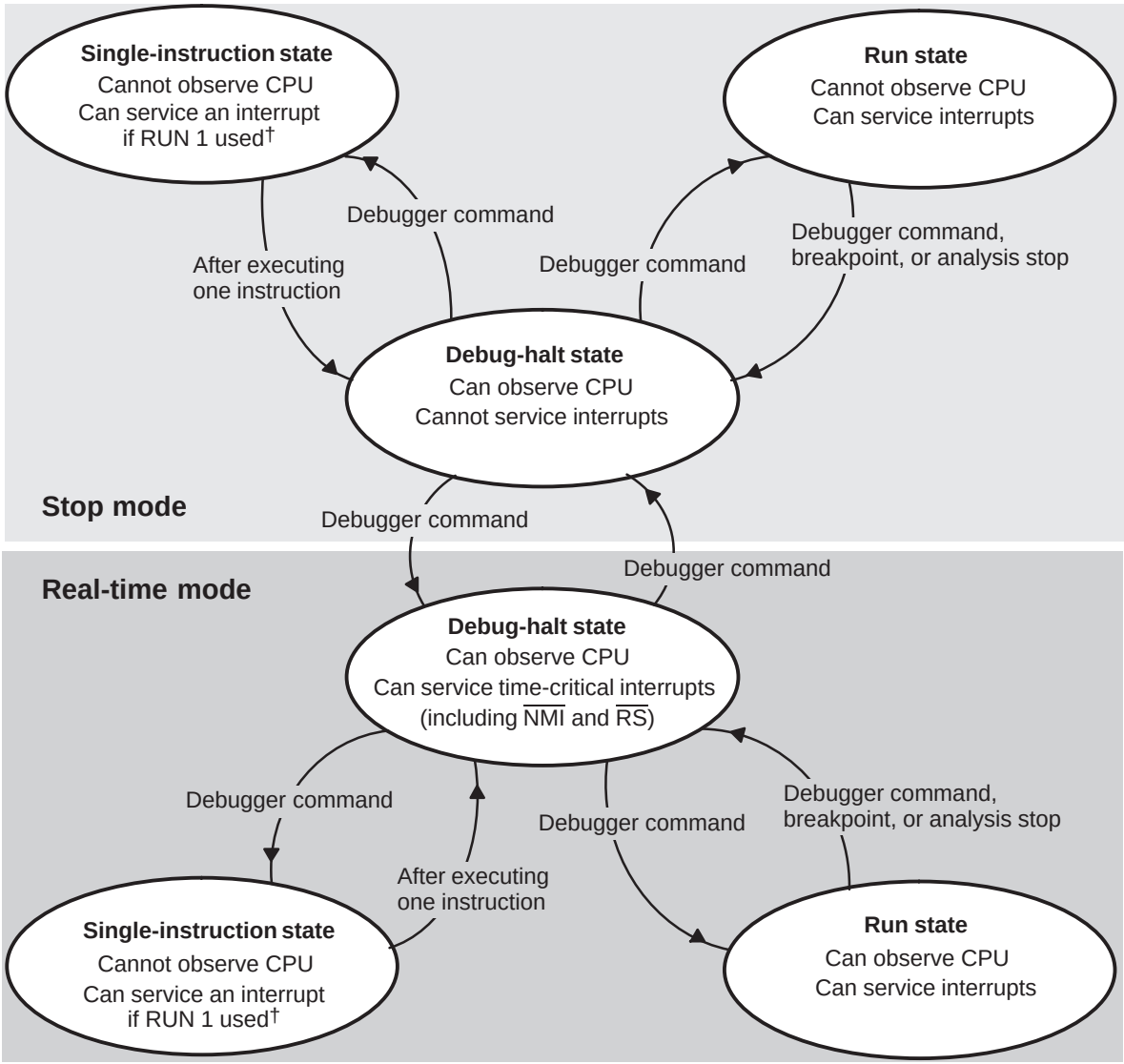
Do not use breakpoints within time-critical interrupt service routines. They will cause the device to enter the debug-halt state, just as if the breakpoint were located in normal code. Once in the debug-halt state, the CPU services requests for $\overline{RS}$, $\overline{NMI}$, and those interrupts enabled in the DBGIER and the IER.

After approving a maskable interrupt, the CPU disables the interrupt in the IER. This prevents subsequent occurrences of the interrupt from being serviced until the IER is restored by a return from interrupt (IRET) instruction or until the interrupt is deliberately re-enabled in the interrupt service routine (ISR). Do not reenable that interrupt's IER bit while using breakpoints within the ISR. If you do so and the interrupt is triggered again, the CPU performs a new context save and restarts the interrupt service routine.

7.4.3 Summary of Stop Mode and Real-Time Mode

Figure 7–4 (page 7-12) is a graphical summary of the differences between the execution states of stop mode and real-time mode. Table 7–3 (page 7-13) is a summary of how interrupts are handled in each of the states of stop mode and real-time mode.

Figure 7–4. Stop Mode Versus Real-Time Mode



† If you use a RUN 1 debugger command to execute a single instruction, an interrupt can be serviced in the single-instruction state. If you use a STEP 1 debugger command for the same purpose, an interrupt cannot be serviced.

Table 7–3. Interrupt Handling Information By Mode and State

Mode	State	If This Interrupt Occurs ...	The Interrupt Is ...
Stop	Debug-halt	$\overline{RS}$	Not serviced
		$\overline{NMI}$	Not serviced
		Maskable interrupt	Latched in IFR but not serviced
	Single-instruction	$\overline{RS}$	If running: Serviced If stepping: Not serviced
		$\overline{NMI}$	If running: Serviced If stepping: Not serviced
		Maskable interrupt	If running: Serviced If stepping: Latched in IFR but not serviced
	Run	$\overline{RS}$	Serviced
		$\overline{NMI}$	Serviced
		Maskable interrupt	Serviced
Real-time	Debug-halt	$\overline{RS}$	Serviced
		$\overline{NMI}$	Serviced
		Maskable interrupt	If time-critical: Serviced. If not time-critical: Latched in IFR but not serviced
	Single-instruction	$\overline{RS}$	If running: Serviced If stepping: Not serviced
		$\overline{NMI}$	If running: Serviced If stepping: Not serviced
		Maskable interrupt	If running: Serviced If stepping: Latched in IFR but not serviced
	Run	$\overline{RS}$	Serviced
		$\overline{NMI}$	Serviced
		Maskable interrupt	Serviced

Note:

Unless you are using a real-time operating system, do not enable the real-time operating system interrupt (RTOSINT). RTOSINT is completely disabled when bit 15 in the IER is 0 and bit 15 in the DBGIER is 0.

7.5 Aborting Interrupts With the ABORTI Instruction

Generally, a program uses the IRET instruction to return from an interrupt. The IRET instruction restores all the values that were saved to the stack during the automatic context save. In restoring status register ST1 and the debug status register (DBGSTAT), IRET restores the debug context that was present before the interrupt.

In some target applications, you might have interrupts that must not be returned from by the IRET instruction. Not using IRET can cause a problem for the emulation logic, because the emulation logic assumes the original debug context will be restored. The abort interrupt (ABORTI) instruction is provided as a means to indicate that the debug context will not be restored and the debug logic needs to be reset to its default state. As part of its operation, the ABORTI instruction:

- ❏ Sets the DBGM bit in ST1. This disables debug events.
- ❏ Modifies select bits in DBGSTAT. The effect is a resetting of the debug context. If the CPU was in the debug-halt state before the interrupt occurred, the CPU does not halt when the interrupt is aborted. Teh CPU automatically switches to the run state. If you want to abort an interrupt, but keep the CPU halted, insert a breakpoint after the ABORTI instruction.

The ABORTI instruction does not modify the DBGIER, the IER, the INTM bit, or any analysis registers (for example, registers used for breakpoints, watchpoints, and data logging).

7.6 DT-DMA Mechanism

The debug-and-test direct memory access (DT-DMA) mechanism provides access to memory, CPU registers, and memory-mapped registers (such as emulation registers and peripheral registers) without direct CPU intervention. DT-DMAs intrude on CPU time; however, you can block them by setting the debug enable mask bit (DBGM) in ST1.

Because the DT-DMA mechanism uses the same memory-access mechanism as the CPU, any read or write access that the CPU can perform in a single operation can be done by a DT-DMA. The DT-DMA mechanism presents an address (and data, in the case of a write) to the CPU, which performs the operation during an unused bus cycle (referred to as a *hole*). Once the CPU has obtained the desired data, it is presented back to the DT-DMA mechanism. The DT-DMA mechanism can operate in the following modes:

- **Nonpreemptive mode.**The DT-DMA mechanism waits for a hole on the desired memory buses. During the hole, the DT-DMA mechanism uses them to perform its read or write operation. These holes occur naturally while the CPU is waiting for newly fetched instructions, such as during a branch.

- **Preemptive mode.** In preemptive mode, the DT-DMA mechanism forces the creation of a hole and performs the access.

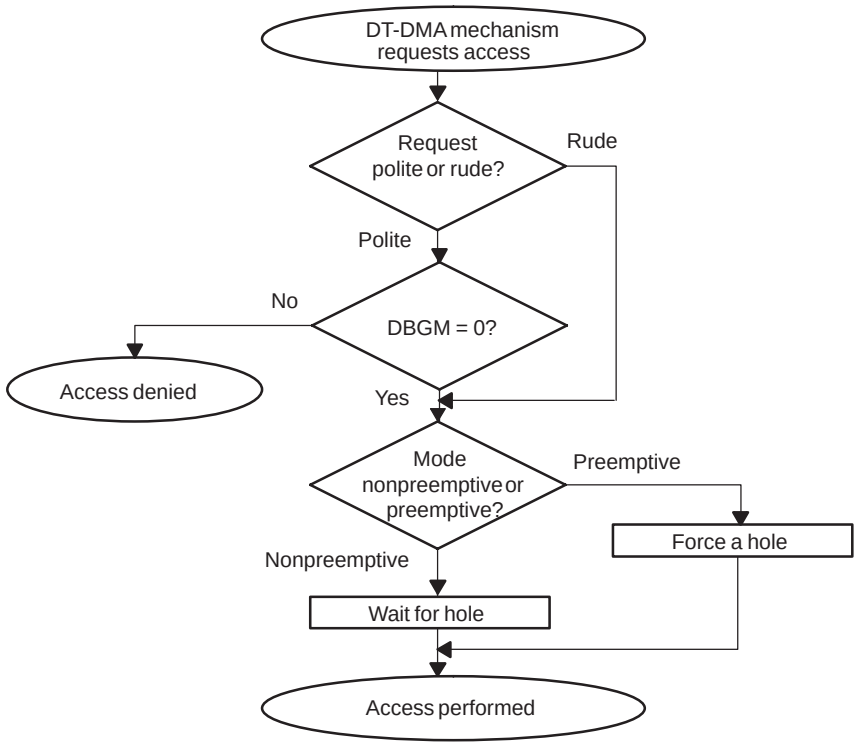
Nonpreemptive accesses to zero-wait-state memory take no cycles away from the CPU. If wait-stated memory is accessed, the pipeline stalls during each wait state, just as a normal memory access would cause a stall. In real-time mode, DT-DMAs to program memory cannot occur when application code is being run from memory with more than one wait state.

DT-DMAs can be polite or rude.

- **Polite accesses.** Polite DT-DMAs require that DBGM = 0.
- **Rude accesses.** Rude DT-DMAs ignore DBGM.

Figure 7–5 summarizes the process for handling a request from the DT-DMA mechanism.

Figure 7–5. Process for Handling a DT-DMA Request



Some key concepts of the DT-DMA mechanism are:

- Even if DBGM = 0, when the mechanism is in nonpreemptive mode, it must wait for a hole. This minimizes the intrusiveness of the debug access on a system.

- Real-time-mode accesses are typically polite (although there may be reasons, such as error recovery, to perform rude accesses in real-time mode). If the DBGM bit is permanently set to 1 due to a coding bug but you need to regain debug control, use rude accesses, which ignore the state of DBGM.

- In stop mode, DBGM is ignored, and the DT-DMA mode is set to preemptive. This ensures that you can gain visibility to and control of your system if an otherwise unrecoverable error occurs (for example, if ST1 is changed to an undesired value due to stack corruption).

- The DT-DMA mechanism does not cause a program-flow discontinuity. No interrupt-like save/restore is performed. When a preemptive DT-DMA forces a hole, no program address counters increment during that cycle.
- A DT-DMA request awakens the device from the idle state (initiated by the IDLE instruction). However, unlike returning from an interrupt, the CPU returns to the idle state upon completion of the DT-DMA.

Note:

The information shown on the debugger screen is gathered at different times from the target; therefore, it does not represent a snapshot of the target state, but rather a composite. It also takes the host time to process and display the data. The data does not correspond to the current target state, but rather, the target state as of a few milliseconds ago.

7.7 Analysis Breakpoints, Watchpoints, and Counter(s)

This section describes three types of analysis features: analysis breakpoints, watchpoints, and counters. Data logging is described in section 7.8.

7.7.1 Analysis Breakpoints

An analysis breakpoint is sometimes called a hardware breakpoint, because it acts like a software breakpoint instruction (in this case, the ESTOP0 instruction) but does not require a modification to the application software. An analysis breakpoint triggers a debug event when an instruction at a breakpoint address would have entered the decode 2 phase of the pipeline; this halts the CPU before the instruction is executed. A bus comparator watches the program address bus, comparing its contents against a reference address and a bit mask value.

Consider the following example. If a hardware breakpoint is set at T0, the CPU stops after returning from the T1 subroutine, with the instruction counter (IC) pointing to T0.

```
   NOP
   CALL   T1
T0:MOVB   AL, #0x00
   SB     TIMINGS, UNC
T1:NOP
   RET
T2:NOP
```

Hardware breakpoints allow masking of address bits. For example, a hardware breakpoint could be placed on the address range 00 0200_{16}–00 $02FF_{16}$ by specifying the following mask address, where the eight LSBs are don't cares:

00 0000 0000 0010 XXXX XXXX_2

7.7.2 Watchpoints

A hardware watchpoint triggers a debug event when either an address or an address and data match a compare value. The address portion is compared against a reference address and bit mask, and the data portion is compared against a reference data value and a bit mask.

When comparing two addresses, you can set two watchpoints. When comparing an address and a data value, you can set only one watchpoint. When performing a read watchpoint, the address is available a few cycles earlier than the data; the watchpoint logic accounts for this.

The point where execution stops depends on whether the watchpoint was a read or write watchpoint, and whether it was an address or an address/data

read watchpoint. In the following example, a read address watchpoint occurs when the address X is accessed, and the CPU stops with the instruction counter (IC) pointing three instructions after that point:

```
MOV    AR4,#X
MOV    AL,*+AR4[0]  ; Data read
nop
nop
nop                 ; The IC will point here
```

For a read watchpoint that requires both an address and data match, the CPU stops with the IC pointing six instructions after that point:

```
MOV    AR4,#X
MOV    AL,*+AR4[0]  ; Data read
nop
nop
nop
nop
nop
nop                 ; The IC will point here
```

In the following example, a write address watchpoint occurs when the address Y is accessed, and the CPU stops with the IC pointing six instructions after that point:

```
MOV    AR4,#Y
MOV    *+AR4[0],AL  ; Data write
nop
nop
nop
nop
nop
nop                 ; The IC will point here
```

7.7.3 Benchmark Counter/Event Counter(s)

The 40-bit performance counter on the ’27x can be used as a benchmark counter to increment every CPU clock cycle (it can be configured not to count when the CPU is in the debug-halt state). Wait states affect the counter. Wait states in the read 1 and write pipeline phases of an executing instruction affect the counter, regardless of whether an instruction is being single-stepped or run. However, wait states in the fetch 1 pipeline phase do not affect the counter during single-stepping, because the cycle counting does not begin until the decode 2 pipeline phase. The counter counts wait states caused by instructions that are fetched but not executed. In most cases, these effects cancel each other out. Benchmarking is best used for larger portions of code. Do not rely heavily on the precision of the benchmarking. (For more information about the pipeline, see Chapter 4.)

Alternatively, you can configure the 40-bit performance counter as two 16-bit or one 32-bit event counter if you want to generate a debug event when the counter equals a match value. The comparison between the counter value and the match value is done before the count value is incremented. For example, suppose you initialize a counter to 0. A match value of 0 causes an immediate debug event (when the action to be counted occurs), and the counter holds 1 afterward.

You can also clear the counter when a hardware breakpoint or address watchpoint occurs. With this feature, you can implement a mechanism similar to a watchdog timer: if a certain address is not seen on the address bus within a certain number of CPU clock cycles, a debug event occurs.

7.8 Data Logging

Data logging enables the ’27x to send selected memory values to a host processor using the standard JTAG port and an XDS510 or other compatible scan controller. You control data logging activity with your application code.

To perform data logging, you must create a linear buffer of 32-bit words to hold a packet of information. Your application code controls the size, format, and location of this buffer and also determines when to send a buffer’s contents to the host. You can control the size of a data logging buffer in two ways:

- ❑ Specify a count value in the upper eight bits of ADDRH (when the number of 32-bit words you want to log is between 1 and 256)
- ❑ Specify an end address

Note:

When the debugger is not active, the data logging transfers are considered complete as soon as they are enabled to prevent the application software from getting stuck when there is nothing to receive the data.

7.8.1 Creating a Data Logging Transfer Buffer

To create a data logging transfer buffer, follow these steps in your application code:

1) Execute the EALLOW instruction to enable access to emulation registers.

2) Specify the start address of the buffer in ADDRL and the six LSBs of ADDRH (see Figure 7–6 and Figure 7–7). The address in ADDRL and ADDRH is called the transfer address.

3) Use either of the following methods to specify when data logging is to end:

   a) If the number of words you want to log is between 1 and 256, specify a count value in the upper eight bits of ADDRH (see Figure 7–7). The form of the count value is 256–*n,* where *n* is the number of 32-bit words you want to log. As each word is transferred, both the transfer address and the count value are decremented.

   b) If the number of words you want to log is greater than 256, specify a data logging end address in REFL and the six LSBs of REFH (see Figure 7–8 and Figure 7–9). Load the ten MSBs of REFH with 0s. When using this method, be sure to set the data logging end address control register (EVT_CNTRL) first, and then the DMA control register

(DMA_CNTRL). EVT_CNTRL is described in Table 7–5 (page 7-24), and DMA_CNTRL is described in Table 7–4 (page 7-23).

Note:

The application must *not* read from the end address of the buffer during the data logging operation. When the end address appears on the address bus, the ’27x ends the transfer.

4) Execute the EDIS instruction to disable access to emulation registers.

See Table 7–4 and Table 7–5 on the following pages for descriptions of the registers associated with data logging.

Figure 7–6. ADDRL (at Data-Space Address 00 0838_{16})

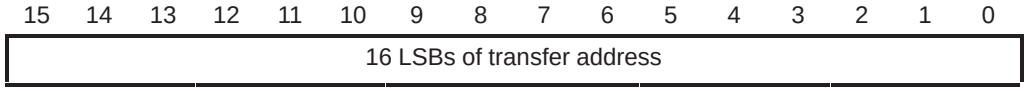


Figure 7–7. ADDRH (at Data-Space Address 00 0839_{16})

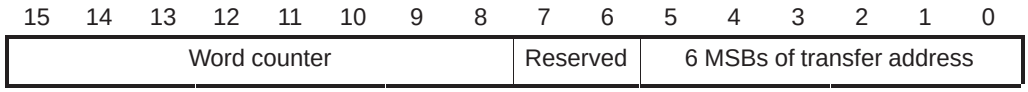


Figure 7–8. REFL (at Data-Space Address 00 $084A_{16}$)

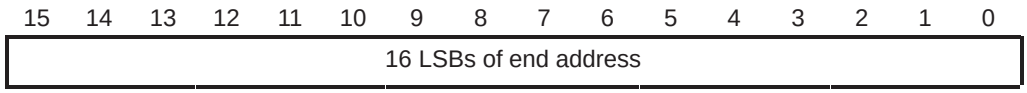


Figure 7–9. REFH (at Data-Space Address 00 $084B_{16}$)

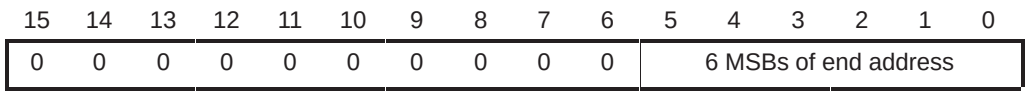


Table 7–4. Start Address and DMA Registers

Address	Name	Access	Description
00 0838_{16}	ADDRL	R/W	**Start address register (lower 16 bits)** 15:0 Lower 16 bits of start address
00 0839_{16}	ADDRH	R/W	**Word counter/start address register (upper 6 bits)** 15:8 Word counter. When using this to stop the data logging transfer, set the counter to 256 – *n*, where *n* is the number of 32-bit words to transfer. Otherwise set the counter to 0. 7:6 Reserved. Set to 0. 5:0 Upper 6 bits of start address
00 $083E_{16}$	DMA_CNTRL	R/W	**DMA control register** 15:14 Set to 0 13 Set to 1 12 Set to 1 11 Give higher priority to: 0: CPU (nonpreemptive mode) 1: Data logging (preemptive mode) 10 Allow data logging during time-critical ISR? 0: No 1: Yes 9 Allow data logging while DBGM = 1? 0: No (polite accesses) 1: Yes (rude accesses) 8:6 Set to 1 5:4 0: EMU0/EMU1 using TCK 1: EMU0/EMU1 using FCK/2 2: JTAG signals 3: Reserved 3:2 Method for ending data logging session: 0: Use the count register to stop data logging 1: Use an end address to stop data logging 1:0 Data logging control/status: 0: Release resource from data logging operation 1: Claim resource for data logging operation 2: Enable resource for data logging operation 3: Data logging operation is complete. Bits 14:10 are corrupted when this occurs.
00 $083F_{16}$	DMA_ID	R	**DMA ID register** 15:14 Resource control: 0: Resource is free 1: Application owns resource 2: Debugger owns resource 13:12 Set to 3. 11:0 Set to 1.

Table 7–5. End-Address Registers

Address	Name	Access	Description
00 0848_{16}	MASKL	R/W	Set to 0
00 0849_{16}	MASKH	R/W	Set to 0
00 $084A_{16}$	REFL	R/W	**Data logging end reference address (lower 16 bits)** 15:0 Lower 16 bits of start address
00 $084B_{16}$	REFH	R/W	**Data logging end reference address (upper 6 bits)** 15:6 Set to 0 5:0 Upper 6 bits of start address
00 $084E_{16}$	EVT_CNTRL	R/W	**Data logging end address control register** 15:14 Set to 0 13 Set to 1 12 Set to 1 11:5 Set to 0 4:2 Set to 1 1:0 End-address resource control/status: 0: Release end-address resource. 1: Claim end-address resource. 2: Enable end-address resource. 3: Data logging operation has ended. Bits 14:10 are corrupted when this occurs.
00 $084F_{16}$	EVT_ID	R	**Data logging end address ID register** 15:14 Resource control: 0: Resource is free 1: Application owns resource 2: Debugger owns resource 13:12 Set to 1 11:0 Set to 2

7.8.2 Accessing the Emulation Registers Properly

Make sure your application code follows the following protocol when accessing the emulation registers that have been provided for data logging. Each resource has a control register and an ID register.

1) Enable writes to memory-mapped registers by using the EALLOW instruction.

2) Write to the appropriate control register to claim the resource you want to use. The resource for data logging transfers uses DMA_CNTRL (see Table 7–4 on page 7-23). The resource for detecting the data logging end address uses EVT_CNTRL (see Table 7–5).

3) Wait at least three cycles so that the write to the control register (done in the write phase of the pipeline) occurs before the read from the ID register in step 4. You can fill in the extra cycles with NOP (no operation) instructions or with other instructions that do not involve accessing the emulation registers.

4) Read the appropriate ID register and verify that the application is the owner. The resource for data logging transfers uses DMA_ID (see Table 7–4 on page 7-23). The resource for detecting the data logging end address uses EVT_ID (see Table 7–5 on page 7-24). If the application is not the owner, then go back to step 2 until this succeeds (you may want a time-out function to prevent an endless loop). This step is optional. The application would fail to become the owner only if the debugger already owns the resource.

5) If the application is the owner, the remaining registers for that function can be programmed, and the control register written to again, to enable the function. However, if the application is not the owner, then all of its writes are ignored.

6) Disable writes to memory-mapped emulation registers by executing the EDIS instruction.

If an interrupt occurs between the EALLOW instruction in step 1 and the EDIS instruction in step 6, access to emulation registers are automatically disabled by the CPU before the interrupt service routine begins and automatically reenabled when the CPU returns from the interrupt. This means that there is no need to disable interrupts between the EALLOW instruction and the EDIS instruction.

The debugger can, at your request, seize ownership of a register from the application; however, that is not the normal mode of operation.

7.8.3 Data Log Interrupt (DLOGINT)

The completion of a data logging transfer (determined either by the word counter or by the end address) triggers a DLOGINT request. DLOGINT is serviced only if it is properly enabled. If the CPU is halted in real-time mode, DLOGINT must be enabled in both the DBGIER and the IER. Otherwise, DLOGINT must be enabled in the IER and by the INTM bit in status register ST1.

This interrupt capability is most useful when there are multiple buffers of data to be transferred through data logging and the completion of one transfer should begin the next.

7.8.4 Examples of Data Logging

Example 7–1 shows how to log 20 32-bit words, starting at address 00 0100_{16} in data memory. The accesses are preemptive (they have higher priority than the CPU) and rude (they ignore the state of the DBGM bit). In addition, data logging can occur during time-critical interrupt service routines. The application can determine whether the data logging operation is complete by polling the LSB of the DMA control register (DMA_CNTRL) at 00 $083E_{16}$. When the operation is complete, that bit is set to 1.

Example 7–1. Initialization Code for Data Logging With Word Counter

```
; Base addresses
ADMA      .set   0838h

; Offsets
DMA_ADDRL .set   0
DMA_ADDRH .set   1
DMA_CNTRL .set   6
DMA_ID    .set   7

EALLOW
MOV    AR4, #ADMA                 ; AR4 pointing to register base addr
MOV    *+AR4[#DMA_CNTRL],#1       ; Attempt to claim resource
NOP
NOP
NOP
CMP    *+AR4[#DMA_ID],#7001h      ; Value expected in ID register
B      FAIL, NEQ                  ; If we don't see the correct ID, then we
                                  ;  failed (the resource is already in use)

MOV    *+AR4[#DMA_ADDRL],#0100h   ; Set starting address of buffer,
                                  ;  and then the count
MOV    *+AR4[DMA_ADDRH],#((256 – 20) << 8)

MOV    *+AR4[DMA_CNTRL],#3E62h
EDIS
```

Example 7–2 shows how to log from address 00 0100_{16} to address 00 $02FF_{16}$ in data memory. The accesses are nonpreemptive (they have lower priority than the CPU), and are polite (they are not performed when the DBGM bit is 0). The data logging cannot occur when a time-critical interrupt is being serviced. An end address of 00 $02FF_{16}$ is used to end the transfer. The application must not read from 00 $02FF_{16}$ during the data logging; a read from that address stops the data logging. As in Example 7–1, the application can poll the LSB of DMA_CNTRL for a 1 to determine whether the data logging operation is complete.

Example 7–2. Initialization Code for Data Logging With End Address

```
; Base addresses
ADMA      .set   0838h
DEVT      .set   0848h

; Offsets
DMA_ADDRL .set   0
DMA_ADDRH .set   1
DMA_CNTRL .set   6
DMA_ID    .set   7
MASKL     .set   0
MASKH     .set   1
REFL      .set   2
REFH      .set   3
EVT_CNTRL .set   6
EVT_ID    .set   7

EALLOW
MOV    AR5, #DEVT                ; AR5 pointing to End Address registers
MOV    AR4, #ADMA                ; AR4 pointing to Start/Control base
MOV    *+AR5[#EVT_CNTRL],#1      ; Attempt to claim End Address
MOV    *+AR4[#DMA_CNTRL],#1      ; Attempt to claim Start/Control
NOP
NOP
NOP
CMP    *+AR5[#EVT_ID],#5002h     ; Value expected in ID register
B      FAIL, NEQ                 ; If we don't see the correct ID, FAIL

CMP    *+AR4[#DMA_ID],#7001h     ; Value expected in ID register
B      FAIL, NEQ                 ; If we don't see the correct ID, FAIL

MOV    *+AR5[#MASKL],#0          ; Attempt to claim End Address
MOV    *+AR5[#MASKH],#0          ; Attempt to claim End Address
MOV    *+AR5[#REFL],#02FFh       ; Stop data logging at address 0x02FF
MOV    *+AR5[#REFH],#0           ; Attempt to claim End Addr
MOV    *+AR5[#EVT_CNTRL],#  ( 2 |  (1<<2)  |  (1<<12)  |  (1<<13)  )

MOV    *+AR4[#DMA_ADDRL],#0100h  ; Set buffer start address and then the count
MOV    *+AR4[DMA_ADDRH],#0

MOV    *+AR4[DMA_CNTRL],#3066h
EDIS
```

7.9 Sharing Analysis Resources

You can use analysis breakpoints, watchpoints, and a benchmark/event counter through the debugger, and you can use data logging through application code. Table 7–6 lists the analysis resources, and Figure 7–10 shows which resources are available to be used at the same time.

When the application owns analysis resources, they will be cleared (made unowned and set to the completed state) by a reset. When the debugger owns the resources, they are not cleared by reset but by the JTAG test-logic reset. This ensures that when you are using the debugger, the resources can be used even while the target system undergoes a reset.

Table 7–6. Analysis Resources

Resource	Purpose
BA0	Break on contents of program address or memory address bus
BA1	Break on contents of program address or memory address bus
BD	Break on contents of program data, memory read data, or memory write data in addition to an address bus
Data log	Perform data logging using counter
Benchmark	Count CPU cycles

Figure 7–10. Valid Combinations of Analysis Resources

	BA0	BA1	BD	Data log	Benchmark
BA0	Yes	Yes	No	Yes†	Yes
BA1	Yes	Yes	No	No	No
BD	No	No	Yes	No	No
Data log	Yes†	No	No	Yes	No
Benchmark	Yes	No	No	No	Yes

† The data logging mode that uses the word counter allows this combination, but not the data logging mode that uses the end address (see section 7.8, *Data Logging*).

7.10 Diagnostics and Recovery

Debug registers within the core keep track of the state of several key signals. This allows diagnosis of such problems as a floating READY signal, $\overline{\text{NMI}}$ signal, or $\overline{\text{RS}}$ (reset) signal. Should the debug software attempt an operation that does not complete after a certain time-out period (as determined by the debug software), it attempts to determine the probable cause and display the situation to you. You can then abort, correct the situation or allow it to correct itself, or chose to override it.

Such situations include:

- $\overline{\text{RS}}$ being asserted
- A ready signal not being asserted for a memory access
- $\overline{\text{NMI}}$ being asserted
- The absence of a functional clock
- The occurrence of a JTAG test-logic-reset

Chapter 8

Assembly Language Programming Tips

To help you manage your programs for the ’27x, this chapter introduces special assembly language instructions and helpful programming techniques.

For more information about the instructions mentioned throughout this chapter, see Chapter 6, *Assembly Language Instructions*.

Topic **Page**

8.1 Initializing the Processor After Reset 8-2
8.2 Performing Special Branch Operations 8-4
8.3 Managing Interrupts 8-9
8.4 Using the LOOPZ and LOOPNZ Instructions 8-17
8.5 Managing Memory 8-20
8.6 Implementing a Circular Buffer With Circular Addressing Mode .. 8-30

8.1 Initializing the Processor After Reset

A hardware reset returns pointers and status bits to their reset values, which may or may not be the values your program requires. When writing initialization routines, you may wish to consult Table 8–1 and Table 8–2 to make sure your program changes any pointers and key status bits that must hold values other than their reset values.

Table 8–1. Setting Pointers

Pointer(s)	Reset Value	How to Change the Pointer Value

AR0–AR5	0	If you need to use these auxiliary registers as pointers to memory, set them to the desired values using one of the following syntaxes of the MOV instruction:
		MOV AR*x*, *loc*
		MOV @AR*x*, #*16bit*

XAR6, XAR7	0	If you need to use these extended auxiliary registers as pointers to memory, set them to the desired values using one of the following syntaxes of the MOV instruction:
		MOV XAR*n*, *locLong*
		MOV XAR*n*, #22*bit*

DP	0
	(data page 0 selected)
		To make the DP point to a different data page, use one of the following syntaxes:
		MOV DP, #*10bit*
		MOVW DP, #*16bit*
		The MOV syntax loads the 10 least significant bits of the DP. MOVW loads the entire DP.

SP	0	To make sure the SP points to the correct stack address, use the following MOV syntax: MOV @SP, #*16bit*

Table 8–2. Setting Key Status Bits

Status Bit(s)	Reset Value	Reset Condition	How to Change the Value
SXM	0	Sign-extension mode is off.	If you need sign-extension mode on, set SXM with the SETC SXM instruction.
OVM	0	Overflow mode is off.	If you need overflow mode on, set SXM with the SETC OVM instruction.
PM	0	Instructions that are affected by PM will shift the P register value left by 1 bit.	If you need a different product shift mode, change PM with the following instruction: SPM *ShiftMode* *ShiftMode* — Mode 1 — Shift left by 1 0 — No shift –1 — Shift right by 1 –2 — Shift right by 2 –3 — Shift right by 3 –4 — Shift right by 4 –5 — Shift right by 5 –6 — Shift right by 6
INTM	1	Maskable interrupts are disabled.	To enable maskable interrupts, clear INTM with the CLRC INTM instruction.
DBGM	1	Debug events are disabled.	To enable debug events, clear DBGM with the CLRC DBGM instruction.
PAGE0	0	PAGE0 stack addressing mode is selected.	If instead, you want PAGE0 direct addressing mode selected, use the SETC PAGE0 instruction.
VMAP	0	The interrupt vectors are mapped to addresses 000000_{16}–$00003F_{16}$ in program space.	To map the interrupt vectors to addresses $3F\ FFC0_{16}$–$3F\ FFFF_{16}$, use the SETC VMAP instruction.
EALLOW	0	Access to emulation registers is not allowed.	To enable access to the emulation registers, use the EALLOW instruction. (To clear the EALLOW bit again, use the EDIS instruction.)
ARP	0	AR0 is the current auxiliary register.	If you want the ARP to point to a different auxiliary register, you can use one of the following syntaxes: NOP *AR*x* (*x* is 0, 1, 2, 3, 4, or 5) NOP *XAR*n* (*n* is 6 or 7)

8.2 Performing Special Branch Operations

Special branch operations that can be implemented on the ’27x include:

- ❑ Branching to an address determined at run time (see section 8.2.1)
- ❑ Creating a fast function call and return (see section 8.2.2)
- ❑ Branching based on a single condition (see section 8.2.3)
- ❑ Branching based on multiple conditions (see section 8.2.4)

For more details about instructions mentioned in this section, see Chapter 6, *Assembly Language Instructions*.

8.2.1 Branching to an Address Determined at Run Time

The following instructions enable your programs to branch to addresses it computed or loaded earlier:

- ❑ **LB *XAR7**

  The address in extended auxiliary register XAR7 is loaded into the program counter (PC).

- ❑ **CALL *XAR7**

  The return address (the address of the next instruction) is stored to the stack. Then the address in XAR7 is forced into the PC. When the subroutine/function is complete, a return instruction loads the return address to the PC.

The following example loads XAR7 from a data-memory location and then uses XAR7 for a branch.

```
      MOV   XAR7, *AR1   ; Load XAR7 from memory location.
                         ; pointed to by AR1. Depending on
                         ; AR1, XAR7 is loaded with
                         ; address for PartB or PartC.
      LB *XAR7           ; Branch to PartB or PartC.
      .
      .
      .
PartB:Code for PartB...
      .
      .
      .
PartC:Code for PartC...
      .
      .
      .
```

8.2.2 Creating a Fast Function Call and Return

The standard call-return operation uses the CALL instruction and the RET instruction. The CALL instruction stores the return address to two 16-bit locations in the stack. The RET must perform two data transfers to the restore the program counter (PC). You can avoid using the stack and restore the PC with a single data transfer by creating a fast function call and return. To do this, call the function with the FFC (fast function call) instruction and return from the subroutine with the LB *XAR7 instruction (see section 8.2.1).

The FFC instruction saves the return address to extended auxiliary register XAR7, and the LB *XAR7 transfers the return address from XAR7 to the PC. For example:

```
      FFC   XAR7, FuncA     ; Save return address to XAR7.
                            ; Call function A.
      .
      .
      .
FuncA:Code for function A...
      .
      .
      LB    *XAR7           ; Load PC with return address.
```

8.2.3 Branching Based on a Single Condition

The following instructions allow a program to branch to a new address based on a single condition:

- ❏ BANZ instruction (branch if auxiliary register not equal to zero)
- ❏ B instruction (branch)
- ❏ SB instruction (short branch)

This section introduces these instructions. For more details about them, see their descriptions in Chapter 6, *Assembly Language Instructions*.

BANZ instruction

The BANZ instruction has the following syntax:

BANZ *16BitOffset*, **AR***x***––**

16BitOffset is a 16-bit signed offset that can range from –32 768 to 32 767. For ARx, use any one of the 16-bit auxiliary registers AR0–AR7. If the content of the specified auxiliary register is not zero, program control is forced to the new address, PC + *16BitOffset*. Regardless of whether the branch is taken, the auxiliary register is decremented by 1.

One application for the BANZ instruction is in a loop. In the following example, auxiliary register AR2 acts as a loop counter. If AR2 holds a number N, the program loops N times after the initial execution of the ADD instruction. The result is N + 1 additions to the accumulator (ACC). The ADD instruction uses an indirect addressing mode that increments AR5 with each execution; therefore, a new value is added to ACC during each execution.

```
Loop   ADD    ACC, *AR5++     ; Add memory value to ACC.
       BANZ   Loop, AR2–-     ; If AR2 not 0 yet, add again.
```

B and SB instructions

The syntax for the B instruction follows:

B *16BitOffset*, *cond*

16BitOffset is a 16-bit signed offset that can range from –32 768 to 32 767, and *cond* is one of the conditions listed in Table 8–3. If the specified condition is true, the specified 16-bit offset is added to the current PC value (the start address of the B instruction). Program control is forced to the new address, PC + *16BitOffset*.

As an example, suppose you want to test the accumulator (ACC) and then branch forward 200 locations if ACC = 0. The code would be:

```
TEST   ACC       ; Compare ACC to 0 (ACC – 0 = ?)
                 ; If ACC = 0, Z is set.
B      200, EQ   ; If Z = 1, branch forward.
```

The syntax for the SB instruction is similar to that of the B instruction:

SB *8BitOffset*, *cond*

The difference is a shorter offset. *8BitOffset* is an 8-bit signed offset that can range from –128 to 127. If the specified condition is true, program control is forced to address PC + *8BitOffset*.

In the following example, the program branches backward 15 locations if the addition generates a carry.

```
ADD    ACC, #60  ; Add 60 to accumulator (ACC).
                 ; If carry generated, C is set.
SB     –15, C    ; If C = 1, branch backward.
```

Table 8–3. Conditions and Their Corresponding Flag Tests

cond	**Condition**	**Flag Test Performed**
NEQ	Not equal to 0	Z = 0
EQ	Equal to 0	Z = 1
GT	Greater than 0	Z = 0 AND N = 0
GEQ	Greater than or equal to 0	N = 0
LT	Less than 0	N = 1
LEQ	Less than or equal to 0	Z = 1 OR N = 1
HI	Higher	C = 1 AND Z = 0
HIS or C	Higher or same or C = 1	C = 1
LO or NC	Lower or C = 0	C = 0
LOS	Lower or same	C = 0 OR Z = 1
NOV	No overflow	V = 0
OV	Overflow	V = 1
NTC	TC = 0	TC = 0
TC	TC = 1	TC = 1
UNC	Unconditional	None

8.2.4 Branching Based on Multiple Conditions

You might want your program to make a branch decision based on more than one of the conditions in Table 8–3 (page 8-7). To use multiple conditions, you must use a series of B and/or SB instructions. In the following example, the program branches to a subroutine called Routine2 only if C = 1, V = 0, and Z = 1.

```
      SB Skip, NC     ; If C = 0, skip the branches.
      SB Skip, OV     ; If V = 1, skip the next branch.
      B  Routine2,EQ  ; If Z = 1, branch to Routine2.
Skip: .
      .
      .
```

The assembler converts the label Skip into the appropriate numerical offset for each of the SB instructions. If you use numerical offsets instead of using a label, keep the following points in mind:

- Before a branch instruction adds an offset to the program counter (PC), the PC is pointing to the start address of that branch instruction.
- The SB instruction fills one word in memory; the B instruction fills two words. Consider the code example again. The second SB instruction would use an offset of 3 to branch over the two words of the B instruction. The first SB instruction would use an offset of 4 to branch over the three words of the other branch instructions.
- If the offset for a B instruction is a value from –128 to 127, the assembler will convert the B instruction into an SB instruction.

8.3 Managing Interrupts

The discussions in this section can help you make the most of the ’27x interrupts and related features. These discussions assume that you understand the material in Chapter 3, *Interrupts and Reset*. They cover the following topics:

- ❏ Assigning interrupt vectors (see section 8.3.1)
- ❏ Using the interrupt acknowledge instruction (see section 8.3.3)
- ❏ Changing register values that were saved on the stack (see section 8.3.4)
- ❏ Using nested interrupts (see section 8.3.5)
- ❏ Checking interrupt flags (see section 8.3.6)

8.3.1 Assigning Interrupt Vectors

An interrupt vector is a start address for an interrupt service routine (ISR). The ’27x allots two consecutive 16-bit locations for each vector. The location at the lower address must hold the 16 least significant bits of the 22-bit vector. The location at the higher address must hold the six most significant bits right-justified.

To store an interrupt vector, you can use the .long directive of the assembler. This directive stores a 32-bit value in the required format (the least significant word at the lower address). For example, suppose you want to store a vector for interrupt $\overline{\text{INT1}}$, and suppose the start address for its ISR is labeled Int1_ISR. The following directive stores that start address and labels it INT1.

```
INT1   .long  Int1_ISR
```

When an interrupt occurs, the CPU reads the value at the corresponding position in the interrupt vector table. For example, every time the CPU approves a reset request, it reads the reset vector from the two lowest addresses in the table. In addition, if a vector has not been assigned for an interrupt and the interrupt occurs, the CPU uses whatever value happens to be at the vector location.

To avoid incorrect vector reads, it is best to assign vectors for all 32 interrupts. Be sure to assign them in the order given in Table 3–1 (page 3-3). Example 8–1 shows hypothetical assignments for all the interrupts. Every interrupt that is not used by the program is assigned a vector that points to a special routine called Unused. This routine uses the IACK instruction to notify an external device and then forces the program to return from the interrupt. Use of the IACK instruction is discussed in section 8.3.2.

Example 8–1. Assigned Interrupt Vectors

```
;*************************
; Interrupt vector table *
;*************************
RESET     .long  Reset_ISR
INT1      .long  Int1_ISR
INT2      .long  Int2_ISR
INT3      .long  Int3_ISR
INT4      .long  Int4_ISR
INT5      .long  Int5_ISR
INT6      .long  Int6_ISR
INT7      .long  Int7_ISR
INT8      .long  Int8_ISR
INT9      .long  Unused       ; Interrupt not used.
INT10     .long  Unused       ; Interrupt not used.
INT11     .long  Unused       ; Interrupt not used.
INT12     .long  Unused       ; Interrupt not used.
INT13     .long  Unused       ; Interrupt not used.
INT14     .long  Unused       ; Interrupt not used.
DLOGINT   .long  DLOG_ISR
RTOSINT   .long  Unused       ; Interrupt not used.
Reserved  .long  Unused       ; Interrupt not used.
NMI       .long  NMI_ISR
ILLEGAL   .long  ILGL_ISR
USER1     .long  ISR20        ; Run using TRAP #20
USER2     .long  ISR21        ; Run using TRAP #21
USER3     .long  Unused       ; Interrupt not used.
USER4     .long  Unused       ; Interrupt not used.
USER5     .long  Unused       ; Interrupt not used.
USER6     .long  Unused       ; Interrupt not used.
USER7     .long  Unused       ; Interrupt not used.
USER8     .long  Unused       ; Interrupt not used.
USER9     .long  Unused       ; Interrupt not used.
USER10    .long  Unused       ; Interrupt not used.
USER11    .long  Unused       ; Interrupt not used.
USER12    .long  Unused       ; Interrupt not used.

;****************************************
; Special subroutine for unused interrupts *
;****************************************
Unused    IACK #Unused        ; Notify external device.
          IRET                ; Return from interrupt.
```

8.3.3 Using the Interrupt Acknowledge (IACK) Instruction

The IACK instruction can be used to acknowledge interrupts or indicate the occurrence of some internal event with the use of external signals. Issuing an IACK instruction causes the operand, a 16-bit constant, to be placed on the lower 16 bits of the data-write data bus, DWDB(15:0), and the IACK signal to

be asserted during the write operation. Therefore, the assertion of IACK and the presence of a specified value on DWDB(15:0) can indicate to an external device that a particular event has occurred in the code (for example, an interrupt has been recognized and approved).

As an example, suppose the IACK instruction notifies an external device that interrupt $\overline{\text{INT2}}$ has been approved and that its corresponding interrupt service routine (ISR) is being executed. In the interrupt vector table, a vector for $\overline{\text{INT2}}$ has been defined:

```
INT2   .long  Int2_ISR        ; Set vector for INT2.
```

When $\overline{\text{INT2}}$ is approved by the CPU, the vector forces a branch to the following ISR. The IACK instruction sends the start address of the ISR (referenced by the label INT2_ISR) out on DWDB(15:0). Custom logic can be added to check the IACK signal and DWDB(15:0) to determine when this ISR begins.

```
Int2_ISR:    IACK  #INT2_ISR ; Send interrupt acknowledge.
             .
             .               ; Perform function for INT2.
             .
             IRET            ; Return
```

8.3.4 Changing the Values Saved During the Automatic Context Save

As discussed in Chapter 3, *Interrupts and Reset*, the CPU automatically saves certain CPU registers and the return address to the stack before it executes the interrupt service routine (ISR). There may be times when it is necessary to read or modify one of these registers or the return address before returning from the ISR. For example, suppose your ISR must check whether the CPU was executing a loop instruction (LOOPNZ or LOOPZ) when the interrupt occurred. The ISR must read the stored ST1 value and check the value in the LOOP bit position. This section reviews important points of the automatic context save and then describes a technique to ensure that you are reading and modifying the correct stack values.

Review of the automatic context save

Figure 8–1 shows two scenarios for the automatic context save. At this point the SP has already been incremented by 1 in preparation for the automatic context save.On the left side of the figure, the stack pointer (SP) points to an odd address before the context save. The right side of the figure shows the same operation when the SP points to an even address. Important points to remember are:

- **The values are saved in pairs and aligned to even addresses.** Each pair is saved to the stack by a single 32-bit write operation. All the 32-bit

writes are aligned to even addresses. For example, suppose the SP points to an odd address before the CPU saves the first register pair, T:ST0 (see the left side of Figure 8–1). ST0 is saved to the previous even address, and T is saved to the odd address. Regardless of the starting address in the SP, the register values and the return address are saved to the same locations.

- **After the context save, SP could be odd or even.** After each register pair is saved, the SP is incremented by 2. Therefore, if the SP starts at an odd address, it points to an odd address when the context save is done (see the left side of Figure 8–1). Likewise, if the SP starts at an even address, it points to an even address when the save is done (see the right side of Figure 8–1).

Figure 8–1. Accessing Values Saved to the Stack During Interrupt Processing

SP starts at odd address

SP position	Stack contents	Operand
	ST0	
Before save →	T	*–SP[14]
(odd address)	AL	
	AH	*–SP[12]
	PL	
	PH	*–SP[10]
	AR0	
	AR1	*–SP[8]
	ST1	
	DP	*–SP[6]
	IER	
	DBGSTAT	*–SP[4]
	Return address (16 LSBs)	
	Return address (6 MSBs)	*–SP[2]
	Empty location	
After save →	Empty location	
(odd address)		

SP starts at even address

SP position	Stack contents	Operand
Before save →	ST0	*–SP[14]
(even address)	T	
	AL	*–SP[12]
	AH	
	PL	*–SP[10]
	PH	
	AR0	*–SP[8]
	AR1	
	ST1	*–SP[6]
	DP	
	IER	*–SP[4]
	DBGSTAT	
	Return address (16 LSBs)	*–SP[2]
	Return address (6 MSBs)	
After save →	Empty location	
(even address)	Empty location	

Reading and modifying the correct values

Without some planning ahead, it is easy to be off by one address when you try to read or modify one of the saved values. Rather than try to predict whether the SP will end up odd or even, you can take advantage of the fact that 32-bit operations are aligned to even addresses. To make sure you read the correct value, read its pair. For example, if you want PL, read PL:PH. Once you have brought a pair into the CPU, you can manipulate either member of that pair with the appropriate instructions. Then, if necessary, you can replace the old stack value with a new one.

To get a pair into the CPU, you can copy it to the 32-bit accumulator (ACC) with this form of the MOVL instruction:

MOVL ACC, *–SP[*6bit*]

The operand *–SP[*6bit*] belongs to PAGE0 stack addressing mode (see section 5.4.2 on page 5-8.). It references a stack value by subtracting a 6-bit offset (*6bit*) from the value in the SP. Before using this operand, make sure PAGE0 stack addressing mode is enabled: clear the PAGE0 bit in status register ST1.

It is important to use even values (2, 4, 6, etc.) for the 6-bit offset. As shown in Figure 8–1 (page 8-12), each even offset corresponds to one (and only one) stack pair, regardless of whether the SP is odd or even. For example, assuming the SP has not been modified since the automatic context save, the operand *–SP[6] always (and only) references the pair DP:ST1. The following instruction loads DP:ST1 into the accumulator:

```
MOVL  ACC, *–SP[6]
```

The result is DP in the high word of ACC (AH) and ST1 in the low word of the ACC (AL). At this point, DP can be manipulated by operations on AH, and ST1 can be manipulated by operations on AL.

If you wish to replace a value on the stack, you can use the opposite form of the MOVL instruction with the same offset. For example, to write a new DP:ST1 value to the stack, use:

```
MOVL  *–SP[6], ACC
```

8.3.5 Using Nested Interrupts

An interrupt is *nested* if the interrupt request occurs and is approved by the CPU while the CPU is executing code within an interrupt service routine (ISR). Nested interrupts cannot occur unless they are nonmaskable or you enable them within an ISR. What is required to enable a nested interrupt depends on the situation:

- **Real-time emulation mode with the CPU in the debug-halt state.** The ISR belongs to a time-critical interrupt and can be interrupted only by time-critical interrupts. Thus, the approval of nested interrupts depends on the interrupt enable register (IER) and the debug interrupt enable register (DBGIER). For example, suppose the ISR for $\overline{\text{INT4}}$ is to be interrupted *only* by a requests from $\overline{\text{INT2}}$ and $\overline{\text{INT3}}$. The ISR for $\overline{\text{INT4}}$ must set the INT2 and INT3 bits (and clear all the others) in the IER and DBGIER.

  During each automatic context save, the IER is saved to the stack. Unless you change the IER value on the stack, the original value is restored to the IER if the ISR is concluded with an IRET instruction. The DBGIER value is neither saved nor restored.

- **All other situations.** The approval of a maskable interrupt depends on the IER and the interrupt global mask bit (INTM) in status register ST1. Before an ISR begins, the CPU sets INTM to globally disable maskable interrupts. First, the ISR should make sure the appropriate bits are set or cleared in the interrupt enable register (IER). For example, if the ISR for $\overline{\text{INT4}}$ is to be interrupted *only* by requests from $\overline{\text{INT2}}$ and $\overline{\text{INT3}}$, the ISR must set bits INT2 and INT3 (and clear all the others) in the IER. Then the ISR must clear INTM to globally reenable maskable interrupts.

  During each automatic context save, the IER is saved to the stack; the INTM bit is also saved, as part of ST1. Unless you change the IER or INTM values on the stack, the original values are restored if you conclude the ISR with an IRET instruction.

The code segments in Example 8–2 illustrate how to enable nested interrupts that depend on the IER and the INTM bit. In this example, the ISR for $\overline{\text{INT4}}$ can be interrupted by requests from $\overline{\text{INT2}}$ or $\overline{\text{INT3}}$; the ISR for $\overline{\text{INT2}}$ can be interrupted by $\overline{\text{INT3}}$; and the ISR for $\overline{\text{INT3}}$ is not interruptible.

Example 8–2. Enabling and Disabling Nested Interrupts

Interrupt vectors defined (in interrupt vector table):

```
INT2      .long  Int2_ISR         ; Vector for INT2
INT3      .long  Int3_ISR         ; Vector for INT3
INT4      .long  Int4_ISR         ; Vector for INT4
```

Masks defined:

```
M_INT2_3  .equ   0110b            ; Binary mask to enable INT3 & INT2 in IER
M_INT3    .equ   0100b            ; Binary mask to enable INT3 in IER
```

Interrupt service routines:

```
Int4_ISR:
          IACK   #Int4_ISR        ; Send interrupt acknowledge.
          AND    IER, #0b         ; Disable all maskable interrupts.
          OR     IER, #M_INT2_3   ; Enable INT2 and INT3.
          CLRC   INTM             ; Globally enable interrupts.
          .
          .                       ; Perform function for INT4.
          .
          IRET                    ; Return from interrupt.

Int2_ISR:
          IACK   #Int2_ISR        ; Send interrupt acknowledge.
          AND    IER, #0b         ; Disable all maskable interrupts.
          OR     IER, #M_INT3     ; Enable INT3.
          CLRC   INTM             ; Globally enable interrupts.
          .
          .                       ; Perform function for INT2.
          .
          IRET                    ; Return from interrupt.

Int3_ISR:
          IACK   #Int3_ISR        ; Send interrupt acknowledge.
          AND    IER, #0b         ; Disable all maskable interrupts.
          .
          .                       ; Perform function for INT3.
          .
          IRET                    ; Return from interrupt.
```

8.3.6 Checking Interrupt Flags

As described in section 3.3 (page 3-5), the maskable interrupts $\overline{\text{INT1}}$–$\overline{\text{INT14}}$, DLOGINT, and RTOSINT each have a flag bit in the interrupt flag register (IFR). When one of these interrupts is recognized at its pin, the corresponding flag bit is set. Once set, this bit is latched until the CPU begins to process the interrupt or until the bit is cleared by an instruction. Thus, even if the interrupt is not properly enabled and will not be serviced, the flag is a record that the interrupt occurred. If you want your program to respond to an interrupt request but do not need or desire an interrupt service routine, you can have your program check the IFR flag and act accordingly.

To check the IFR, first use the PUSH IFR instruction to copy its contents to the stack. Then use a subsequent instruction to test the stack value. If you are using this technique for a single interrupt, you can use the test bit (TBIT) instruction for the test. This instruction effectively copies the tested bit to the test/control flag bit (TC) in status register ST0 . In the following example, the interrupt $\overline{\text{INT1}}$ is monitored, and the program loops until $\overline{\text{INT1}}$ occurs. The TBIT instruction uses PAGE0 stack addressing mode to look at the IFR value on the stack. The loop is implemented with the short branch (SB) instruction.

```
Loop   PUSH   IFR          ; Copy IFR to memory location
                          ; given by stack pointer (SP).
                          ; Increment SP by 1.
       TBIT   *–SP[1], #0  ; Test bit 0 of IFR value.
       SB     Loop, NTC    ; If TC=INT1 flag=0, check again.
```

If you want to monitor multiple interrupts, you can use the compare (CMP) instruction to compare the IFR value to a 16-bit constant. In the following example, the IFR value is pushed to the stack and compared to the constant 0011_{16}. An IFR value of 0011_{16} indicates that the flag bits for INT5 and INT1 have been set. If the program finds IFR = 0011_{16}, it branches to a subroutine called Routine3.

```
PUSH   IFR                ; Copy IFR to stack.
                          ; Increment SP by 1.
CMP    *–SP[1], #0011h    ; IFR = 0011h?
B      Routine3, EQ       ; If equal, branch to Routine3.
```

8.4 Using the LOOPZ and LOOPNZ Instructions

You can use the LOOPZ (loop while zero) instruction or the LOOPNZ (loop while not zero) instruction to stall a program until a specified bit pattern appears in a register or in data memory. The instruction allows you to specify a location to monitor and a mask value to compare against the value at that location. The comparison is a bitwise AND operation. The syntaxes for the loop instructions follow:

LOOPNZ *loc*, #*16BitMask*
LOOPZ *loc*, #*16BitMask*

loc can reference a 16-bit CPU register or a 16-bit data-space location. *16BitMask* is a 16-bit constant to be compared against the content of the specified location. Chapter 6, *Assembly Language Instructions*, contains more details about LOOPNZ (see page 6-119) and LOOPZ (see page 6-122).

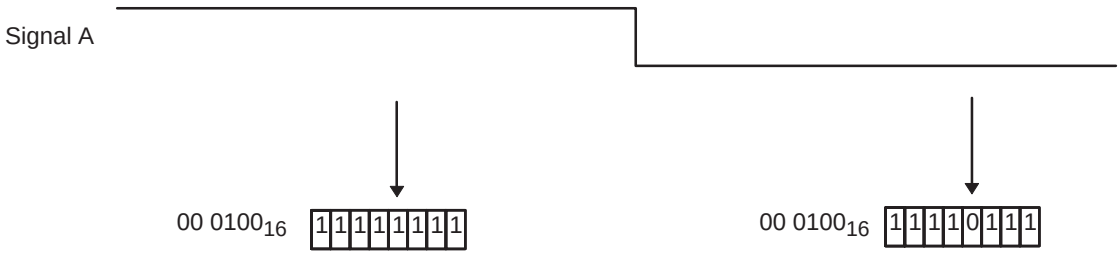
To check the same location during each execution of the loop instruction, make *loc* an operand that does not modify a pointer (for example, *AR2 or @5). To check a different location during each execution, choose an operand that modifies a pointer (for example, *SP–– or *AR2++). Using the operand for circular addressing mode (*AR6%++) enables you to repeatedly monitor the locations in a circular buffer.

8.4.1 Examples of the Loop Instructions

Consider Example 8–3, in which a single location is monitored by the LOOPNZ (loop while not zero) instruction. Signal A, external to the ’27x, is directly mapped to bit 3 at address 00 0100_{16} in data memory. When Signal A is high, bit 3 is 1; when Signal A is low, bit 3 is 0. While Signal A is high, bit 3 of the data-memory value and bit 3 of the mask value are both 1. The LOOPNZ instruction gets a nonzero result from its AND operation and, thus, is executed again. When Signal A goes low, the compared bits no longer match. The next time the LOOPNZ instruction performs its AND operation, the result is 0, and the CPU proceeds to the instruction after LOOPNZ.

Example 8–3. LOOPNZ Monitoring External Signal

```
MOVW   DP, #4        ; Select data page 4 (addresses 100h–1FFh).
LOOPNZ @0, #1000b    ; Monitor 100 until bit 3 is 0.
```



In Example 8–4, the LOOPZ (loop while zero) instruction monitors a set of consecutive addresses. Each time the instruction executes, it increments the pointer, auxiliary register AR1. Each data-memory value is checked for an odd value (bit 0 is 1). After several iterations, the instruction finds an odd value at address 00 0103$_{16}$. The compared bits match, the LOOPZ instructions gets a nonzero result, and the CPU proceeds to the instruction after LOOPZ.

Example 8–4. LOOPZ With Indirect Addressing

```
MOV    AR1, #100h       ; Load AR1 with 100h
LOOPZ  *AR1++, #0001h   ; Loop until bit 0 is 1
```

Number of times LOOPZ executed	AR1	Memory	Mask
1	0100$_{16}$	8	0001$_{16}$
2	0101$_{16}$	6	0001$_{16}$
3	0102$_{16}$	2	0001$_{16}$
4	0103$_{16}$	5	0001$_{16}$

8.4.2 Preventing an Endless Loop

Until a LOOPNZ or LOOPZ instruction finds the bit pattern it is seeking, it repeats. A loop instruction could stall a program indefinitely if the bit pattern never appears. However, either loop instruction can be interrupted by a hardware interrupt, and you can use this feature to prevent an endless loop.

If an interrupt is serviced during an active loop instruction, the return address that is saved on the stack points back to the loop instruction. This forces CPU to continue with the loop after the interrupt. However, you can create an ISR that determines when a loop instruction has run too long and increments the return address. Then, when the CPU returns from the interrupt, it branches to the new return address and executes the instruction after the loop instruction.

Example 8–5 uses a timer interrupt to keep watch over a LOOPZ instruction. The timer is set to cause an interrupt after N clock cycles. The LOOPZ instruction begins monitoring address 00 0100_{16} for an odd value (bit 0 is 1). After N cycles, the timer causes an interrupt. If the LOOPZ instruction is still active, the timer's ISR increments the return address on the stack by 2 (to skip over the two words of the LOOPZ instruction).

Example 8–5. Using a Timer to Prevent Endless Looping of a Loop Instruction

```
; Initialize timer interrupt and timer. Start timer
          .
          .
          .
; Perform conditional loop
          MOV    AR1, #100h    ; AR1 points to hexadecimal address 100.
          LOOPZ  *AR1, #01h    ; Monitor address until its bit 0 is 1.
          Next instruction
          .
          .
          .
Timer_ISR:
          MOVL   ACC, *–SP[6]  ; Load saved ST1 to accumulator.
          TBIT   AL, 5         ; Is saved LOOP bit set?
          B      Return, NTC   ; If No, then return because LOOPZ is done.
          MOVL   ACC, *–SP[2]  ; If Yes, load return address to accumulator,
          ADD    ACC, #2       ;  increment return address to end LOOPZ, and
          MOVL   *–SP[2], ACC  ;  place incremented return address on stack.
Return:   IRET                 ; Return from interrupt.
```

8.5 Managing Memory

This section introduces useful memory-management techniques:

- ❏ Moving blocks of data (see section 8.5.1)
- ❏ Accessing individual bytes (see section 8.5.2)
- ❏ Accessing an array with a base pointer and an index (see section 8.5.3)
- ❏ Aligning long data to even addresses (see section 8.5.4)
- ❏ Allocating/deallocating temporary space on the stack (see section 8.5.5)
- ❏ Safely mixing 16- and 32-bit values on the stack (8.5.6)

8.5.1 Moving Blocks of Data

The fastest way to move large blocks of data is to use the repeat instruction (RPT) to repeat a data-move instruction. The source and destination addresses are generated once rather than for each execution, reducing the number of overhead cycles for the block transfer. In addition, once a repeat loop begins, it cannot be interrupted by any interrupt.

Moving blocks of data within data space

To transfer a block of data between two data-memory sections, you can use one of the following repeated-move operations. These operations are also recommended for blocks that are mapped to both data space and program space.

RPT *count*
|| MOV ***_ind_, *(0:**_16bit_**)**

RPT *count*
|| MOV *(0:_16bit_**), ***_ind_

The operands are:

- ❏ *count.* This value is the number of times to repeat the MOV instruction after the first execution. You can specify a constant by using immediate addressing mode (for example, #3) or reference a register or data-memory value with another addressing mode (for example, @AR1).

- ❏ *****_ind._ This operand specifies a source address anywhere in data space. To access a series of consecutive data-space locations, the operand must use an indirect addressing mode and must increment or decrement a pointer. If the operand specifies an increment (for example, *AR4++), the CPU increments the pointer by 1 at the *end* of each execution. If the operand specifies a decrement (for example, *––XAR6), the CPU decrements the pointer at the *beginning* of each operation.

- ***(0:***16bit***).** This operand specifies an address within the first 64K of addresses in data space. For *16bit*, you specify the 16 LSBs of the address. The six LSBs are 0s. During each repetition, the CPU increments the address by 1 so that a series of consecutive data-space values are accessed.

Consider the following example. It transfers three words from addresses 00 0080_{16}–00 0082_{16} to addresses 0F 0000_{16}–0F 0002_{16}.

```
MOV    XAR6,#0F0000h      ; XAR6 holds destination address.
RPT    #2                 ; Do MOV 3 times (repeat twice).
|| MOV   *XAR6++,*(0:80h) ; Move data value.
```

Moving blocks of data between program space and data space

To transfer a block that is mapped only to program space to new locations in data space, repeat the program read (PREAD) instruction:

RPT *count*
|| PREAD * *ind*, ***XAR7**

To transfer a block of from data space to locations that are mapped only to program space, repeat the program read (PWRITE) instruction:

RPT *count*
|| PWRITE *XAR7, * *ind*

The operands are:

- *count.* This value is the number of times to repeat the MOV instruction after the first execution. You can specify a constant by using immediate addressing mode (for example, #3) or reference a register or data-memory value with another addressing mode (for example, @AR1).

- ***** *ind.* This operand specifies a source address anywhere in data space. To access a series of consecutive data-space locations, the operand must use an indirect addressing mode and must increment or decrement a pointer. If the operand specifies an increment (for example, *AR4++), the CPU increments the pointer by 1 at the *end* of each execution. If the operand specifies a decrement (for example, *––XAR6), the CPU decrements the pointer at the *beginning* of each operation.

- ***XAR7**

  This operand uses extended auxiliary register XAR7 as the pointer to program space. At the start of the first execution, the value in XAR7 is loaded to the fetch counter (FC). During each repetition, the CPU increments the FC by 1 so that a series of consecutive program-space values are accessed.

The following code example transfers three words from addresses 00 0040_{16}–00 0042_{16} in program space to addresses 00 0080_{16}–00 0082_{16} in data space.

```
MOV    XAR7,#40h       ; XAR7 holds program-space address.
MOV    AR2,#80h        ; AR2 holds data-space address.
RPT    #2              ; Do PREAD 3 times (repeat twice).
|| PREAD  *AR2++,*XAR7 ; Move value to data space.
```

8.5.2 Accessing Individual Bytes

The following syntaxes of the MOVB instruction operate on bytes of data within registers and memory. For more detailed information about these syntaxes, see the description for the MOVB instruction on page 6-158.

MOVB A*X***.LSB**, *loc*
MOVB *loc***, A***X***.LSB**
MOVB A*X***.MSB,** *loc*
MOVB *loc***, A***X***.MSB**

The operands are:

- **A***X***.LSB**. This is either the least significant byte of AL (AL.LSB) or the least significant byte of AH (AH.LSB).
- **A***X***.MSB**. This is either the most significant byte of AL (AL.MSB) or the most significant byte of AH (AH.MSB).
- *loc*. With this operand, you reference a register value by using register addressing mode or a memory value by using a direct or indirect addressing mode.

8.5.3 Accessing an Array With a Base Pointer and an Index

The following operands enable you to access indexed values in an array:

Operand	Description	
*+ARx[AR0] or *+XARn[AR0]	Base pointer: Index: Indexing direction: Maximum array size:	ARx or XARn AR0 Index added to base pointer 65 536
*+ARx[AR1] or *+XARn[AR1]	Base pointer: Index: Indexing direction: Maximum array size:	ARx or XARn AR1 Index added to base pointer 65 536
*+ARx[*3bit*] or *+XARn[*3bit*]	Base pointer: Index: Indexing direction: Maximum array size:	ARx or XARn *3bit* (a 3-bit constant) Index added to base pointer 8
*–SP[*6bit*]	Base pointer: Index: Indexing direction: Maximum array size:	SP *6bit* (a 6-bit constant) Index subtracted from base pointer 64

Notes: 1) ARx = AR0, AR1, AR2, AR3, AR4, or AR5

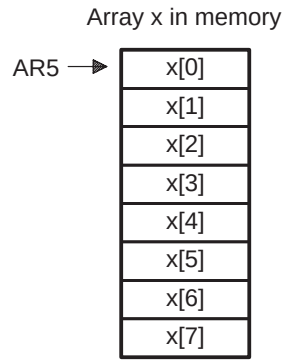
2) XARn = XAR6 or XAR7

3) To use *–SP[*6bit*], you must make sure the PAGE0 bit of status register ST1 is 0, so that PAGE0 stack addressing mode is enabled.

Example 8–6 uses auxiliary register AR5 and 3-bit index values to access elements in an 8-element array.

Example 8–6. Using 3-Bit Indexes for an 8-Element Array

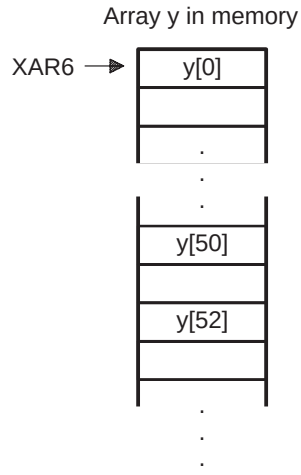
```
MOV    AR5, #x          ; AR5 points to base of x array.
MOV    AH, *+AR5[0]     ; Load x[0] to AH.
ADD    AH, *+AR5[6]     ; Add x[6] to AH.
MOV    *+AR5[7], AH     ; Store sum to x[7].
```



Example 8–7 uses auxiliary register XAR6 as the base pointer and auxiliary register AR1 as the index to access several elements in a large array. The elements are 32-bit values and have been aligned so that the low word of each element is at an even address. Therefore, only even-numbered index values are used.

Example 8–7. Using AR1 as the Index for a Large Array

```
MOV    XAR6, #y         ; AR5 points to base of y array.
MOV    AR1, #50         ; Index = 50
MOVL   ACC, *+XAR6[AR1] ; Load y[50] to ACC.
ADD    ACC, #4          ; Add 4 to ACC.
MOV    AR1, #52         ; Index = 52
MOVL   *+XAR6[AR1], ACC ; Store sum to y[52].
```



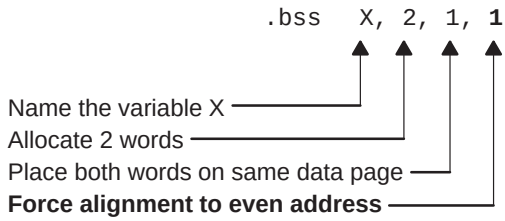
8.5.4 Aligning Long Data to Even Addresses

The ’27x core expects memory wrappers and peripheral-interface logic to align any 32-bit data read or write to an even address. If the address-generation logic generates an odd address, the memory wrapper or peripheral-interface must begin reading or writing at the previous even address. You can use the .long and .bss assembler directives to ensure that data is aligned to even addresses.

When you need to store a 22-bit interrupt vector or a 32-bit data value to memory, use the .long directive. For example, the following directive stores 5678_{16} to an even address and 1234_{16} to the following odd address. The 32-bit constant is labeled Constant1.

```
Constant1 .long 12345678h
```

When you need to allocate space for a 32-bit variable, use the even-align flag on the .bss directive. The following sample of the .bss directive reserves an even address for the low word of the variable X and reserves the following odd address for the high word of X.



8.5.5 Allocating and Deallocating Temporary Space on the Stack

As shown in the following example, you can allocate temporary space on the stack by adding to the stack pointer (SP), and you can deallocate that space by subtracting from the SP.

```
      CALL   Abs          ; Call Abs subroutine.
      .
      .
      .
Abs   ADDB   SP, #2       ; Create 2 temporary locations.
      MOVL   *–SP[2], ACC ; Store ACC to those locations.
      MOV    ACC, P       ; Copy P into ACC.
      ABS    ACC          ; Find absolute value of P.
      MOVL   *AR4, ACC    ; Copy result to locations
                          ; referenced by AR4.
      MOVL   ACC, *–SP[2] ; Restore ACC.
      SUBB   SP, #2       ; Return SP to its original
                          ;  position.
      RET                 ; Return from subroutine.
```

8.5.6 Safely Mixing 16- and 32-Bit Values on the Stack

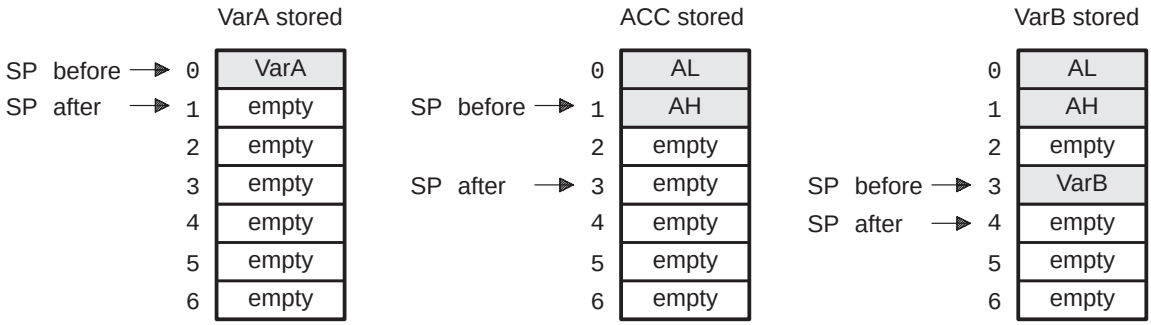
Each 32-bit read from or write to memory, including the stack, is forcibly aligned to an even address. For example, suppose your program is reading values from the stack. If the SP points to an odd address, the CPU first reads the first 16 bits from the previous even address, then reads the second 16 bits from the odd address. If you do not take the proper precautions, 32-bit stack operations could read the wrong values or corrupt values already on the stack.

Consider Example 8–8, which stores a 16-bit variable followed by a 32-bit variable. Because the SP points to an odd address before the 32-bit write operation, the 16-bit value is overwritten.

One way to avoid what happened in Example 8–8 is to always increment the stack pointer by 1 prior to one or more consecutive 32-bit operations. This technique is shown in Example 8–9 (page 8-28). Notice that in Example 8–9 there is an empty location in the middle of the stored values, regardless of whether VarA is stored to an odd address or an even address.

Example 8–8. 32-Bit Write Operation Overwriting an Existing Value

```
PUSH   @VarA         ; Store Variable A to stack.
MOVL   *SP++, ACC    ; Store ACC to stack.
PUSH   @VarB         ; Store Variable B to stack.
```



To reduce the chances of these empty locations between stack values, you can use the ASP (align stack pointer) instruction instead. If the SP is at an odd address, the instruction aligns the SP to the next even address, by adding 1 to the SP. If the SP is at an even address, the instruction does not change the SP. The ASP instruction is used in Example 8–10 (page 8-29). In this example, an empty location is still present when VarA is stored to an even address, but not when VarA is stored to an odd address. On average, using the ASP instruction instead of incrementing the SP reduces the number of empty locations.

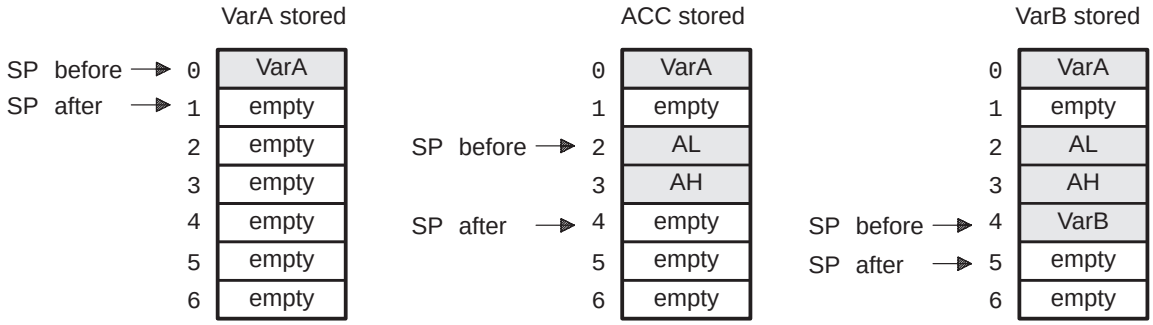
Two general rules apply to mixing 16-bit and 32-bit values on the stack:

- If you use the ASP instruction, 32-bit and 16-bit variables can be mixed as long as there is always an even number of 16-bit values between 32-bit values.
- If you do not use the ASP instruction, or if you want to be sure of no conflicts, store all 32-bit values before you store 16-bit values.

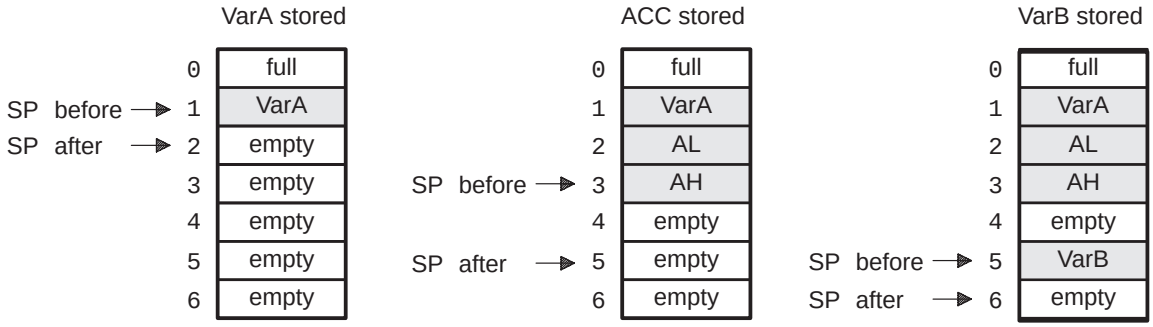
Example 8–9. Incrementing SP by 1 to Prevent Overwrite

```
PUSH   @VarA          ; Store Variable A to stack.
NOP    *SP++          ; SP = SP + 1 (to prevent overwrite)
MOVL   *SP++, ACC     ; Store ACC to stack.
PUSH   @VarB          ; Store Variable B to stack.
```

VarA stored to even address:



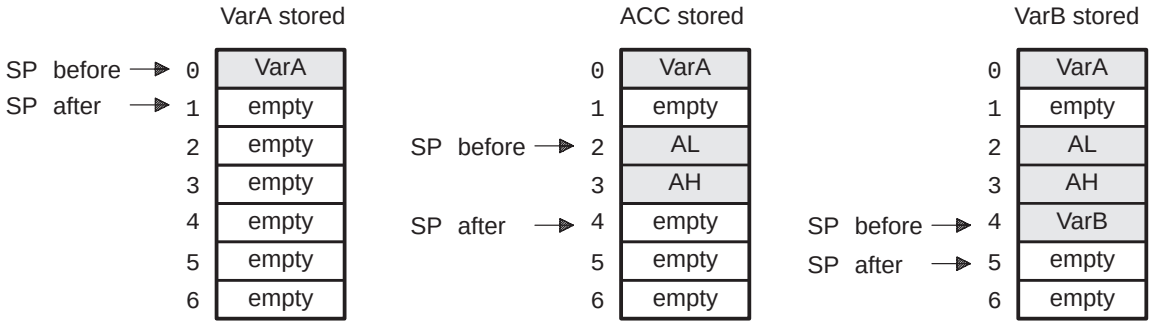
VarA stored to odd address:



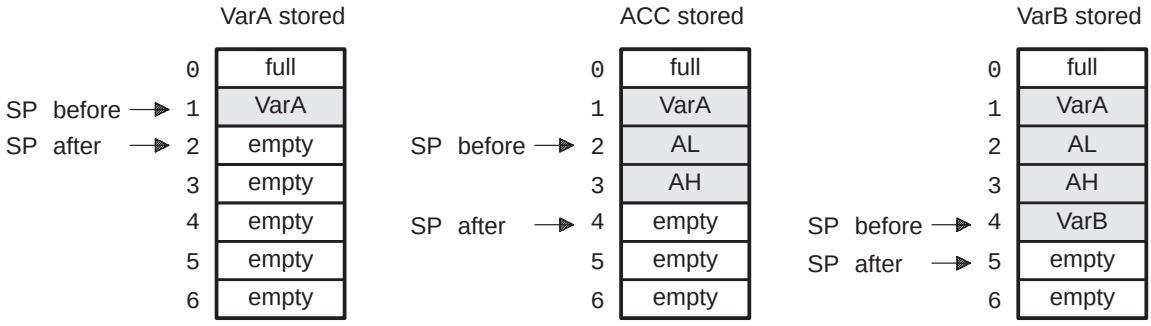
Example 8–10. Using ASP Instruction to Prevent Overwrite

```
PUSH   @VarA         ; Store Variable A to stack.
ASP                  ; Align SP to even address (to prevent overwrite)
MOVL   *SP++, ACC    ; Store ACC to stack.
PUSH   @VarB         ; Store Variable B to stack.
```

VarA stored to even address:



VarA stored to odd address:



To reverse the effect of the the ASP instruction, use the NASP (unalign stack pointer) instruction. If the ASP instruction changes the SP, it also sets the SPA bit in status register ST1. If the NASP instruction reads SPA = 1, it subtracts 1 from the SP and clears the SPA bit.

8.6 Implementing a Circular Buffer With Circular Addressing Mode

Circular buffers in memory help with algorithms such as convolution, correlation, and finite impulse response (FIR) filters. This section provides an extended example that shows how to implement a circular buffer using the ’27x circular addressing mode. This mode is introduced in section 5.5.4 on page 5-16, but important points are:

- The buffer size is determined by the eight least significant bits (LSBs) of auxiliary register AR1. Specifically, AR1 must be set to the buffer size minus 1, where the size cannot be greater than 256. For example, loading AR1 with 127 creates a buffer of size 128.

- Extended auxiliary register XAR6 is the circular pointer; it points to the current address in the buffer. This is the only circular pointer available on the ’27x.

- The top of the buffer must be aligned on a 256-word boundary, regardless of the buffer size. That is, the eight LSBs of the first address in the buffer must be 0. It is best to let the linker handle this alignment. For example, to align a section called BUFFER, you could include the following code in the linker command file:

```
SECTIONS
{
   BUFFER:   align(256) { } > RAM PAGE 1
   .  .  .
}
```

- Circular addressing for the ’27x has a single operand, *AR6%++. The operand causes a post increment of AR6, which is XAR6(15:0). AR6 is post incremented by 1 for 16-bit accesses and by 2 for 32-bit accesses.

- When the eight LSBs of AR6 match the eight LSBs of AR1, the bottom of the buffer has been reached. At this point, the eight LSBs of AR6 are cleared, so that XAR6 points again to the top of the buffer.

- If at least one of the instructions accessing your circular buffer performs a 32-bit operation, you must make sure AR6 and AR1 are both even to avoid jumping past the end of the buffer.

In Example 8–11, the objective is to repeatedly calculate a sum of products of the following form:

Output = (A × Z[m]) + (B × Z[n])

A and B are fixed values. Z is an array of two elements, Z[1] and Z[2], which collectively form the circular buffer. The circular pointer is modified so that it alternately points to Z[1] and Z[2]. The buffer receives a new value from a peripheral during each successive calculation. Each time a new value is received, it replaces the older of the two Z values. During each calculation, A is multiplied by the older Z value and B is multiplied by the newer Z value. To understand the process, consider the following hypothetical steps:

1) Load initial values for Z[1] and Z[2].

   Z[1] = 1
   Z[2] = 2

2) Transfer a new value to Z[1]:

   Z[1] = 3 (new value)
   Z[2] = 2 (old value)

3) Calculate the following sum of products:

   Output = (A × Z[2]) + (B × Z[1])
   Output = (A × 2) + (B × 3)

4) Transfer a new value to Z[2]

   Z[1] = 3 (old value)
   Z[2] = 4 (new value)

5) Calculate the following sum of products:

   Output = (A × Z[1]) + (B × Z[2])
   Output = (A × 3) + (B × 4)

6) Transfer a new value to Z[1].

   Z[1] = 5 (new value)
   Z[2] = 4 (old value)

7) Calculate the following sum of products:

   Output = (A × Z[2]) + (B × Z[1])
   Output = (A × 4) + (B × 5)

Example 8–11. Using a Circular Buffer

```
; This routine illustrates the circular addressing capability of the
; ’27x devices. It repeatedly calculates the following sum of products:
;             Output = (A x Z[m]) + (B x Z[n])
;
;             A and B are constants
;             Z[m] and Z[n] are variables in a circular buffer.
;             These variables are alternately loaded and are
;             alternately multiplied by A and B. For example:
;
;             1) Output = (A x Z[2]) + (B x Z[1])
;             2) Output = (A x Z[1]) + (B x Z[2])
;             3) Go to step 1.
;
SIZE   .set   2                    ; Buffer size = 2
Zn     .usect “BUFFER”,SIZE        ; Allocate space for buffer.
       .bss   Input,2,1            ; Allocate space for Input and Output,
                                   ; make sure they are on same data page.

Output .set   input+1              ; Define Output as the address after Input.

A      .set   1                    ; A = 1
B      .set   2                    ; B = 2

       .text
       MOV    XAR6, #Zn            ; XAR6 points to top of buffer.
       MOV    AR1, #SIZE–1         ; AR1 gets (Buffer size – 1).
       SPM    0                    ; P will not be shifted.
       SETC   SXM                  ; Turn on sign extension.
       CLRC   OVM                  ; Turn off overflow mode (no saturation).
       MOVW   DP, #Input           ; Set data page for direct addressing.

MATH:  MOV    *AR6%++,*(0:Input)   ; Load new value into buffer.
                                   ; Increment AR6: XAR6 –-> Z[old]
       MOV    P, #0                ; Clear P register.
       MPY    ACC, *AR6%++, #A     ; ACC = A x Z[old], XAR6 –-> Z[new]
       MPY    P, *AR6%++, #B       ; P = B x Z[new], XAR6 –-> Z[old]
       ADD    ACC, P               ; ACC = (A x Z[old]) + (B x Z[new])
       MOVL   @Output, ACC         ; Store output.
       B      MATH                 ; Repeat the process.
```

Note: XAR6 —> means XAR6 points to

Appendix A

Register Quick Reference

For the status and control registers of the ’27x, this appendix summarizes:

- Their reset values
- The instructions available for accessing them
- The functions of their bits

Topic **Page**

A.1 Reset Values of and Instructions for Accessing the Registers A-2

A.2 Register Figures A-3

A.1 Reset Values of and Instructions for Accessing the Registers

Table A–1 lists the CPU status and control registers, their reset values, and the instructions that are available for accessing the registers.

Table A–1. Reset Values of the Status and Control Registers

Register	Description	Reset Value	Instructions
ST0	Status register 0	$0000\,0000\,0000\,0000_2$	PUSH, POP
ST1	Status register 1	$0000\ 0000\ 0000\ V011_2$	PUSH, POP
IFR	Interrupt flag register	$0000\ 0000\ 0000\ 0000_2$	PUSH, AND, OR
IER	Interrupt enable register	$0000\ 0000\ 0000\ 0000_2$	MOV, AND, OR
DBGIER	Debug interrupt enable register	$0000\ 0000\ 0000\ 0000_2$	PUSH, POP

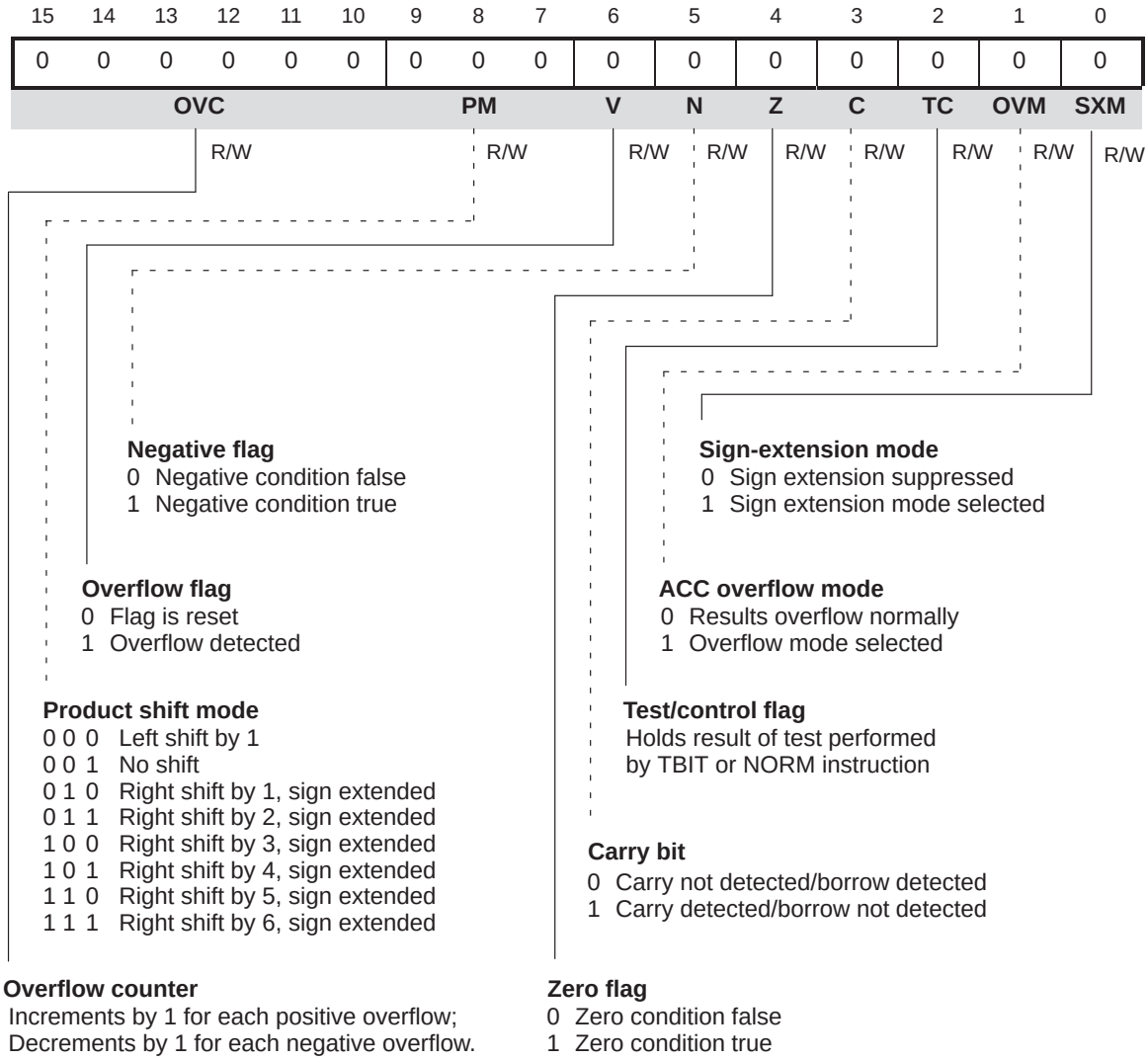
Note: V: Bit 3 of ST1 (the VMAP bit) depends on the level of the VMAP input signal at reset. If the VMAP signal is low, the VMAP bit is 0 after reset; if the VMAP signal is high, the VMAP bit is 1 after reset.

A.2 Register Figures

The following figures summarize the content of the ’27x status and control registers. Each figure in this section provides information in this way:

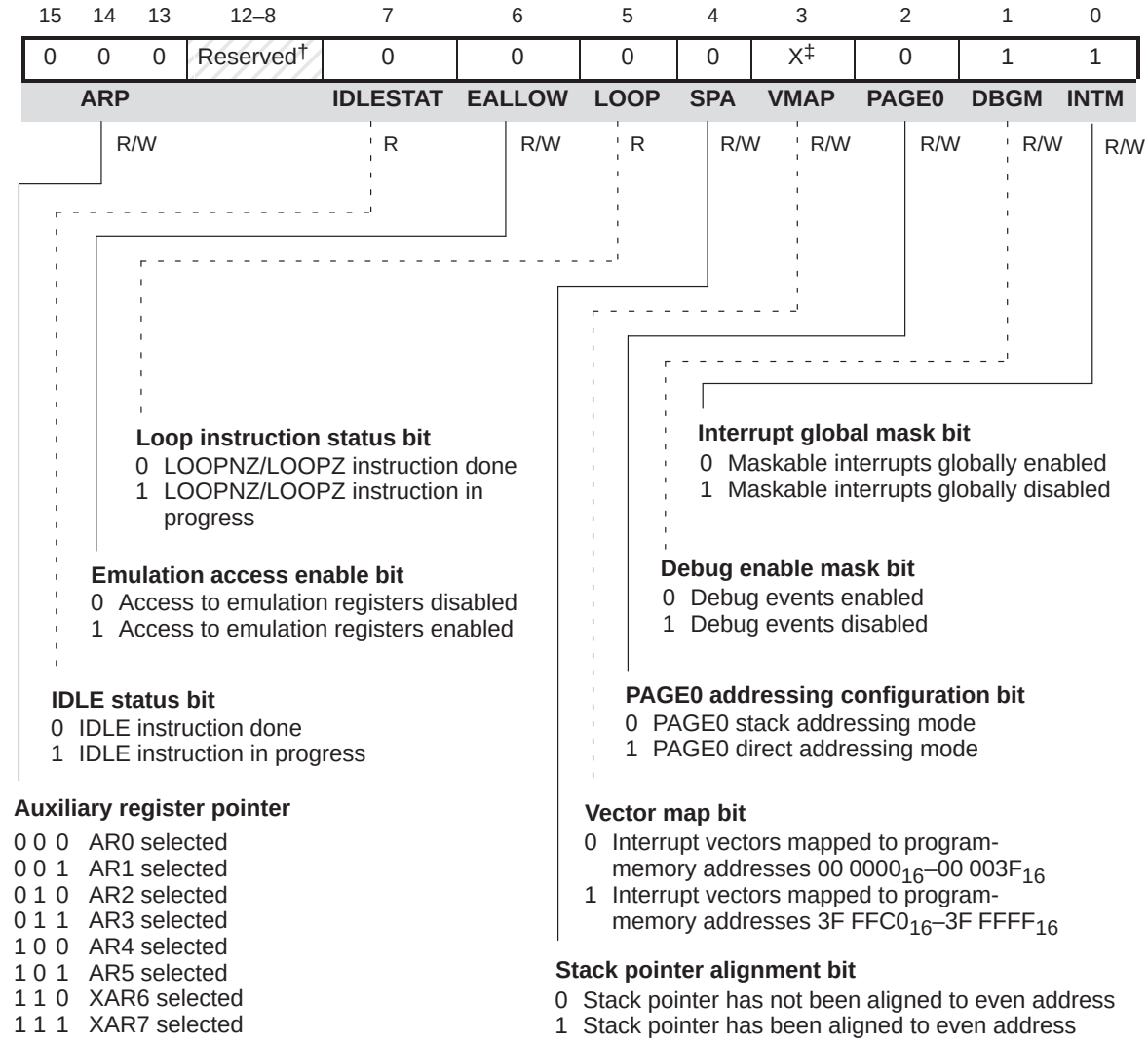
- The value shown in the register is the value after reset.
- Each unreserved bit field or set of bits has a callout that very briefly describes its effect on the processor.
- Each nonreserved bit field or set of bits is labeled with one of the following symbols:
  - R indicates that your software can read the bit field but cannot write to it.
  - R/W indicates that your software can read the bit field and write to it.
- Where needed, footnotes provide additional information for a particular figure.

Status register ST0



Note: For more details about ST0, see section 2.3 on page 2-13.

Status register ST1

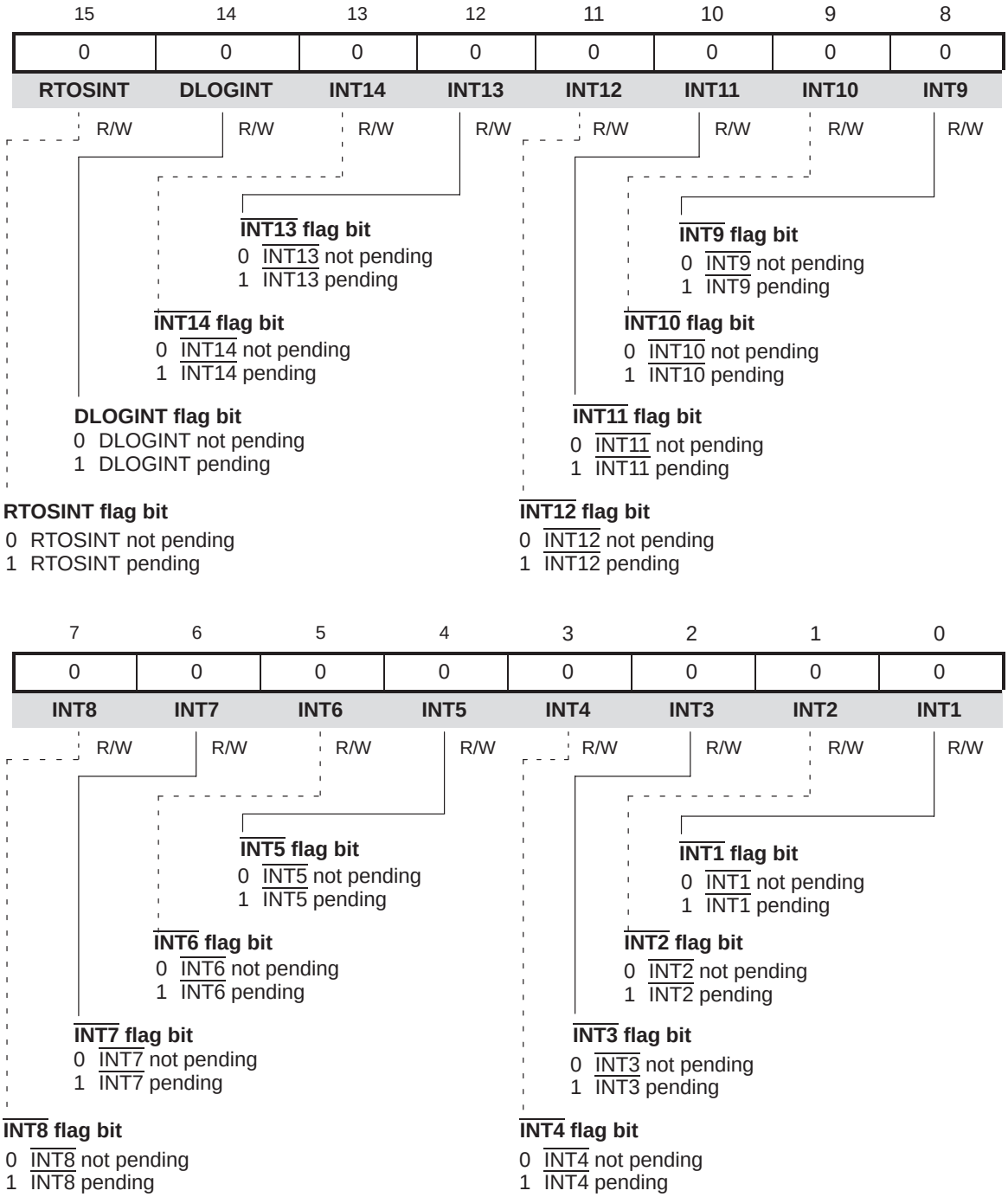


† These reserved bits are always 0s and are not affected by writes.

‡ The VMAP bit depends on the level of the VMAP input signal at reset. If the VMAP signal is low, the VMAP bit is 0 after reset; if the VMAP signal is high, the VMAP bit is 1 after reset.

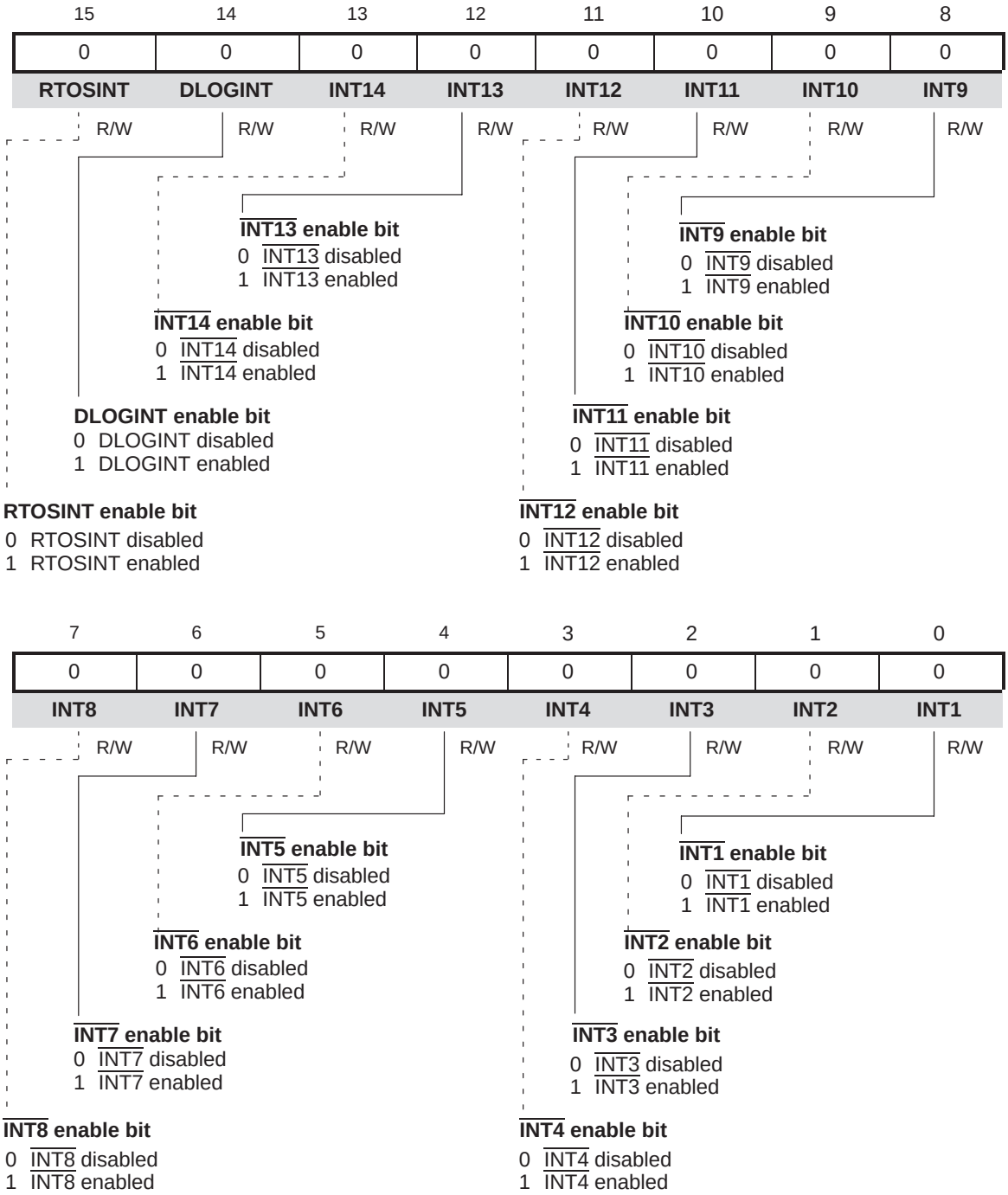
Note: For more details about ST1, see section 2.4 on page 2-19.

Interrupt flag register (IFR)



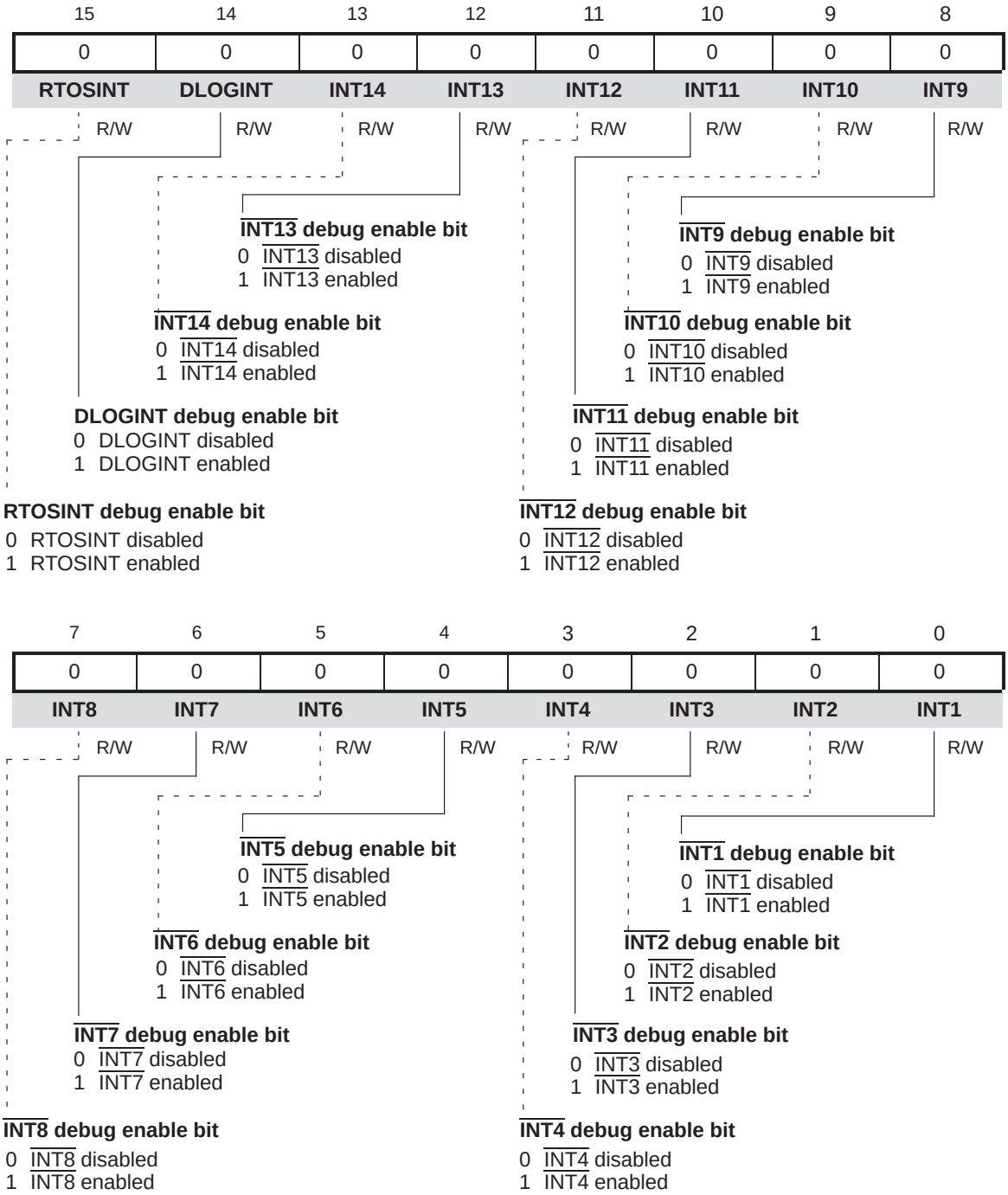
Note: For more details about the IFR, see section 3.3.1 on page 3-6.

Interrupt enable register (IER)



Note: For more details about the IER, see section 3.3.2 on page 3-7.

Debug interrupt enable register (DBGIER)



Note: For more details about the DBGIER, see section 3.3.2 on page 3-7.

Appendix B

Submitting ROM Codes to TI

This appendix defines the scope of code-customized DSPs and describes the procedures for developing prototype and production units. Information on submitting object code and on ordering customer ROM-coded devices is also included.

Topic **Page**

B.1 Scope **B-2**
B.2 Procedure **B-3**
B.3 Code Submittal **B-6**
B.4 Ordering **B-7**

B.1 Scope

A repetitive routine (for example, boot code) or an entire system algorithm can be embedded (programmed) into the on-chip ROM of a TMS320 DSP. With external memory expansion still available, this reduces the total chip count and allows for more flexibility in program design. Multiple functions are easily implemented by a single device, thus enhancing the system's capabilities. In many instances, embedded ROM code can reduce the bulk and mechanical size of the end application.

The embedded device, due to its customer-specific code, can only be offered for sale as such to that customer or the customer's formally designated representative. The customer's intellectual property (that is, his unique embedded code level) within the device is protected by a unique part number, as well as customer copyright indicated by device symbolization.

Code-customized DSP processors offer these advantages:

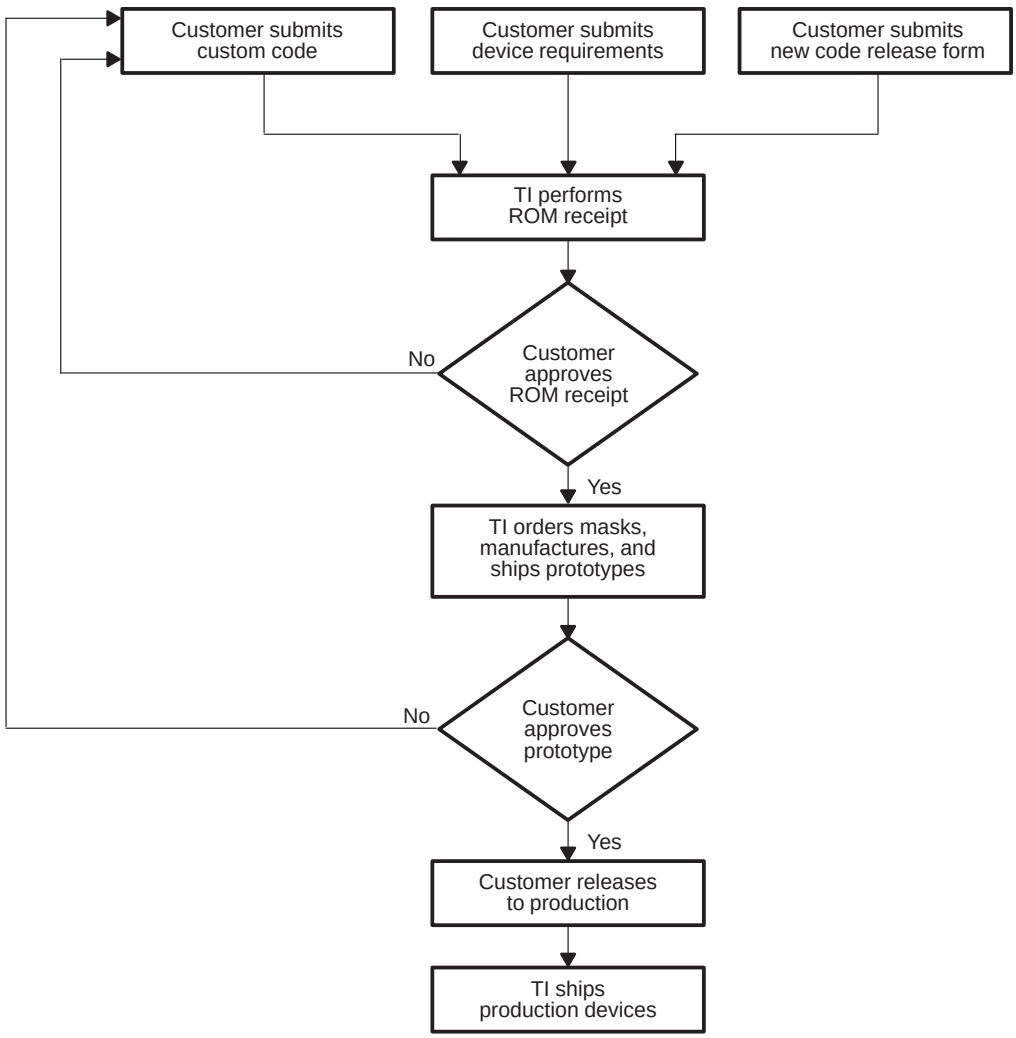
- Lower system cost for volume-driven applications
- Extended system memory expansion capability
- Reduced system hardware and wiring
- More compact/less expensive PCB
- Enhanced security for proprietary software implementations

Standard TMS320 development tools are used to develop, test, refine, and finalize the algorithms. A microprocessor/microcomputer mode pin is available on all ROM-coded TMS320 DSP devices when accessing either on-chip or off-chip memory is required. The microprocessor mode is used to develop, test, and refine a system application. In this mode of operation, the TMS320 acts as a standard microprocessor by using external memory only. When the algorithm has been finalized, you may submit the code to Texas Instruments for masking into the on-chip program ROM. At that time, the TMS320 becomes a microcomputer that executes a customized program out of the on-chip ROM. Should the code need changing or upgrading later, the TMS320 may once again be used in the microprocessor mode for development to manage the transition to the revised ROM code. This simplifies the upgrade process by allowing for a "rolling (code) change", and reduces the possibility of finished and work-in-process inventory obsolescence, while affording an orderly continuation of end-product output.

B.2 Procedure

Figure B–1 illustrates the procedural flow for TMS320 masked parts. When ordering, there is a one-time nonrefundable (NRE) charge for mask tooling and related one-time engineering costs. This charge also covers the costs for a finite number of supplied prototype units. A minimum production order per year is required for any masked-ROM device, and assurance of that order is expected at the time of NRE order acceptance.

Figure B–1. TMS320 ROM Code Prototype and Production Flowchart



B.2.1 Customer Required Information

For TI to accept the receipt of a customer ROM algorithm, each of the following three items must be received by the TI factory.

1) The customer completes and submits a New Code Release Form (NCRF—available from TI Field Sales Office) describing the custom features of the device (for example, customer information, prototype and production quantities and dates, any exceptions to standard electrical specifications, customer part numbers, and symbolization, package type, etc.).
2) If nonstandard specifications are requested on the NCRF, the customer submits a copy of the specification for the DSP in the customer's system, including functional description and electrical specification (including absolute maximum ratings, recommended operating conditions, and timing values).
3) When the customer has completed code development and has verified this code with the development system, the standard TMS320 tagged object code is submitted to the TI factory via any of the following.
   - ❑ MS-DOS™ formatted, 3.5-inch disk compatible with IBM™ PC™
   - ❑ PC-to-PC electronic transmittal (for example, via modem or Internet)

The completed NCRF, customer specification (if required), and ROM code should be given to the TI Field Sales Office or sent to:

Texas Instruments Digital Signal Processor Products
ATTN: TMS320 DSP Marketing Manager-ROM Receipt, M/S 704
P.O. Box 1443
Houston, Texas 77251–1443

B.2.2 TI Performs ROM Receipt

Code review and ROM receipt is performed on the customer's code and a unique manufacturing ROM code number (such as Dxxxxx) is assigned to the customer's algorithm. All future correspondence should indicate this number. The ROM receipt procedure reads the ROM code information, processes it, reproduces the customer's ROM object code on the same media on which it was received, and returns the processed and the original code to the customer for verification of correct ROM receipt.

B.2.3 Customer Approves ROM Receipt

The customer then verifies that the ROM code received and processed by TI is correct and that no information was misinterpreted in the transfer. The customer must then return written confirmation of correct ROM receipt verification or resubmit the code for processing. This written confirmation of verification constitutes the contractual agreement for creation of the custom mask and manufacture of ROM verification prototype units.

B.2.4 TI Orders Masks, Manufactures, and Ships Prototypes

TI generates the prototype photomasks, processes, manufactures, and tests microcomputer prototypes containing the customer's ROM pattern for shipment to the customer for ROM code verification. These devices have been made using the custom mask but are for the purposes of ROM verification only. For expediency, the prototype devices are tested only at room temperatures (25°C). **Texas Instruments recommends that prototype devices not be used in production systems.** Prototype devices are symbolized with a **P** preceding the manufacturing ROM code number (for example, PDxxxxx) to differentiate them from production devices.

B.2.5 Customer Approves Prototype

The customer verifies the operation of these prototypes in the system and responds with written customer prototype approval or disapproval. This written customer prototype approval constitutes the contractual agreement to initiate volume production using the verified prototype ROM code.

B.2.6 Customer Release to Production

With customer algorithm approval, the ROM code is released to production and TI begins shipment of production devices according to the customer's final specifications and order requirements.

Two lead times are quoted in reference to the preceding flow:

- **Prototype lead time** is the elapsed time from the receipt of written ROM receipt verification to the delivery of the prototype devices.
- **Production lead time** is the elapsed time from the receipt of written customer prototype approval to the delivery of production devices. For the latest TMS320 family lead times, contact the nearest TI Field Sales Office.

B.3 Code Submittal

The customer's object code can be submitted via 3.5-inch disk or via electronic transmittal (that is, modem, Internet, other). For 'C1x or 'C2x family codes, Intel™ Hex or TI-tagged format is required; for all other families, COFF format from the cross-assembler/linker is needed.

When a code is submitted to Texas Instruments for masking, the code is reformatted by TI to accommodate the TI mask-making and test program generation systems. Application-level verification by the customer is, therefore, necessary. Although the code has been reformatted, it is important that the changes remain transparent to the user and do not affect the execution of the algorithm submitted. Those formatting changes consist essentially of adding ease-of-manufacturing code in reserved and not used (customer) locations only. Resulting code has the code address beginning at the base address of the ROM in the TMS320 device and progressing without gaps to the last address of the ROM on the TMS320 device. Note that because these changes have been made, a checksum comparison is not a valid means of verification. Upon satisfactory verification of the TI returned code, the customer advises TI in writing that it is verified, and this enables release to manufacturing and acceptance of initial orders.

B.4 Ordering

Customer embedded-code devices are user-specified, and thus, each is an unreleased new product until prototype approval and formal release to production. With each initial order of a ROM-coded device, the customer must include written recognition that he understands the following:

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 reliability of these prototype units cannot be defined.

Sometimes to shorten time to market and upon mutual agreement, the customer may order (and TI will accept) a Risk Production order prior to prototype approval. Under this noncancellable order arrangement, the customer agrees to accept delivery of product containing his code as initially verified and TI agrees to ship to that requirement. The customer is, in effect, agreeing to not change the originally submitted code for the Risk Production order units. He must use the term “Risk Production” in a letter or in a note on the order as a matter of record.

TI does reserve the right to sell excess customer ROM-coded devices as standards to reduce the financial liability incurred through premature ordered quantity reductions or overbuilds. Units thus marketed by TI have all original customer custom symbols or other means of external identification, removed and replaced by a standard product symbol to mask the custom die presence. It is standard practice to require a one-time statement from the customer stating that the customer knows and concurs.

Your local TI Field Sales Office and/or TI Authorized Distributor can be of further assistance on embedded ROM procedure questions and in actually processing your code.

Appendix C

Design Considerations for Using XDS510 Emulator

This appendix assists you in meeting the design requirements of the Texas Instruments XDS510™ emulator with respect to 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 appendix, 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
(732) 981–0060 outside the US and Canada
FAX: (732) 981–9667 Telex: 833233
WWW: http://www.ieee.org
E-mail: customer.service@ieee.org

Topic **Page**

C.1 Designing Your Target System's Emulator Connector (14-Pin Header) C-2
C.2 Bus Protocol C-4
C.3 Emulator Cable Pod C-5
C.4 Emulator Cable Pod Signal Timing C-6
C.5 Emulation Timing Calculations C-7
C.6 Connections Between the Emulator and the Target System C-10
C.7 Physical Dimensions for the 14-Pin Emulator Connector C-14
C.8 Emulation Design Considerations C-16

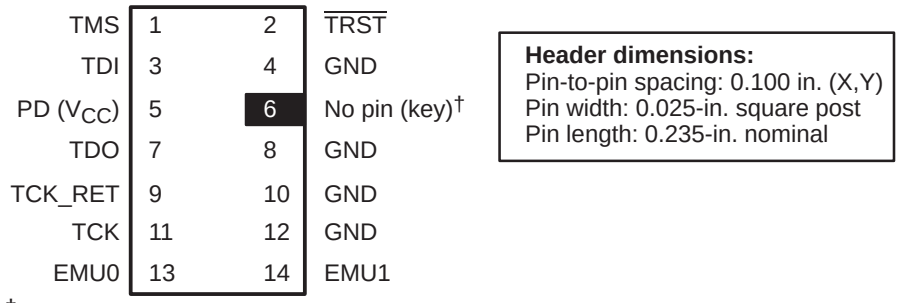
C.1 Designing Your Target System's Emulator Connector (14-Pin Header)

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 should 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. Can 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.

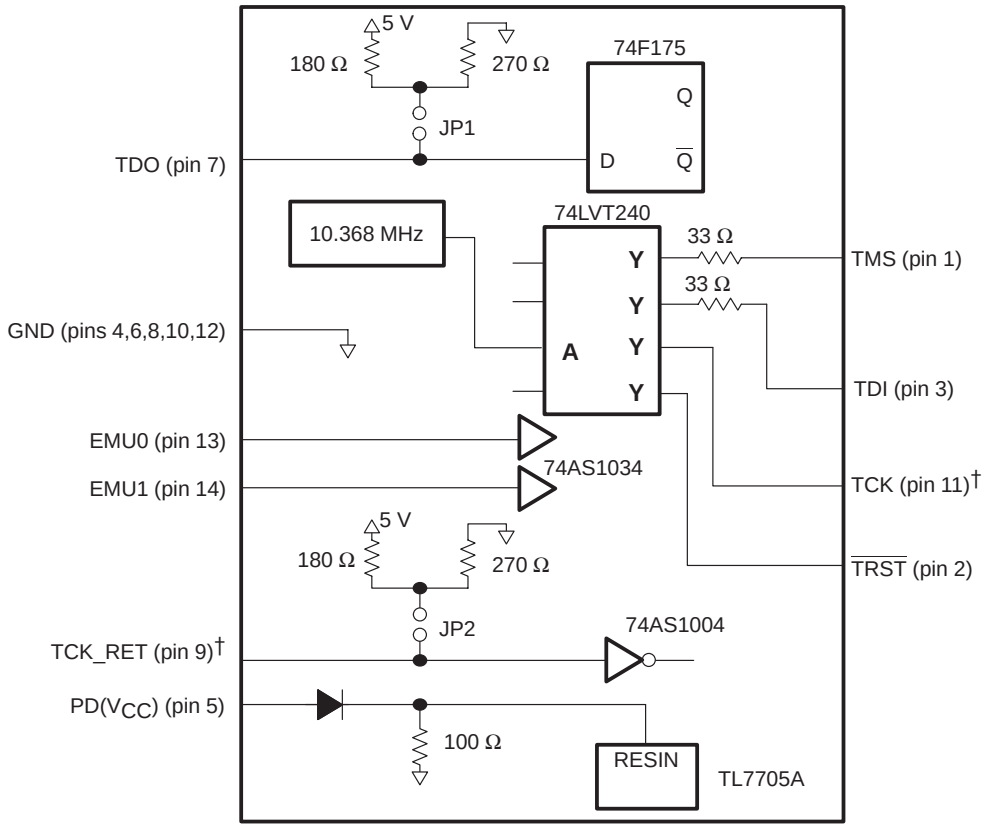
The IEEE 1149.1 specification does not provide rules for bus master (emulator) devices. Instead, it states that the device expects a bus master to provide bus slave compatible timings. The XDS510 provides timings that meet the bus slave rules.

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

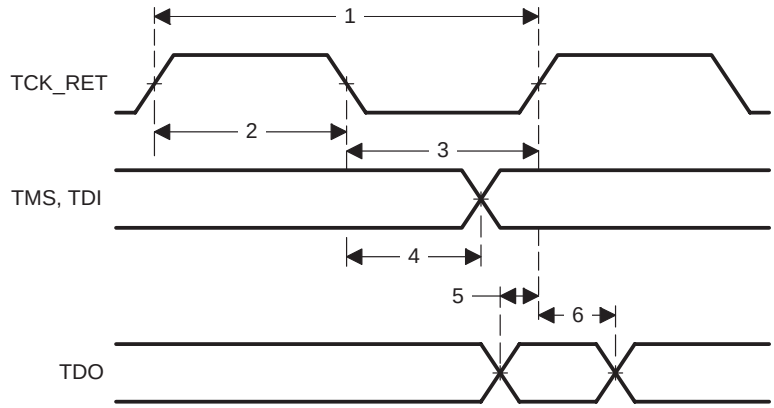


Table C–2. Emulator Cable Pod Timing Parameters

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

Example C–1 and Example C–2 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

$$t_{pd\,(TCK\_RET\text{-}TMS/TDI)} = \frac{\left[t_{d\,(TMSmax)} + t_{su\,(TTMS)}\right]}{t_{TCKfactor}}$$

$$= \frac{(20\ ns + 10\ ns)}{0.4}$$

$$= 75\ ns,\ or\ 13.3\ MHz$$

$$t_{pd\,(TCK\_RET–TDO)} = \frac{\left[t_{d\,(TTDO)} + t_{su\,(TDOmin)}\right]}{t_{TCKfactor}}$$

$$= \frac{(15\ ns + 3\ ns)}{0.4}$$

$$= 45\ ns,\ or\ 22.2\ MHz$$

In this case, because the TCK_RET-to-TMS/TDI path requires more time to complete, it is the limiting factor.

Example C–2. Key Timing for a Single- or Multiple-Processor System With Buffered Input and Output

$$t_{pd\,(TCK\_RET\text{-}TMS/TDI)} = \frac{\left[t_{d\,(TMSmax)} + t_{su\,(TTMS)} + t_{bufskew}\right]}{t_{TCKfactor}}$$

$$= \frac{(20\ ns + 10\ ns + 1.35\ ns)}{0.4}$$

$$= 78.4\ ns,\ or\ 12.7\ MHz$$

$$t_{pd\,(TCK\_RET–TDO)} = \frac{\left[t_{d\,(TTDO)} + t_{su\,(TDOmin)} + t_{d\,(bufmax)}\right]}{t_{TCKfactor}}$$

$$= \frac{(15\ ns + 3\ ns + 10\ ns)}{0.4}$$

$$= 70\ ns,\ or\ 14.3\ MHz$$

In this case also, because the TCK_RET-to-TMS/TDI path requires more time to complete, it is the limiting factor.

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:

$$
\begin{aligned}
t_r &= 5(R_{pullup} \times N_{devices} \times C_{load\_per\_device}) \\
&= 5(4.7\ k\Omega \times 16 \times 15\ pF) \\
&= 5(4.7 \times 10^{3}\ \Omega \times 16 \times 15 \times 10^{-12}\ F) \\
&= 5(1128 \times 10^{-9}\ ) \\
&= 5.64\ \mu s
\end{aligned}
$$

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

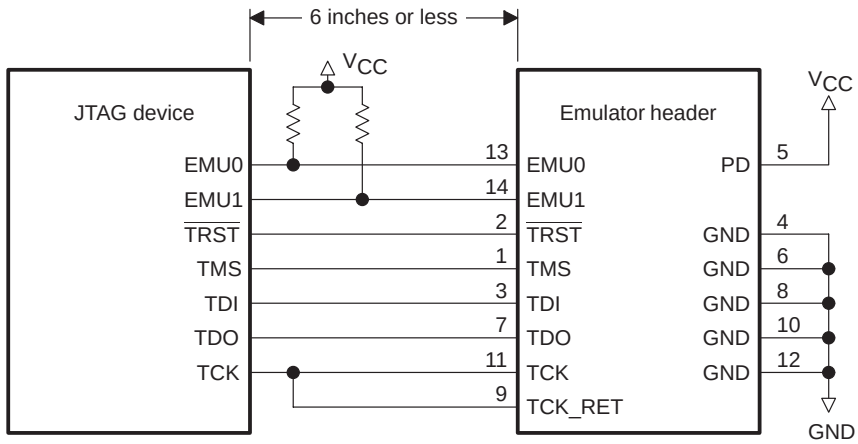
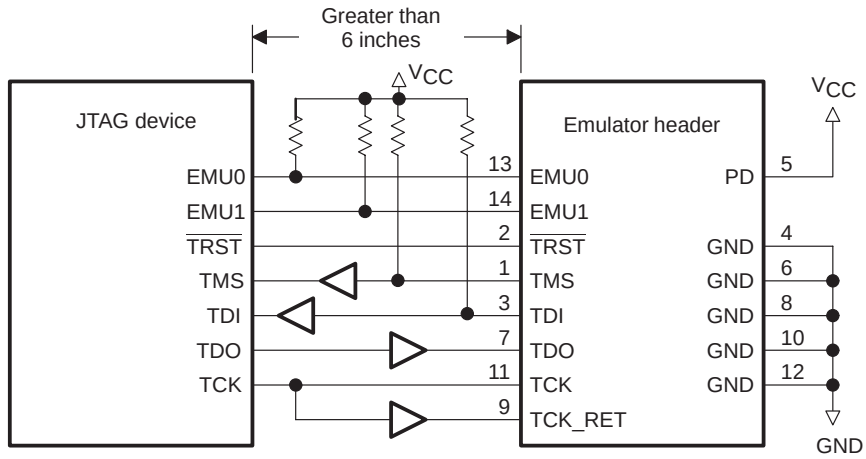


Figure C–5 shows the connections necessary for buffered transmission signals. The distance between the emulation header and the processor is greater than 6 inches. Emulation signals TMS, TDI, TDO, and TCK_RET are buffered through the same device package.

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.

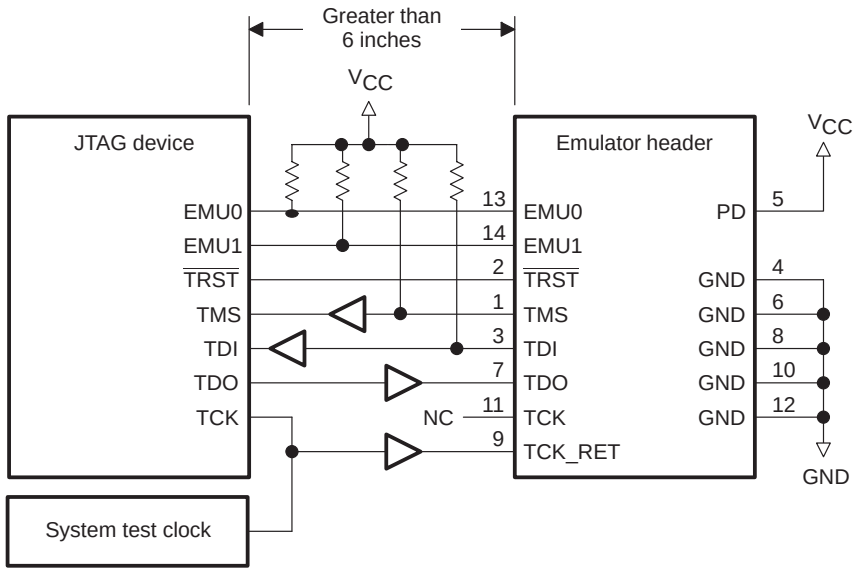
To have high-quality signals (especially the processor TCK and the emulator TCK_RET signals), you may have to employ special care when routing the printed wiring board trace. You also may have to use termination resistors to match the trace impedance. The emulator pod provides optional internal parallel 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:

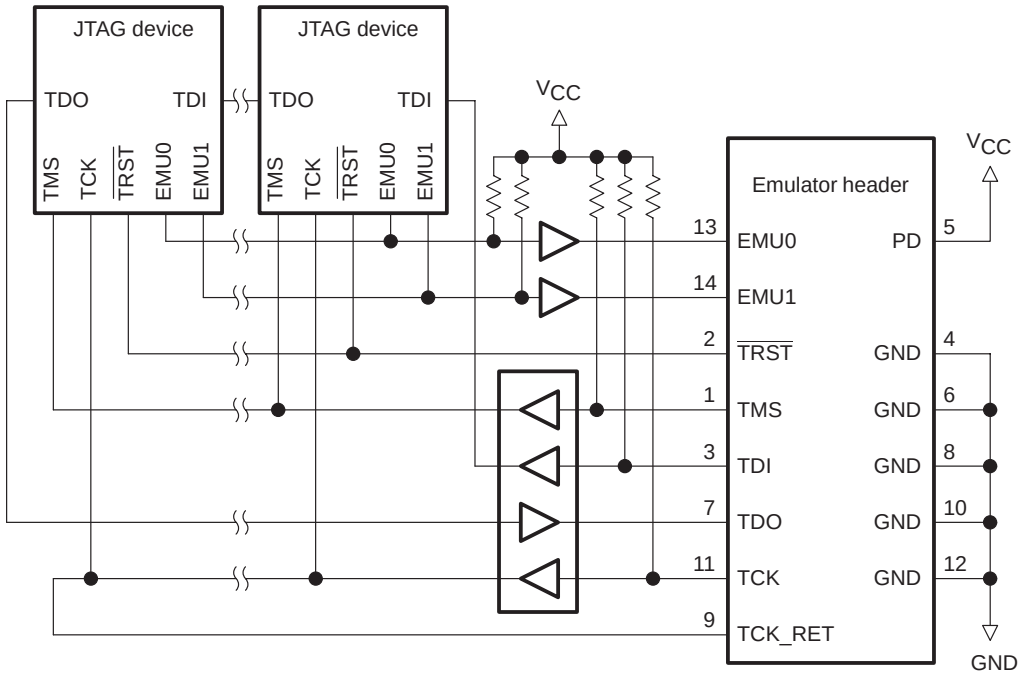
- The emulator provides only a single 10.368-MHz test clock. If you allow the target system to generate your test clock, you can set the frequency to match your system requirements.
- In some cases, you may have other devices in your system that require a test clock when the emulator is not connected. 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 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.
- ❏ 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.

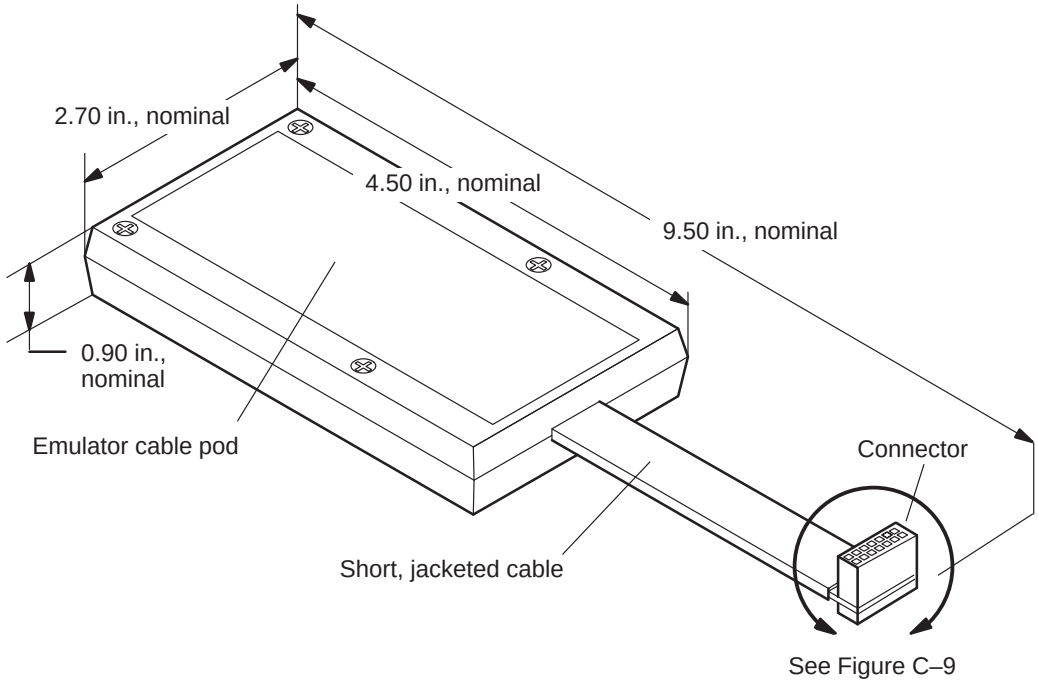
Figure C–7. Multiprocessor Connections



C.7 Physical Dimensions for the 14-Pin Emulator Connector

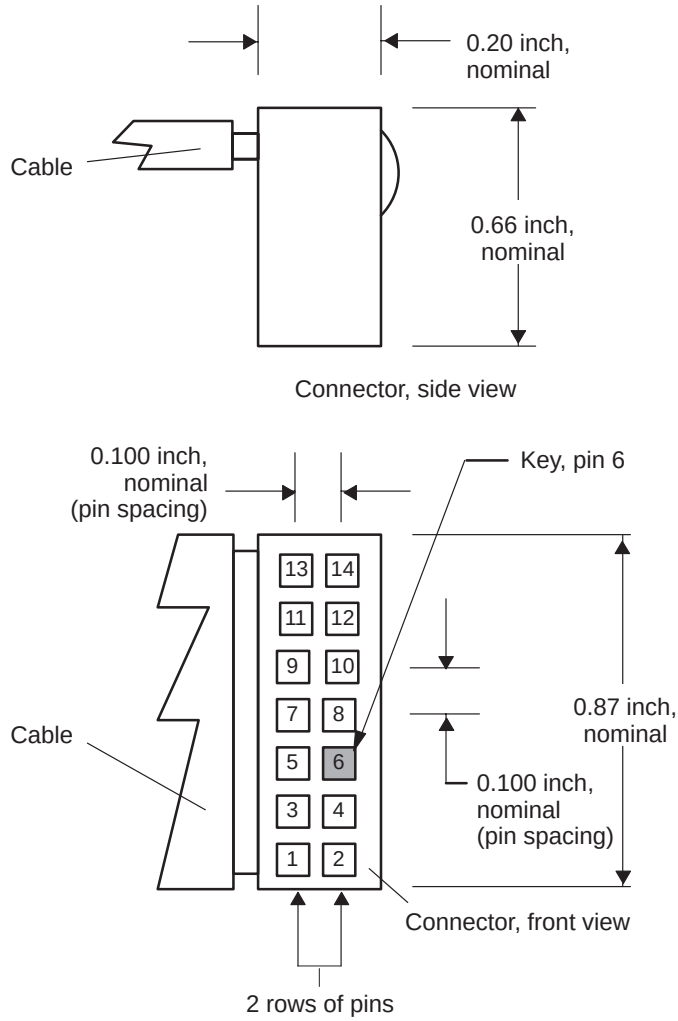
The JTAG emulator target cable consists of a 3-foot section of jacketed cable that connects to the emulator, an active cable pod, and a short section of jacketed cable that connects to the target system. The overall cable length is approximately 3 feet 10 inches. Figure C–8 and Figure C–9 (page C-15) 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



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.

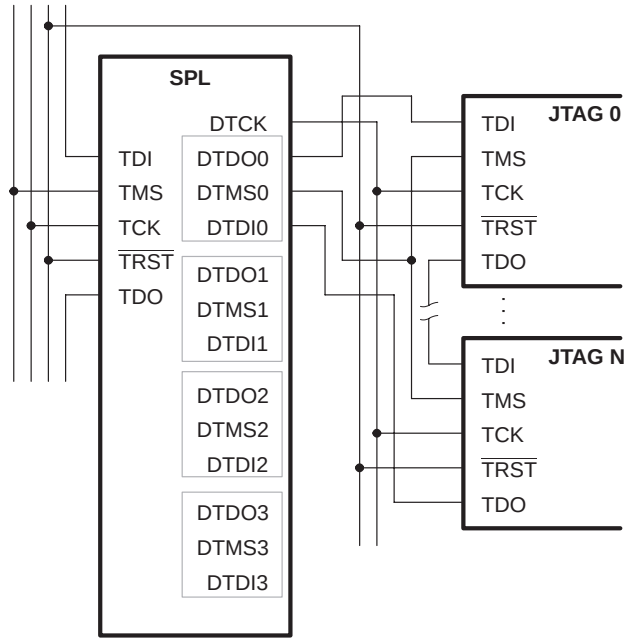
A system of multiple, secondary JTAG scan paths has better fault tolerance and isolation than a single scan path. Since an SPL has the capability of adding all secondary scan paths to the main scan path simultaneously, it can support global emulation operations, such as starting or stopping a selected 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)

Example C–3 and Example C–4 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 $t_{pd(TCK-DTDI)}$

Of the following two cases, the worst-case path delay is calculated to determine the maximum system test clock frequency.

Example C–3. 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 this case, the TCK-to-DTMS/DTDL path is the limiting factor.

Example C–4. Key Timing for a Single- or Multiprocessor-System With Buffered Input and Output (SPL)

$$t_{pd\,(TCK\text{-}TDMS)} = \frac{\left[t_{d\,(DTMSmax)} + t_{(DTCKHmin)} + t_{su\,(TTMS)} + t_{(bufskew)}\right]}{t_{TCKfactor}}$$

$$= \frac{(31\text{ ns} + 2\text{ ns} + 10\text{ ns} + 1.35\text{ ns})}{0.4}$$

$$= 110.9\text{ ns, or } 9.0\text{ MHz}$$

$$t_{pd\,(TCK\text{–}DTDI)} = \frac{\left[t_{d\,(TTDO)} + t_{d\,(DTCKLmax)} + t_{su\,(DTDLmin)} + t_{d\,(bufskew)}\right]}{t_{TCKfactor}}$$

$$= \frac{(15\text{ ns} + 15\text{ ns} + 7\text{ ns} + 10\text{ ns})}{0.4}$$

$$= 120\text{ ns, or } 8.3\text{ MHz}$$

In this case, 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:

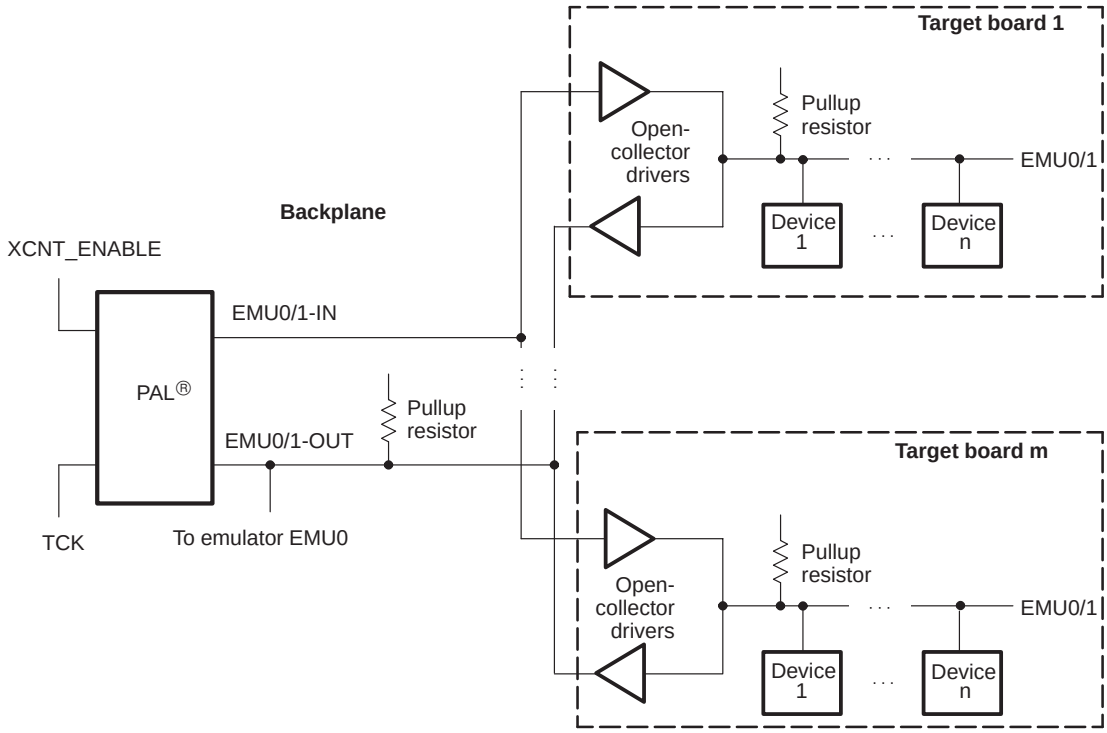
- Signal Event. The EMU0/1 pins can be configured via software to signal internal events. In this mode, driving one of these pins low can cause devices to signal such events. To enable this operation, the EMU0/1 pins function as open-collector sources. External devices such as logic analyzers can also 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 and Figure C–13 show 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 when using the EMU0/1 pins to send out analysis information.

Figure C–11. EMU0/1 Configuration to Meet Timing Requirements of Less Than 25 ns



Notes: 1) The low time on EMU0/1-IN should 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 when analysis information is being sent out.

These seven important points apply to the circuitry shown in Figure C–11 and Figure C–13 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.
- When sending out analysis information, 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. When analysis information is being sent out, 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 when analysis information is being sent out.
- 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

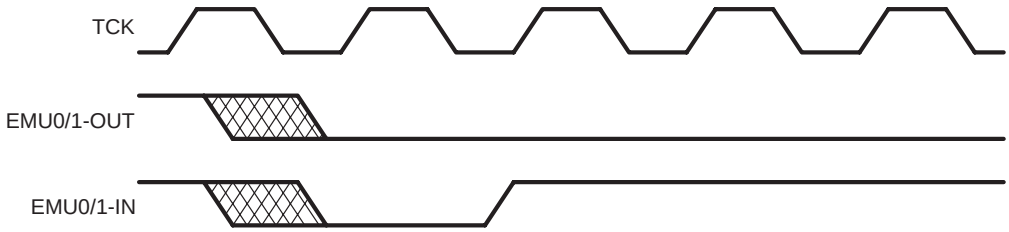
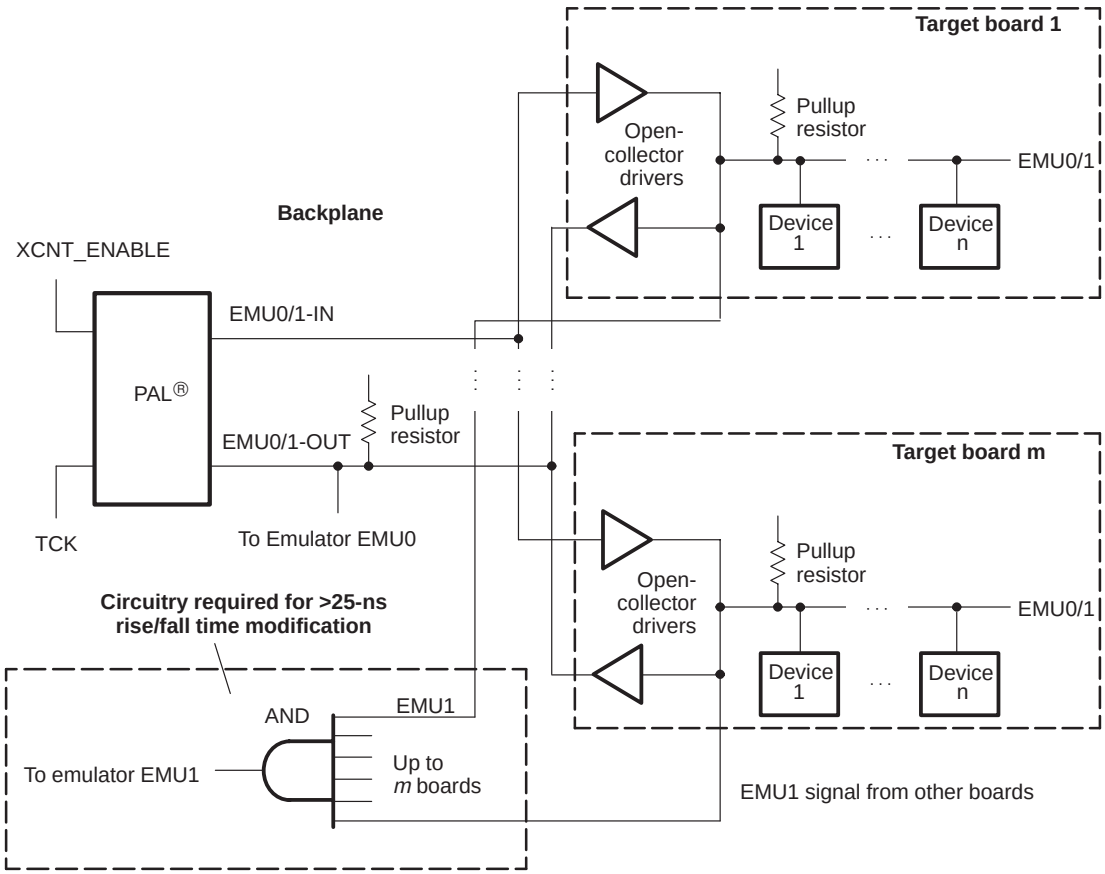


Figure C–13. EMU0/1 Configuration With Additional AND Gate to Meet Timing Requirements of Greater Than 25 ns

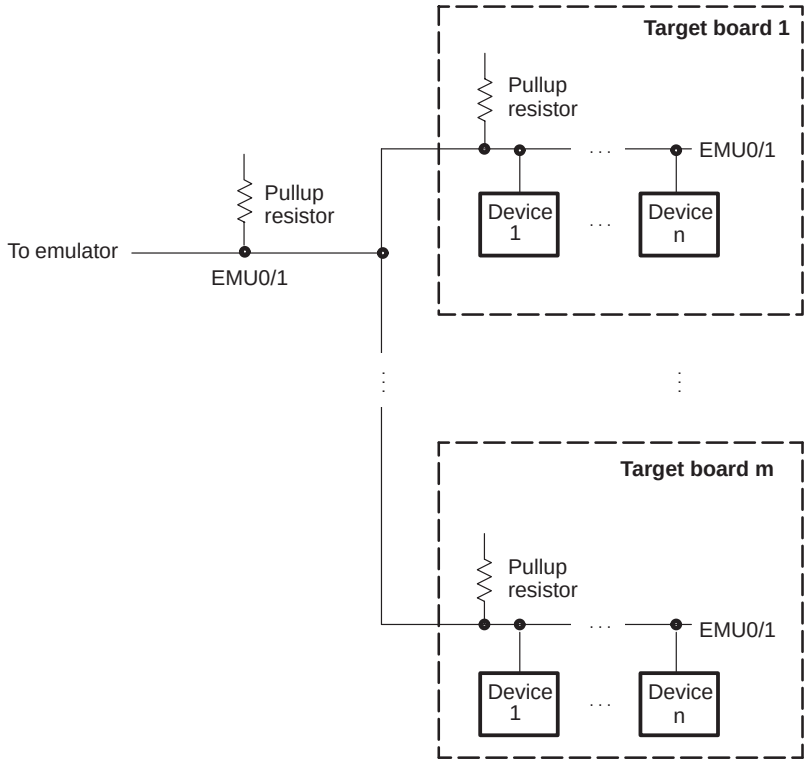


Notes: 1) The low time on EMU0/1-IN should be at least one TCK cycle and less than 10 μs. Software will set the EMU0/1-OUT port to a high state.

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 when analysis information is being sent out.

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

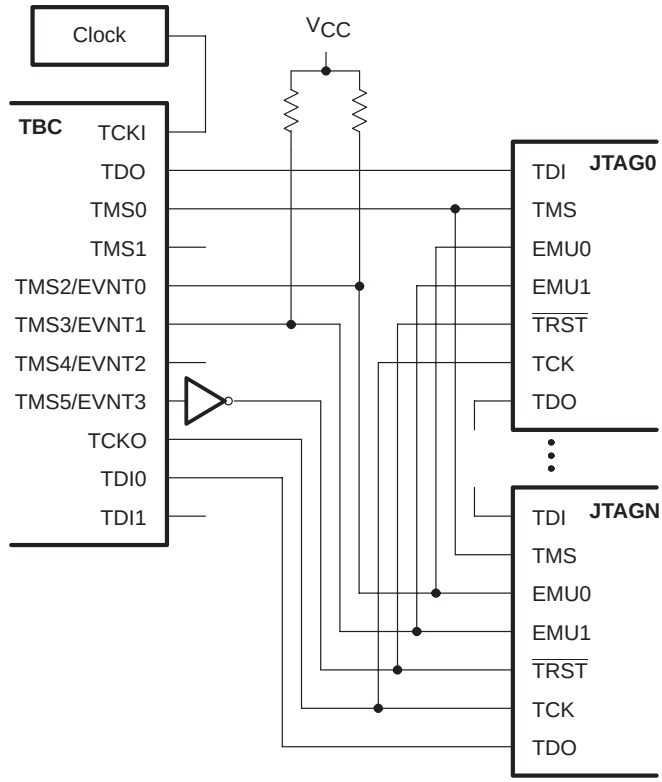


Note: The open-collector driver and pullup resistor on EMU1 must be able to provide rise/fall times of less than 25 ns. Rise times of more than 25 ns can cause the emulator to detect false edges when analysis information is being sent out. If this condition cannot be met, then the EMU0/1 signals from the individual boards must be ANDed together (as shown in Figure C–13) 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

16-bit operation: An operation that reads or writes 16 bits.

32-bit operation: An operation that reads or writes 32 bits.

A

absolute branch: A branch to an address that is permanently assigned to a memory location. See also *offset branch*.

ACC: See *accumulator (ACC)*.

access: A term used in this document to mean *read from* or *write to*. For example, to access a register is to read from or write to that register.

accumulator (ACC): A 32-bit register involved in a majority of the arithmetic and logical calculations done by the ’27x. Some instructions that affect ACC use all 32 bits of the register. Others use one of the following portions of ACC: AH (bits 31 through 16), AL (bits 15 through 0), AH.MSB (bits 31 through 24), AH.LSB (bits 23 through 16), AL.MSB (bits 15 through 8), and AL.LSB (bits 7 through 0).

address-generation logic: Hardware in the CPU that generates the addresses used to fetch instructions or data from memory.

address reach: The range of addresses beginning with $00\ 0000_{16}$ that can be used by a particular addressing mode.

address register arithmetic unit (ARAU): Hardware in the CPU that generates addresses for values that must be fetched from data memory. The ARAU is also the hardware used to increment or decrement the stack pointer (SP) and the auxiliary registers (AR0, AR1, AR2, AR3, AR4, AR5, XAR6, and XAR7).

addressing mode: The method by which an instruction interprets its operands to acquire the data and/or addresses it needs.

AH: *High word of the accumulator.* The name given to bits 31 through 16 of the accumulator.

AH.LSB: *Least significant byte of AH*. The name given to bits 23 through 16 of the accumulator.

AH.MSB: *Most significant byte of AH*. The name given to bits 31 through 24 of the accumulator.

AL: *Low word of the accumulator.* The name given to bits 15 through 0 of the accumulator.

AL.LSB: *Least significant byte of AL*. The name given to bits 7 through 0 of the accumulator.

AL.MSB: *Most significant byte of AL*. The name given to bits 15 through 8 of the accumulator.

ALU: See *arithmetic logic unit (ALU)*.

analysis logic: A portion of the emulation logic in the core. The analysis logic is responsible for managing the following debug activities: hardware breakpoints, hardware watchpoints, data logging, and benchmark/event counting.

approve an interrupt request: Allow an interrupt to be serviced. If the interrupt is maskable, the CPU approves the request only if it is properly enabled. If the interrupt is nonmaskable, the CPU approves the request immediately. See also *interrupt request* and *service an interrupt*.

AR0–AR5: *Auxiliary registers 0–5*. Six 16-bit registers used as pointers to any address within the first 64K addresses of data space. The registers are operated on by the address register arithmetic unit (ARAU) and are selected by the auxiliary register pointer (ARP). See also *AR6/AR7* and *XAR6/XAR7*.

AR6/AR7 : *Auxiliary registers 6 and 7.* AR6 is the name given to the 16 LSBs of extended auxiliary register 6 (XAR6). AR7 is the name given to the 16 LSBs of extended auxiliary register 7 (XAR7). AR6 and AR7 are available as general-purpose registers, but they are not used as pointers. See also *XAR6/XAR7*.

ARAU: See *address register arithmetic unit (ARAU)*.

arithmetic logic unit (ALU): A 32-bit hardware unit in the CPU that performs 2s-complement arithmetic and Boolean logic operations. The ALU accepts inputs from data from registers, from data memory, or from the program control logic. The ALU sends results to a register or to data memory.

arithmetic shift: A shift that treats the shifted value as signed. See also *logical shift*.

ARP: See *auxiliary register pointer (ARP)*.

ARP indirect addressing mode: The indirect addressing mode that uses the current auxiliary register to point to a location in data space. The current auxiliary register is the auxiliary register pointed to by the ARP. See also *auxiliary register pointer (ARP)*.

automatic context save: A save of system context (modes and key register values) performed by the CPU just prior to executing an interrupt service routine. See also *context save*.

auxiliary register: One of eight registers used as a pointer to a memory location. The register is operated on by the auxiliary register arithmetic unit (ARAU) and is selected by the auxiliary register pointer (ARP). See also *AR0–AR5*, *AR6/AR7*, and *XAR6/XAR7*.

auxiliary-register indirect addressing mode: The indirect addressing mode that allows you to use the name of an auxiliary register in an operand and that uses that register as a pointer. See also *ARP indirect addressing mode*.

auxiliary register pointer (ARP): A 3-bit field in status register ST1 that selects the current auxiliary register. When an instruction uses ARP indirect addressing mode, that instruction uses the current auxiliary register to point to data space. When an instruction specifies auxiliary register *n* by using auxiliary-register indirect addressing mode, the ARP is updated, so that it points to auxiliary register *n*. See also *current auxiliary register*.

B

background code: The body of code that can be halted during debugging because it is not time-critical.

barrel shifter: Hardware in the CPU that performs all left and right shifts of register or data-space values.

bit field: One or more register bits that are differentiated from other bits in the same register by a specific name and function.

bit manipulation: The testing or modifying of individual bits in a register or data-space location.

boundary scan: The use of scan registers on the border of a chip or section of logic to capture the pin states. By scanning these registers, all pin states can be transmitted through the JTAG port for analysis.

branch: 1) A forcing of program control to a new address. 2) An instruction that forces program control to a new address but neither saves a return address (like a call) nor restores a return address (like a return).

break event: A debug event that causes the CPU to enter the debug-halt state.

breakpoint: A place in a routine specified by a breakpoint instruction or hardware breakpoint, where the execution of the routine is to be halted and the debug-halt state entered.

C

C bit: See *carry (C) bit*.

call: 1) The operation of saving a return address and then forcing program control to a new address. 2) An instruction that performs such an operation. See also *return*.

carry (C) bit: A bit in status register ST0 that reflects whether an addition has generated a carry or a subtraction has generated a borrow.

circular addressing mode: The indirect addressing mode that can be used to implement a circular buffer.

circular buffer: A block of addresses referenced by a pointer using circular addressing mode, so that each time the pointer reaches the bottom of the block, the pointer is modified to point back to the top of the block.

clear : To clear a bit is to write a 0 to it. To clear a register or memory location is to load all its bits with 0s. See also *set*.

COFF: *Common object file format.* A binary object file format that promotes modular programming by supporting the concept of sections, where a section is a relocatable block of code or data that ultimately occupies a space adjacent to other blocks of code in the memory map.

conditional branch instruction: A branch instruction that may or may not cause a branch, depending on a specified or predefined condition (for example, the state of a bit).

context restore: A restoring of the previous state of a system (for example, modes and key register values) prior to returning from a subroutine. See also *context save*.

context save: A save of the current state of a system (for example, modes and key register values) prior to executing the main body of a subroutine that requires a different context. See also *context restore*.

core: The portion of the ’27x that consists of a CPU, a block of emulation circuitry, and a set of signals for interfacing with memory and peripheral devices.

current auxiliary register: The register selected by the auxiliary register pointer (ARP) in status register. For example, if ARP = 3, the current auxiliary register is AR3. See also *auxiliary registers*.

current data page: The data page selected by the data page pointer. For example, if DP = 0, the current data page is 0. See also *data page*.

D

D1 phase: See *decode 1 (D1) phase*.

D2 phase: See *decode 2 (D2) phase*.

data logging: Transferring one or more packets of data from CPU registers or memory to an external host processor.

data log interrupt (DLOGINT): A maskable interrupt triggered by the on-chip emulation logic when a data logging transfer has been completed.

data page: A 64-word portion of the total 4M words of data space. Each data page has a specific start address and end address. See also *data page pointer (DP)* and *current data page.*

data page pointer (DP): A 16-bit pointer that identifies which 64-word data page is accessed in DP direct addressing mode. For example, for as long as DP = 500, instructions that use DP direct addressing mode will access data page 500.

data-/program-write data bus (DWDB): The bus that carries data during writes to data space or program space.

data-read address bus (DRAB): The bus that carries addresses for reads from data space.

data-read data bus (DRDB): The bus that carries data during reads from data space.

data-write address bus (DWAB): The bus that carries addresses for writes to data space.

DBGIER: See *debug interrupt enable register (DBGIER)*.

DBGM bit: See *debug enable mask (DBGM) bit*.

DBGSTAT: See *debug status register (DBGSTAT)*.

debug-and-test direct memory access (DT–DMA): An access of a register or memory location to provide visibility to this location during debugging. The access is performed with variable levels of intrusiveness by a hardware DT-DMA mechanism inside the core.

debug enable mask (DBGM) bit: A bit in status register ST1 used to enable (DBGM = 0) or disable (DBGM = 1) debug events such as analysis breakpoints or debug-and-test direct memory accesses (DT-DMAs).

debug event: An action such as the decoding of a software breakpoint instruction, the occurrence of an analysis breakpoint/watchpoint, or a request from a host processor that may result in special debug behavior, such as halting the device or pulsing one of the debug interface signals EMU0 or EMU1. See also *break event* and *debug enable mask (DBGM) bit*.

debug-halt state: A debug execution state that is entered through a break event. In this state the CPU is halted. See also *single-instruction state* and *run state*.

debug host: See *host processor*.

debug interrupt enable register (DBGIER): The register that determines which of the maskable interrupts are time-critical when the CPU is halted in real-time mode. If a bit in the DBGIER is 1, the corresponding interrupt is time-critical/enabled; otherwise, it is disabled. Time-critical interrupts also must be enabled in the interrupt enable register (IER) to be serviced.

debug status register (DBGSTAT): A register that holds special debug status information. This register, which need not be read from or written to, is saved and restored during interrupt servicing, to preserve the debug context during debugging.

decode an instruction: To identify an instruction and prepare the CPU to perform the operation the instruction requires.

decode 1 (D1) phase: The third of eight pipeline phases an instruction passes through. In this phase, the CPU identifies instruction boundaries in the instruction-fetch queue and determines whether the next instruction to be executed is an illegal instruction. See also *pipeline phases*.

decode 2 (D2) phase: The fourth of eight pipeline phases an instruction passes through. In this phase, the CPU accepts an instruction from the instruction-fetch queue and completes the decoding of that instruction, performing such activities as address generation and pointer modification. See also *pipeline phases*.

decrement: To subtract 1 or 2 from a register or memory value. The value subtracted depends on the circumstance. For example, if you use the operand *––AR4, the auxiliary register AR4 is decremented by 1 for a 16-bit operation and by 2 for a 32-bit operation.

device reset: See *reset*.

direct addressing modes: The addressing modes that access data space as if it were 65 536 separate blocks of 64 words each. DP direct addressing mode uses the data page pointer (DP) to select a data page from 0 to 65 535. PAGE0 direct addressing mode uses data page 0, regardless of the value in the DP.

discontinuity: See *program-flow discontinuity*.

DLOGINT: See *data log interrupt (DLOGINT)*.

DP: See *data page pointer (DP)*.

DP direct addressing mode: A direct addressing mode that uses the data page pointer (DP) to select a data page from 0 to 65 535. See also *PAGE0 direct addressing mode*.

DRAB: See *data-read address bus (DRAB)*.

DRDB: See *data-read data bus (DRDB)*.

DT–DMA: See *debug-and-test direct memory access (DT-DMA)*.

DWAB: See *data-write address bus (DWAB)*.

DWDB: See *data-/program-write data bus (DWDB)*.

E

E phase: See *execute (E) phase*.

EALLOW bit: See *emulation access enable (EALLOW) bit*.

EMU0 and EMU1 pins: Pins known as the TI extensions to the JTAG interface. These pins can be used as either inputs or outputs and are available to help monitor and control an emulation target system that is using a JTAG interface.

emulation access enable (EALLOW) bit: A bit in status register ST1 that enables (EALLOW = 1) or disables (EALLOW = 0) access to the emulation registers. The EALLOW instruction sets the EALLOW bit, and the EDIS instruction clears the EALLOW bit.

emulation logic: The block of hardware in the core that is responsible controlling emulation activities such as data logging and switching among debug execution states.

emulation registers: Memory-mapped registers that are available for controlling and monitoring emulation activities.

enable bit: See *interrupt enable bits*.

execute an instruction: Take an instruction from the decode 2 phase of the pipeline through the write phase of the pipeline.

execute (E) phase: The seventh of eight pipeline phases an instruction passes through. In this phase, the CPU performs all multiplier, shifter, and arithmetic-logic-unit (ALU) operations. See also *pipeline phases*.

extended auxiliary registers: See *XAR6/XAR7*.

F

F1 phase: See *fetch 1 (F1) phase*.

F2 phase: See *fetch 2 (F2) phase*.

FC : See *fetch counter (FC)*.

fetch 1 (F1) phase: The first of eight pipeline phases an instruction passes through. In this phase, the CPU places on the program-read bus the address of the instruction(s) to be fetched. See also *pipeline phases*.

fetch 2 (F2) phase: The second of eight pipeline phases an instruction passes through. In this phase, the CPU fetches an instruction or instructions from program memory. See also *pipeline phases*.

fetch counter (FC) : The register that contains the address of the instruction that is being fetched from program memory.

field : See *bit field*.

H

hardware interrupt: An interrupt initiated by a physical signal (for example, from a pin or from the emulation logic). See also *software interrupt*.

hardware interrupt priority: A priority ranking used by the CPU to determine the order in which simultaneously occurring hardware interrupts are serviced.

hardware reset: See *reset*.

high addresses: Addresses closer to 3F $FFFF_{16}$ than to 00 0000_{16}. See also *low addresses*.

high bits: See *MSB*.

high word: The 16 MSBs of a 32-bit value. See also *low word*.

host processor: The processor running the user interface for a debugger.

I

IC: See *instruction counter (IC)*.

IDLESTAT (IDLE status) bit: A bit in status register ST1 that indicates when an IDLE instruction has the CPU in the idle state (IDLESTAT = 1).

idle state: The low-power state the CPU enters when it executes the IDLE instruction.

IEEE 1149.1 standard: "IEEE Standard Test Access Port and Boundary-Scan Architecture", first released in 1990. See also *JTAG*.

IER: See *interrupt enable register (IER)*.

IFR: See *interrupt flag register (IFR)*.

illegal instruction: An unacceptable value read from program memory during an instruction fetch. Unacceptable values are 0000_{16}, $FFFF_{16}$, or any value that does not match a defined opcode.

illegal-instruction trap: A trap that is serviced when an illegal instruction is decoded.

immediate address: An address that is specified directly in an instruction as a constant.

immediate addressing modes: Addressing modes that accept a constant as an operand.

immediate constant/data: A constant specified directly as an operand of an instruction.

immediate-constant addressing mode: An immediate addressing mode that accepts a constant as an operand and interprets that constant as data to be stored or processed.

immediate-pointer addressing mode: An immediate addressing mode that accepts a constant as an operand and interprets that constant as the 16 LSBs of a 22-bit address. The six MSBs of the address are filled with 0s.

increment: To add 1 or 2 to a register or memory value. The value added depends on the circumstance. For example, if you use the operand *AR4++, the auxiliary register AR4 is incremented by 1 for a 16-bit operation and by 2 for a 32-bit operation.

indirect addressing modes: Addressing modes that use pointers to access memory. The available pointers are auxiliary registers AR0–AR5, extended auxiliary registers XAR6 and XAR7, and the stack pointer (SP).

instruction boundary: The point where the CPU has finished one instruction and is considering what it will do next — move on to the next instruction.

instruction counter (IC): The register that points to the instruction in the decode 1 phase (the instruction that is to enter the decode 2 phase next). Also, on an interrupt or call operation, the IC value represents the return address, which is saved to the stack or to auxiliary register XAR7.

instruction-fetch mechanism: The hardware for the fetch 1 and fetch 2 phases of the pipeline. This hardware is responsible for fetching instructions from program memory and filling an instruction-fetch queue.

instruction-fetch queue: A queue of four 32-bit registers that receives fetched instructions and holds them for decoding. When a program-flow discontinuity occurs, the instruction-fetch queue is emptied.

instruction-not-available condition: The condition that occurs when the decode 2 pipeline hardware requests an instruction but there are no instructions waiting in the instruction-fetch queue. This condition causes the decode 2 through write phases of the pipeline to freeze until one or more new instructions have been fetched.

instruction register: The register that contains the instruction that has reached the decode 2 pipeline phase.

instruction word: Either an entire 16-bit opcode or one of the halves of a 32-bit opcode.

$\overline{\text{INT1}}$–$\overline{\text{INT14}}$: Fourteen general-purpose interrupts that are triggered by signals at pins of the same names. These interrupts are maskable and have corresponding bits in the interrupt flag register (IFR), the interrupt enable register (IER), and the debug interrupt enable register (DBGIER).

Interrupt boundary: An instruction boundary where the CPU can insert an interrupt between two instructions. See also *instruction boundary*.

interrupt enable bits: Bits responsible for enabling or disabling maskable interrupts. The enable bits are all the bits in the interrupt enable register (IER), all the bits in the debug interrupt enable register (DBGIER), and the interrupt global mask bit (INTM in status register ST1).

interrupt enable register (IER): Each of the maskable interrupts has an interrupt enable bit in this register. If a bit in the IER is 1, the corresponding interrupt is enabled; otherwise, it is disabled. See also *debug interrupt enable register (DBGIER)*.

interrupt flag bit: A bit in the interrupt flag register (IFR). If the interrupt flag bit is 1, the corresponding interrupt has been requested by hardware and is awaiting approval by the CPU.

interrupt flag register (IFR): The register that contains the interrupt flag bits for the maskable interrupts. If a bit in the IFR is 1, the corresponding interrupt has been requested by hardware and is awaiting approval by the CPU.

interrupt global mask (INTM) bit: A bit in status register ST1 that globally enables or disables the maskable interrupts. If an interrupt is enabled in the interrupt enable register (IER) but not by the INTM bit, it is not serviced. The only time this bit is ignored is when the CPU is in real-time mode and is in the debug-halt state; in this situation, the interrupt must be enabled in the IER and in the DBGIER (debug interrupt enable register).

interrupt priority: See *hardware interrupt priority*.

interrupt request: A signal or instruction that requests the CPU to execute a particular interrupt service routine. See also *approve an interrupt request* and *service an interrupt*.

interrupt service routine (ISR): A subroutine that is linked to a specific interrupt by way of an interrupt vector.

interrupt vector: The start address of an interrupt service routine. After approving an interrupt request, the CPU fetches the interrupt vector from your interrupt vector table and uses the vector to branch to the start of the corresponding interrupt service routine.

interrupt vector location: The preset location in program memory where an interrupt vector must reside.

interrupt vector table: The list of interrupt vectors you assign in program memory.

INTM bit: See *interrupt global mask (INTM) bit*.

ISR: See *interrupt service routine (ISR)*.

J

JTAG: *Joint Test Action Group*. The Joint Test Action Group was formed in 1985 to develop economical test methodologies for systems designed around complex integrated circuits and assembled with surface-mount technologies. The group drafted a standard that was subsequently adopted by IEEE as IEEE Standard 1149.1-1990, "IEEE Standard Test Access Port and Boundary-Scan Architecture". See also *boundary scan*; *test access port (TAP)*.

JTAG port: See *test access port (TAP)*.

L

latch: Hold a bit at the same value until a given event occurs. For example, when an overflow occurs in the accumulator, the V bit is set and latched at 1 until it is cleared by a conditional branch instruction or by a write to status register ST0. An interrupt is latched when its flag bit has been latched in the interrupt flag register (IFR).

least significant bit (LSB): The bit in the lowest position of a binary number. For example, the LSB of a 16-bit register value is bit 0. See also *MSB*, *LSByte*, and *MSByte*.

least significant byte (LSByte): The byte in the lowest position of a binary value. The LSByte of a value consists of the eight LSBs. See also *MSByte*, *LSB*, and *MSB*.

location: A space where data can reside. A location may be a CPU register or a space in memory.

logical shift: A shift that treats the shifted value as unsigned. See also *arithmetic shift*.

LOOP (loop instruction status) bit: A bit in status register ST1 that indicates when a LOOPNZ or LOOPZ instruction is being executed (LOOP = 1).

low addresses: Addresses closer to $00\ 0000_{16}$ than to $3F\ FFFF_{16}$. See also *high addresses*.

low bits: See *LSB*.

low word: The 16 LSBs of a 32-bit value. See also *high word*.

LSB: When used in a syntax of the MOVB instruction, LSB means least significant byte. Otherwise, LSB means least significant bit. See *least significant bit (LSB)* and *least significant byte (LSByte)*.

LSByte: See *least significant byte (LSByte)*.

M

maskable interrupt: An interrupt that can be disabled by software so that the CPU does not service it until it is enabled by software. See also *nonmaskable interrupt*.

memory interface: The buses and signals responsible for carrying communications between the core and on-chip memory/peripherals.

memory-mapped register: A register that can be accessed at addresses in data space.

memory wrapper: The hardware around a memory block that identifies access requests and controls accesses for that memory block.

mirror: A range of addresses that is the same size and is mapped to the same physical memory block as another range of addresses.

most significant bit (MSB): The bit in the highest position of a binary number. For example, the MSB of a 16-bit register value is bit 15. See also *LSB*, *LSByte*, and *MSByte*.

most significant byte (MSByte): The byte in the highest position of a binary value. The MSByte of a value consists of the eight MSBs. See also *LSByte*, *LSB*, and *MSB*.

MSB: When used in a syntax of the MOVB instruction, MSB means most significant byte. Otherwise MSB means most significant bit. See *most significant bit (MSB)* and *most significant byte (MSByte)*.

MSByte: See *most significant byte (MSByte)*.

multiplicand register (T): The primary function of this register, also called the T register, is to hold one of the values to be multiplied during a multiplication. The following shift instructions use the four LSBs to hold the shift count: ASR (arithmetic shift right), LSL (logical shift left), LSR (logical shift right), and SFR (shift accumulator right). The T register can also be used as a general-purpose 16-bit register.

N

N (negative flag) bit: A bit in status register ST0 that indicates whether the result of a calculation is a negative number (N = 1). N is set to match the MSB of the result.

nested interrupt: An interrupt that occurs within an interrupt service routine.

$\overline{\text{NMI}}$: A hardware interrupt that is nonmaskable, like reset ($\overline{\text{RS}}$), but does not reset the CPU. $\overline{\text{NMI}}$ simply forces the CPU to execute its interrupt service routine.

nonmaskable interrupt: An interrupt that cannot be blocked by software and is approved by the CPU immediately. See also *maskable interrupt*.

O

offset branch: A branch that uses a specified or generated offset value to jump to an address relative to the current position of the program counter (PC). See also *absolute branch*.

opcode: This document uses opcode to mean the complete code for an instruction. Thus, an opcode includes the binary sequence for the instruction type and the binary sequence and/or constant in which the operands are encoded.

operand : This document uses operand to mean one of the values entered after the instruction mnemonic and separated by commas (or for a shift operand, separated by the symbol <<). For example, in the CLRC INTM instruction, CLRC is the mnemonic and INTM is the operand.

operation: 1) A defined action; namely, the act of obtaining a result from one or more operands in accordance with a rule that completely specifies the result of any permitted combination of operands. 2) The set of such acts specified by a rule, or the rule itself. 3) The act specified by a single computer instruction. 4) A program step undertaken or executed by a computer; for example, addition, multiplication, extraction, comparison, shift, transfer, etc. 5) The specific action performed by a logic element.

OVC: See *overflow counter (OVC)*.

OVM: See *overflow mode (OVM) bit*.

overflow counter (OVC): A 6-bit counter in status register ST0 that can be used to track overflows in the accumulator (ACC). The OVC is enabled only when the overflow mode (OVM) bit in ST0 is 0. When OVM = 0, the OVC is incremented by 1 for every overflow in the positive direction (too large a positive number) and decremented by 1 for every overflow in the negative direction (too large a negative number). The saturate (SAT) instruction modifies ACC to reflect the net overflow represented in the OVC.

overflow flag (V): A bit in status register ST0 that indicates when the result of an operation causes an overflow in the location holding the result (V = 1). If no overflow occurs, V is not modified.

overflow mode (OVM) bit: A bit in the status register ST0 that enables or disables overflow mode. When overflow mode is on (OVM = 1) and an overflow occurs, the CPU fills the accumulator (ACC) with a saturation value. When overflow mode is off (OVM = 0), the CPU lets ACC overflow normally but keeps track of each overflow by incrementing or decrementing by 1 the overflow counter (OVC) in ST0.

P

P register: See *product register (P).*

PAB: See *program address bus (PAB)*.

PAGE0 bit: *PAGE0 addressing mode configuration bit*. This bit, in status register ST1, selects between two addressing modes: PAGE0 stack addressing mode (PAGE = 0) and PAGE0 direct addressing mode (PAGE0 = 1).

PAGE0 direct addressing mode: The direct addressing mode that uses data page 0 regardless of the value in the data page pointer (DP). This mode is available only when the PAGE0 bit in status register ST1 is 1. See also *DP direct addressing mode* and *PAGE0 stack addressing mode*.

PAGE0 stack addressing mode: The indirect addressing mode that references a value on the stack by subtracting a 6-bit offset from the current position of the stack pointer (SP). This mode is available only when the PAGE0 bit in status register ST1 is 0. See also *stack-pointer indirect addressing mode*.

PC: See *program counter (PC)*.

pending interrupt: An interrupt that has been requested but is waiting for approval from the CPU. See also *approve an interrupt request.*

peripheral-interface logic: Hardware that is responsible for handling communications between a processor and a peripheral.

PH: The high word (16 MSBs) of the P register.

phases: See *pipeline phases*.

pipeline: The hardware in the CPU that takes each instruction through eight independent phases for fetching, decoding, and executing. During any given CPU cycle, there can be up to eight instructions in the pipeline, each at a different phase of completion. The phases, listed in the order in which instructions pass through them, are fetch 1, fetch 2, decode 1, decode 2, read 1, read 2, execute, and write.

pipeline conflict: A situation in which two instructions in the pipeline try to access a register or memory location out of order, causing improper code operation. The '27x pipeline inserts as many inactive cycles as needed between conflicting instructions to prevent pipeline conflicts.

pipeline freeze: A halt in pipeline activity in one of the two decoupled portions of the pipeline. Freezes in the fetch 1 through decode 1 portion of the pipeline are caused by a not-ready signal from program memory. Freezes in the decode 2 through write portion are caused by lack of instructions in the instruction-fetch queue or by not-ready signals from memory.

pipeline phases: The eight stages an instruction must pass through to be fetched, decoded, and executed. The phases, listed in the order in which instructions pass through them, are fetch 1, fetch 2, decode 1, decode 2, read 1, read 2, execute, and write.

pipeline-protection mechanism: The mechanism responsible for identifying potential pipeline conflicts and preventing them by adding inactive cycles between the conflicting instructions.

PL: The low word (16 LSBs) of the P register.

PM bits: See *product shift mode (PM) bits*.

PRDB: See *program-read data bus (PRDB)*.

priority: See *interrupt priority*.

product register (P): This register, also called the P register, is given the results of most multiplications done by the CPU. The only other register that can be given the result of a multiplication is the accumulator (ACC). See also *PH* and *PL*.

product shift mode (PM) bits: A 3-bit field in status register ST0 that enables you to select one of eight product shift modes. The product shift mode determines whether or how the P register value is shifted before being used by an instruction. You have the choices of a left shift by 1 bit, no shift, or a right shift by N, where N is a number from 1 to 6.

program address bus (PAB): The bus that carries addresses for reads and writes from program space.

program address generation logic: This logic generates the addresses used to fetch instructions or data from program memory and places each address on the program address bus (PAB).

program control logic: This logic stores a queue of instructions that have been fetched from program memory by way of the program-read bus (PRDB). It also decodes these instructions and passes commands and constant data to other parts of the CPU.

program counter (PC): When the pipeline is full, the 22-bit PC always points to the instruction that is currently being processed—the instruction that has just reached the decode 2 phase of the pipeline.

program-flow discontinuity: A branching to a nonsequential address caused by a branch, a call, an interrupt, a return, or the repetition of an instruction.

program-read data bus (PRDB): The bus that carries instructions or data during reads from program space.

R

R1 phase: See *read 1 (R1) phase*.

R2 phase: See *read 2 (R2) phase*.

read 1 (R1) phase: The fifth of eight pipeline phases an instruction passes through. In this phase, if data is to be read from memory, the CPU drives the address(es) on the appropriate address bus(es). See also *pipeline phases*.

read 2 (R2) phase: The sixth of eight pipeline phases an instruction passes through. In this phase, data addressed in the read 1 phase is fetched from memory. See also *pipeline phases*.

ready signals: When the core requests a read from or write to a memory device or peripheral device, that device can take more time to finish the data transfer than the core allots by default. Each device must use one of the core's *ready signals* to insert wait states into the data transfer when it needs more time. Wait-state requests freeze a portion of the pipeline if they are received during the fetch 1, read 1, or write pipeline phase of an instruction.

real-time mode: An emulation mode that enables you execute certain interrupts (time-critical interrupts), even when the CPU is halted. See also *stop mode*.

real-time operating system interrupt (RTOSINT): A maskable hardware interrupt generated by the emulation hardware in response to certain debug events. This interrupt should be disabled in the interrupt enable register (IER) and the debug interrupt enable register (DBGIER) unless there is a real-time operating system present in your debug system.

reduced instruction set computer (RISC): A computer whose instruction set and related decode mechanism are much simpler than those of microprogrammed complex instruction set computers.

register addressing mode: An addressing mode that enables you to reference registers by name.

register conflict: A pipeline conflict that would occur if an instruction read a register value before that value were changed by a prior instruction. The '27x pipeline inserts as many inactive cycles as needed between conflicting instructions to prevent register conflicts.

register pair: One of the pairs of CPU register stored to the stack during an automatic context save.

repeat counter (RPTC): The counter that is loaded by the RPT (repeat) instruction. The number in the counter is the number of times the instruction qualified by RPT is to be repeated after its initial execution.

reserved: A term used to describe memory locations or other items that you cannot use or modify.

reset: To return the DSP to a known state; an action initiated by the reset ($\overline{RS}$) signal.

return: 1) The operation of forcing program control to a return address. 2) An instruction that performs such an operation. See also *call*.

return address: The address at which the CPU resumes processing after executing a subroutine or interrupt service routine.

RISC: See *reduced instruction set computer (RISC)*.

rotate operation: An operation performed by the ROL (rotate accumulator left) or ROR (rotate accumulator right) instruction. The operation, which involves a shift by 1 bit, can be seen as the rotation of a 33-bit value that is the concatenation of the carry bit (C) and the accumulator (ACC).

RPTC: See *repeat counter (RPTC)*.

RTOSINT : See *real-time operating system interrupt (RTOSINT).*

RUN command : A debugger command used to execute all or a portion of a program. The RUN 1 command causes the debugger to execute a single instruction.

run state: A debug execution state. In this state, the CPU is executing code and servicing interrupts freely. See also *debug-halt state* and *single-instruction state*.

S

select signal: An output signal from the ’27x that can be used to select specific memory or peripheral devices for particular types of read and write operations.

scan controller: A device that performs JTAG state sequences sent to it by a host processor. These sequences, in turn, control the operation of a target device.

service an interrupt : The CPU services an interrupt by preparing for and then executing the corresponding interrupt service routine. See also *interrupt request* and *approve an interrupt request*.

set: To set a bit is to write a 1 to it. If a bit is set, it contains 1. See also *clear*.

sign extend: To fill the unused most significant bits (MSBs) of a value with copies of the value’s sign bit.

sign-extension mode (SXM) bit: A bit in status register ST0 that enables or suppresses sign extension. When sign-extension is enabled (SXM = 1), operands of certain instructions are treated as signed and are sign extended during shifting.

single-instruction state: A debug execution state. In this state, the CPU executes one instruction and then returns to the debug-halt state. See also *debug-halt state* and *run state*.

16-bit operation: An operation that reads or writes 16 bits.

software interrupt: An interrupt initiated by an instruction. See also *hardware interrupt*.

SP: See *stack pointer (SP)*.

SPA bit: See *stack pointer alignment (SPA) bit*.

ST0: See *status registers ST0 and ST1*.

ST1: See *status registers ST0 and ST1*.

stack : The ’27x stack is a software stack implemented by the use of a stack pointer (SP). The SP, a 16-bit CPU register, can be used to reference a value in the first 64K words of data memory (addresses $00\ 0000_{16}$–$00\ FFFF_{16}$).

stack pointer (SP): A 16-bit CPU register that enables you to use any portion of the first 64K words of data memory as a software stack. The SP always points to the next empty location in the stack.

stack pointer alignment (SPA) bit: A bit in status register ST1 that indicates whether an ASP instruction has forced the SP to align to the next even address (SPA = 1).

stack-pointer indirect addressing mode: The indirect addressing mode that references a data-memory value at the current position of the stack pointer (SP). See also *PAGE0 stack addressing mode*.

status registers ST0 and ST1: These CPU registers contain control bits that affect the operation of the ’27x and contain flag bits that reflect the results of operations.

STEP command: A debugger command that causes the debugger to single-step through a program. The STEP1 command causes the debugger to execute a single instruction.

stop mode: An emulation mode that provides complete control of program execution. When the CPU is halted in stop mode, all interrupts (including reset and nonmaskable interrupts) are ignored until the CPU receives a directive to run code again. See also *real-time mode*.

suppress sign extension: Prevent sign extension from occurring during a shift operation. See also *sign extend*.

SXM bit: See *sign-extension mode (SXM) bit*.

T

T register: The primary function of this register, also called the multiplicand register, is to hold one of the values to be multiplied during a multiplication. The following shift instructions use the four LSBs to hold the shift count: ASR (arithmetic shift right), LSL (logical shift left), LSR (logical shift right), and SFR (shift accumulator right). The T register can also be used as a general-purpose 16-bit register.

TAP: See *test access port (TAP)*.

target device/system: The device/system on which the code you have developed is executed.

TC bit: See *test/control flag (TC)*.

test access port (TAP): A standard communication port defined by IEEE standard 1149.1–1990 included in the DSP to implement boundary scan functions and/or to provide communication between the DSP and emulator.

test/control flag (TC): A bit in status register ST0 that shows the result of a test performed by the TBIT (test bit) instruction or the NORM (normalize) instruction.

test-logic-reset: A test and emulation logic condition that occurs when the $\overline{\text{TRST}}$ signal is pulled low or when the TMS signal is used to advance the JTAG state machine to the TLR state. This logic is a different type than that used by the CPU, which resets functional logic.

32-bit operation: An operation that reads or writes 32 bits.

TI extension pins: See *EMU0 and EMU1 pins*.

time-critical interrupt: An interrupt that must be serviced even when background code is halted. For example, a time-critical interrupt might service a motor controller or a high-speed timer. See also *debug interrupt enable register (DBGIER)*.

U

USER1–USER12 interrupts: The interrupt vector table contains twelve locations for user-defined software interrupts. These interrupts, called USER1–USER12 in this document, can be initiated only by way of the TRAP instruction.

V

V bit (overflow flag): A bit in status register ST0 that indicates when the result of an operation causes an overflow in the location holding the result (V = 1). If no overflow occurs, V is not modified.

vector: See *interrupt vector*.

vector location: See *interrupt vector location*.

vector map (VMAP) bit: A bit in status register ST1 that determines the addresses to which the interrupt vectors are mapped. When VMAP = 0, the interrupt vectors are mapped to addresses 00 0000_{16}–00 $003F_{16}$ in program memory. When VMAP = 1, the vectors are mapped to addresses 3F $FFC0_{16}$–3F $FFFF_{16}$ in program memory.

vector table: See *interrupt vector table*.

W

W phase: See *write (W) phase*.

wait state: A cycle during which the CPU waits for a memory or peripheral device to be ready for a read or write operation.

watchpoint: A place in a routine where it is to be halted if an address or an address and data combination match specified compare values. When a watchpoint is reached, the routine is halted and the CPU enters the debug-halt state.

word: In this document, a word is 16 bits unless specifically stated to be otherwise.

write (W) phase: The last of eight pipeline phases an instruction passes through. In this phase, if a value or result is to be written to memory, the CPU sends to memory the destination address and the data to be written. See also *pipeline phases*.

X

XAR6/XAR7: *Extended auxiliary registers 6 and 7*. Two 22-bit registers that are used as pointers to any address in data space. The registers are operated on by the address register arithmetic unit (ARAU) and are selected by the auxiliary register pointer (ARP). Some instructions use XAR7 as a pointer for program space. The FCC (fast function call) instruction stores a return address in XAR7. See also *AR0–AR5 and AR6/AR7*.

XAR7 indirect addressing mode: The indirect addressing mode in which extended auxiliary register XAR7 is used as a pointer to a location in program space.

Z

zero fill: Fill the unused low- and/or high-order bits of a value with 0s.

zero flag (Z): A bit in status register ST0 that indicates when the result of an operation is 0 (Z = 1).

Index

14-pin connector
  dimensions C-15
14-pin header
  header signals C-2
  JTAG C-2

A

abort interrupt (ABORTI instruction) 6-40
ABORTI instruction 6-40, 7-14
ABS instruction 6-41
absolute value (ABS instruction) 6-41
access
  data space 1-10
  program space 1-10
access to CPU registers during emulation 7-15
access to memory during emulation 7-15
accesses
  polite 7-15
  rude 7-15
accessing an array 8-23
accessing values saved to the stack during interrupt processing
  figure 8-12
  reading and modifying correct values 8-13
accumulator 2-5
  ACC 32-bit operations 6-15
  ACC operations 6-13
  AH (high word) 2-5
  AH/AL byte operations 6-13
  AH/AL operations 6-12
  AH.LSB 2-6
  AH.MSB 2-6
  AL (low word) 2-5
  AL.LSB 2-6
  AL.MSB 2-6
accumulator (continued)
  portions that are individually accessible 2-6
  use to modify or read stack values 8-13
ADD instruction 6-43
add long value 6-58
ADDB instruction 6-51
ADDCU instruction 6-55
addition instructions
  ADD (add value to specified location) 6-43
  ADDB (add short value) 6-51
  ADDCU (add value plus carry) 6-55
  ADDL (add long value) 6-58
  ADDU (add unsigned value to ACC) 6-62
  ADRK (add to auxiliary register) 6-65
ADDL instruction 6-58
address buses 1-11
address counters FC, IC, and PC 4-5
address maps 1-6
address register arithmetic unit (ARAU) 1-4, 2-2, 6-52, 6-281
address register operations 6-10
addressing modes
  ARP indirect 5-13
    opcode information 5-24
  auxiliary-register indirect 5-11
    opcode information 5-23
  circular 5-16, 8-30
    opcode information 5-24
  direct 1-10, 2-8, 5-4
    opcode information 5-23
    role of data page pointer (DP) 5-4
  DP direct 5-4
    opcode information 5-23
  immediate 5-2, 5-21
    opcode information 5-21
  indirect 1-10, 2-10, 5-15
    description 5-10
    opcode information 5-23
    operand summary 5-18

addressing modes (continued)
information in opcodes 5-21
overview 5-1
PAGE0 direct 5-5
opcode information 5-23
PAGE0 stack 5-8
opcode information 5-23
register 5-6
opcode information 5-23, 5-24
stack-pointer indirect 5-7
opcode information 5-24
XAR7 indirect 5-15
ADDRH register 7-22
ADDRL register 7-22
ADDU instruction 6-62
ADRK instruction 6-65
AH (high word of accumulator) 2-5
AL (low word of accumulator) 2-5
align stack pointer instruction (ASP) 6-72, 8-27
alignment of 32-bit accesses to even addresses 6-31
allow access to emulation registers 6-98
AL.LSB (part of accumulator) 2-6
AL.MSB (part of accumulator) 2-6
analysis resources
breakpoints 7-18
clearing resources 7-28
counters 7-19
data logging 7-21
sharing resources 7-28
watchpoints 7-18
AND IER and OR IER instructions
note about RTOSINT 3-8
AND instruction 6-66
ANDB instruction 6-71
architectural overview 1-1
arithmetic logic unit (ALU) 1-4, 2-2, 6-52, 6-281
ARP indirect addressing mode 5-13
opcode information 5-24
ASP instruction 6-72, 8-27
ASR instruction 6-74
assembly language instructions 6-1
syntax formats 6-2
automatic context save
changing values saved during 8-11
auxiliary registers
add to 6-65
AR0–AR5, XAR6, XAR7 2-10
BANZ (branch if auxiliary register not equal to 0) 6-80
indirect addressing modes that use 5-10
pointer 2-19, 5-13, 6-65, 6-263
subtract from 6-263
use as base pointers in arrays 8-23
auxiliary-register addressing mode 5-11
auxiliary-register indirect addressing mode
opcode information 5-23

B

B instruction 6-77, 8-5
B0 block memory map 1-8
B1 block memory map 1-9
background code 7-6
BANZ instruction 6-80, 8-5
barrel shifter 1-4, 2-2
base pointer 8-23
benchmark counter 7-19
bits
auxiliary register pointer (ARP) 2-19
carry (C) 2-16
debug enable mask (DBGM) 2-22
debug interrupt enable register (DBGIER) 3-9
emulation access enable (EALLOW) 2-20, 6-98
IDLE status (IDLESTAT) 2-20
interrupt enable register (IER) 3-8
interrupt flag register (IFR) 3-6
interrupt global mask (INTM) 2-23
loop instruction status (LOOP) 2-20
negative flag (N) 2-15
overflow counter (OVC) 2-13
overflow flag (V) 2-15
overflow mode (OVM) 2-17
PAGE0 addressing mode configuration 2-21
product shift mode (PM) 2-14
setting key status 8-3
sign-extension mode 2-18
stack pointer alignment (SPA) 2-20, 6-72, 6-207
test/control flag (TC) 2-17
vector map (VMAP) 2-21
zero flag (Z) 2-16
bitwise AND with short value 6-71
bitwise exclusive OR 6-307
bitwise exclusive OR with short value 6-310

bitwise OR 6-219
bitwise OR with short value 6-223
block B0 1-7
block B1 1-9
block diagram of the CPU
  figure 2-3
branch
  based on multiple conditions 8-8
  based on single condition 8-5
  instructions introduced 2-24
  to address determined at run time 8-4
branch instructions
  B (branch) 6-77
  BANZ (branch if auxiliary register not equal to 0) 6-80
  LB (long branch) 6-118
  SB (short branch) 6-257
break event 7-6
breakpoints 7-18
  caution about time-critical ISRs 7-11
buffered signals
  JTAG C-10
buffering C-10
bus devices C-4
bus protocol in emulator system C-4
buses
  data-/program-write data 1-11
  data-read address 1-11
  data-read data 1-11
  data-write address 1-11
  program address 1-11
  program-read data 1-11
  special operations 1-12
  summary table 1-12
byte accesses 8-22

C

C bit 2-16
cable pod C-5, C-6
CALL instruction 6-82
calls 2-24
carry bit (C) 2-16
caution
  breakpoints within time-critical interrupt service routines 7-11
central processing unit (CPU) 1-3, 2-2
  reset 3-22
    in real-time mode debug-halt state 7-9
    initializing pointers and status bits after 8-2
circular addressing mode 5-16
  opcode information 5-24
circular buffer 8-30
clear status bits 6-84
CLRC DBGM instruction 8-3
CLRC instruction 6-84
CLRC INTM instruction 8-3
CMP instruction 6-87
CMPB instruction 6-91
CMPL instruction 6-93
conditional subtraction 6-284
conditions tested by conditional branch instructions 8-7
configuration
  multiprocessor C-13
connector
  14-pin header C-2
  dimensions, mechanical C-14
  DuPont C-2
control operations 6-18
core 1-2
  components 1-3
  diagram 1-3
counters 7-19
CPU 1-3, 2-2
  reset 3-22
    in real-time mode debug-halt state 7-9
    initializing pointers and status bits after 8-2
CPU registers 2-4
custom ROM codes B-1

D

data
  aligning to even addresses 8-25
  moving blocks of 8-20
data buses 1-11
data log interrupt (DLOGINT) 3-5, 7-25
  initiating by using INTR instruction 6-108
  vector 3-3
data logging 7-21
  accessing emulation registers 7-24
  creating a transfer buffer 7-21

data logging (continued)
examples 7-26
interrupt (DLOGINT) 3-5, 7-25
initiating by using INTR instruction 6-108
interrupt vector 3-3
with end address 7-27
with word counter 7-26
data logging end-address control register 7-24
data move operations 6-16
data page pointer (DP) 2-8
role in direct addressing 5-4
data space
accessing 1-10
address map 1-6
data-/program-write data bus (DWDB) 1-11
data-read address bus (DRAB) 1-11
data-read data bus (DRDB) 1-11
data-write address bus (DWAB) 1-11
DBGSTAT register 6-40, 7-14, D-6
debug
enable mask bit (DBGM) 2-22
event 7-6
execution control modes 7-7
halt state 7-6
interface 7-3
sharing resources 7-28
terminology 7-6
debug enable mask bit (DBGM) 2-22
role in accesses during emulation 7-15
set during interrupt handling 3-14, 3-19
debug interrupt enable register (DBGIER) 3-5, 3-7, 3-9, 7-9
quick reference figure A-8
debug status register (DBGSTAT) 6-40, 7-14, D-6
debug-and-test direct memory access (DT-DMA) mechanism 7-15
debug-halt state 7-7, 7-9
DEC instruction 6-96
decoupled pipeline segments 4-4
decrement by 1 6-96
development interface 7-2
diagnostic applications C-24
diagnostic features for emulation 7-29
diagrams
CPU 2-3
memory map 1-6
multiplication 2-26
diagrams (continued)
pipeline activity 4-8
pipeline conflict 4-13, 4-14
possible maps for block B0 1-8
possible maps for block B1 1-9
relationship between pipeline and address counters 4-6
shift operations 2-28
T320C2700 DSP core 1-3
dimensions
12-pin header C-20
14-pin header C-14
mechanical, 14-pin header C-14
direct addressing
description 5-4
role of data page pointer (DP) 5-4
direct memory access mechanism for emulation 7-15
disallow access to emulation registers 6-99
discontinuity delay 4-11
DMA control register 7-23
DMA ID register 7-23
DMA registers (data logging) 7-23
DP 2-8
role in direct addressing 5-4
DP direct addressing mode 5-4
opcode information 5-23
DT-DMA mechanism 7-15
DT-DMA request process
figure 7-16
DuPont connector C-2

E

EALLOW bit 2-20
clear (EDIS instruction) 6-99
set (EALLOW instruction) 6-98
EALLOW instruction
description 6-98
use in data logging 7-21, 7-24
EDIS instruction
description 6-99
use in data logging 7-22, 7-25
embedded software breakpoint (ESTOP1) 6-101
EMU0/1
configuration C-21, C-23, C-24
emulation pins C-20
IN signals C-21

EMU0/1 (continued)
rising edge modification C-22
signals 7-4
EMU0/1 signals 7-4, C-2, C-3, C-6, C-7, C-13, C-18
emulation
configuring multiple processors C-13
data logging 7-21
disabled 7-5
enabled 7-5
features 1-4, 7-2
JTAG cable C-1
logic 1-3, 1-4, 7-14
pins C-20
timing calculations C-7 to C-9, C-18 to C-26
using scan path linkers C-16
emulation access enable bit (EALLOW) 2-20
emulation signals 1-5
emulation timing C-7
emulator
cable pod C-5
connection to target system, JTAG mechanical dimensions C-14 to C-25
emulation pins C-20
pod interface C-5
pod timings C-6
signal buffering C-10 to C-13
software breakpoints 6-100
target cable, header design C-2 to C-3
end address register (data logging) 7-24
endless loop
preventing 8-19
ESTOP0 instruction 6-100
ESTOP1 instruction 6-101
event counter 7-19
events
break 7-6
debug 7-6
examples
data logging with end address 7-27
data logging with word counter 7-26
loop instructions 8-17
opcode formats for instructions 5-21
execution control modes
real-time mode 7-9
stop mode 7-7
extended auxiliary register XAR6 2-10
as circular pointer 8-30
extended auxiliary register XAR7 2-10
as pointer to program space 5-15

F

fast function call 6-102, 8-5
FC (fetch counter) 4-5
fetch counter (FC) 4-5
FFC instruction 6-102, 8-5
flag tests done by conditional branch instructions 8-7
flags
checking interrupt 8-16
interrupt flag register (IFR) 3-6
flow charts
handling DT-DMA request 7-16
interrupt initiated by the TRAP instruction 3-17
interrupt operation, maskable interrupts 3-11
foreground code 7-6

H

hardware reset 3-22
initializing pointers and status bits after 8-2
hardware reset interrupt 3-16
header
14-pin C-2
dimensions, 14-pin 7-3, C-2
high-impedance mode 7-5
how to change pointer values 8-2
how to set key status bits 8-2

I

IACK instruction 6-103, 8-10
IC (instruction counter) 4-5
IDLE instruction 6-104
IDLE status bit (IDLESTAT) 2-20
idle until interrupt 6-104
IDLESTAT bit 2-20
IEEE 1149.1 (JTAG) signals 7-3
IEEE 1149.1 specification
bus slave device rules C-4
illegal-instruction trap 3-16, 3-21, 6-114, 6-116
immediate addressing modes 5-2
immediate-constant addressing mode 5-2

immediate-pointer addressing mode 5-2
INC instruction 6-106
increment by 1 (INC instruction) 6-106
indirect addressing modes
operand summary 5-18
that use auxiliary registers 5-10
that use stack pointer (SP) 5-7
individually accessible portions of the accumulator 2-6
instruction counter (IC) 4-5
instruction descriptions 6-39
how to use 6-34
instruction set summaries 6-2
alphabetical summary 6-3
opcode summary 6-20
summary by operation type 6-9
instruction syntax formats 6-2
instruction trap 6-114
instruction-fetch mechanism 4-4
instruction-not-available condition 4-10
instructions
ABORTI (abort interrupt) 6-40
ABORTI (abort interrupt) 7-14
ABS (absolute value of ACC) 6-41
ADD (add value to specified location) 6-43
ADDB (add short value to specified register) 6-51
ADDCU (add unsigned value plus carry to ACC) 6-55
ADDL (add long value) 6-58
ADDU (add unsigned value to accumulator) 6-62
ADRK (add to current auxiliary register) 6-65
AND (bitwise AND) 6-66
ANDB (bitwise AND with short value) 6-71
ASP (align stack pointer) 6-72, 8-27
ASR (arithmetic shift right) 6-74
B (branch) 6-77
BANZ (branch if auxiliary register not equal to 0) 6-80
CALL 6-82
CALL *XAR7 5-15
CLRC (clear status bits) 6-84
CMP (compare) 6-87
CMPB (compare with short value) 6-91
CMPL (compare with long value) 6-93
conditional 2-24
data move operations 6-16
instructions (continued)
DEC (decrement specified value by 1) 6-96
EALLOW (allow access to emulation registers) 6-98
EDIS (disallow access to emulation register) 6-99
emulation operations 6-20
ESTOP0 (emulator software breakpoint) 6-100
ESTOP1 (embedded software breakpoint) 6-101
FFC (fast function call) 6-102
IACK (interrupt acknowledge) 6-103
IDLE (idle until interrupt) 6-104
INC (increment specified value by 1) 6-106
INTR (software interrupt) 3-16, 6-108
IRET (return from interrupt and restore register pairs) 6-113
ITRAP0 (instruction trap 0) 6-114
ITRAP1 (instruction trap 1) 6-116
LB (long branch) 6-118
LB *XAR7 5-15
LOOPNZ (loop while not zero) 6-119
LOOPZ (loop while zero) 6-122
LSL (logical shift left) 6-125
LSR (logical shift right) 6-129
MAC (multiply and accumulate, preload T) 4-16, 6-132
math operations 6-17
MOV (move) 1-10, 6-137
MOVA (load T register and add previous to ACC) 6-154
MOVB (move short value) 6-158
MOVH (store high word) 6-168
MOVL (move long value) 6-172
MOVP (load T register and load previous product to the accumulator) 6-176
MOVS (load T register and subtract previous product from ACC) 6-179
MOVU (load ACC with unsigned word) 6-183
MOVW (load entire DP) 6-185
MPY (multiply) 6-186
MPYA (multiply and accumulate previous product) 6-190
MPYB (multiply signed value by unsigned short value) 6-195
MPYS (multiply and subtract previous product) 6-197
MPYU (unsigned multiply) 6-201
MPYXU (multiply signed value by unsigned value) 6-204

instructions (continued)
NASP (unalign stack pointer) 6-207, 8-29
NEG (negative of accumulator value) 6-208
NOP (no operation) 6-212
NORM (normalize ACC value) 6-214
NOT (complement of ACC value) 6-217
OR (bitwise OR) 6-219
ORB (bitwise OR with short value) 6-223
POP (load from stack) 6-224
PREAD (read from program memory) 1-10, 4-16, 5-15, 6-233, 8-21
program flow operations 6-16
PUSH (save on stack) 6-236
PWRITE (write to program memory) 1-10, 4-16, 5-15, 6-244, 8-21
RET (return) 6-247
RETE (return with interrupts enabled) 6-248
ROL (rotate accumulator left) 6-249
ROR (rotate accumulator right) 6-250
RPT (repeat next instruction) 6-251
SAT (saturate accumulator) 6-254
SB (short branch) 6-257
SBBU (subtract unsigned value plus inverse borrow from ACC) 6-260
SBRK (subtract from current auxiliary register) 6-263
SETC (set status bits) 6-264
SFR (shift accumulator right) 6-267
SPM (set product shift mode) 6-271
SUB (subtract value from specified location) 6-273
SUBB (subtract short value from specified register) 6-280
SUBCU (conditional subtraction) 6-284
SUBL (subtract long value from accumulator) 6-288
SUBU (subtract unsigned value from accumulator) 6-292
SXTB (sign extend LSByte of ACC half) 6-296
TBIT (test specified bit) 6-298
TEST (test for accumulator equal to zero) 6-301
TRAP (software trap) 3-16, 6-303
XOR (bitwise exclusive OR) 6-307
XORB (bitwise exclusive OR with short value) 6-310

interface
memory 1-11

interrupt
signals 1-5

interrupt acknowledge instruction 6-103, 8-10

interrupt-control registers (IFR, IER, DBGIER) 2-12

interrupt enable register (IER) 3-5, 3-7, 7-9, 8-14
quick reference figure A-7

interrupt flag register (IFR) 3-6
checking flags 8-16
quick reference figure A-6

interrupt global mask bit (INTM) 2-23, 3-5, 6-104, 7-9, 8-14

interrupt handling in real-time mode 7-9

interrupt handling in stop mode 7-7

interrupt instructions
AND IER 3-7
AND IFR 3-6
INTR 3-7, 3-16
OR IER 3-7
OR IFR 3-6
POP DBGIER 3-9
PUSH DBGIER 3-9
TRAP 3-16

interrupt service routine (ISR) 3-3, 8-9
caution about breakpoints 7-11

interrupts 2-24, 3-1
aborting 7-14
accessing values saved 8-12
assigning vectors 8-9
checking interrupt flags 8-16
control registers (IFR, IER, DBGIER) 2-12
data log interrupt (DLOGINT) 3-5, 7-25
effect on IDLE instruction 6-104
effect on instructions in pipeline 4-4
general purpose 3-5
handling information by emulation mode and state 7-13
IACK instruction 6-103, 8-10
$\overline{\text{INT1}}$–$\overline{\text{INT14}}$ 3-5
INTR instruction 6-108
IRET instruction 6-113
ITRAP0 instruction 6-114
ITRAP1 instruction 6-116
managing 8-9
maskable 3-5
definition 3-2
flow chart of operation 3-11
nested 8-14
$\overline{\text{NMI}}$ 3-20
nonmaskable 3-16
definition 3-2

interrupts (continued)
operation
overview 3-2
real-time mode 7-9
standard 3-10
stop mode 7-7
overview 3-2
real-time operating system interrupt (RTOSINT) 3-5
special cases, clearing IFR flag bit after TRAP instruction 3-6, 3-7
time-critical 7-6
serviced in real-time mode 7-9
vectors 3-3
INTR instruction 3-16, 6-108
IRET instruction 6-40, 6-113, 7-14
ITRAP0 instruction 6-114
ITRAP1 instruction 6-116

J

JTAG C-16
signals 7-3
JTAG emulator
buffered signals C-10
no signal buffering C-10
JTAG header to interface a target to the scan controller
figure 7-3

L

LB *XAR7 instruction 8-5
LB instruction 6-118
load from stack 6-224
load T register and add previous product to ACC 6-154
load T register and load previous product to ACC 6-176
load T register and subtract previous product from ACC 6-179
logical shift left 6-125
logical shift right 6-129
long branch 6-118
LOOP bit 2-20
loop instruction status bit (LOOP) 2-20
loop while not zero instruction 6-119
loop while zero instruction 6-122
LOOPNZ instruction 6-119, 8-17
LOOPZ instruction 6-122, 8-17
LSL instruction 6-125
LSR instruction 6-129

M

MAC instruction 6-132
maskable interrupts 3-5
definition 3-2
flow chart of operation 3-11
math operations 6-17
memory 1-11
address map 1-6
interface 1-11
managing 8-20
map 1-6
map for block B0 1-7
map for block B1 1-9
reserved addresses 1-6, 1-7
memory map diagram 1-6
mixing 16- and 32-bit values on the stack 8-26
modes
high-impedance 7-5
nonpreemptive 7-15
normal with emulation disabled 7-5
normal with emulation enabled 7-5
preemptive 7-15
real-time 7-7, 7-9
slave 7-5
stop 7-7
MOV instruction 6-137
MOVA instruction 6-154
MOVB instruction 6-158, 8-22
move high word 6-168
move instructions
MOV (move) 6-137
MOVA (load T register and add previous product to ACC) 6-154
MOVB (move short value) 6-158
MOVH (store high word) 6-168
MOVL (move long value) 6-172
MOVP (load T register and load previous product to ACC 6-176
MOVS (load T register and subtract previous product from ACC) 6-179
MOVU (load ACC with unsigned word) 6-183

move instructions (continued)
MOVW (load entire DP) 6-185
PREAD (read from program space) 6-233
PWRITE (write to program memory) 6-244
move long value 6-172
move short value 6-158
move value 6-137
MOVH instruction 6-168
moving block of data 8-20
MOVL instruction 8-13
MOVP instruction 6-176
MOVS instruction 6-179
MOVU instruction 6-183
MOVW instruction 6-185
MPY instruction 6-186
MPYA instruction 6-190
MPYB instruction 6-195
MPYS instruction 6-197
MPYU instruction 6-201
MPYXU instruction 6-204
multiplicand register (T) 2-7
multiplier
introduction 2-2
operation 2-26
multiply and accumulate instructions
MAC (multiply and accumulate, preload T) 6-132
MPYA (multiply and accumulate previous product) 6-190
MPYS (multiply and subtract previous product) 6-197
multiply instructions
MAC (multiply and accumulate, preload T) 6-132
MPY (multiply) 6-186
MPYA (multiply and accumulate) 6-190
MPYB (signed by unsigned multiply short value) 6-195
MPYS (multiply and subtract previous product) 6-197
MPYU (unsigned multiply) 6-201
MPYXU (multiply signed value by unsigned value) 6-204

N

N bit 2-15
NASP instruction 6-207, 8-29
NEG instruction 6-208
negative flag (N) 2-15
nested interrupts 8-14
$\overline{\text{NMI}}$ interrupt 3-20
initiating by using INTR instruction 6-108
$\overline{\text{NMI}}$ pin 3-20
nonmaskable interrupts 3-16
definition 3-2
nonpreemptive mode 7-15
NOP instruction 6-212
NORM instruction 6-214
normal mode 7-5
NOT instruction 6-217

O

opcode summary of instructions 6-20
opcodes
addressing-mode information 5-21
immediate addressing modes 5-21
register, direct, and indirect addressing modes 5-23
operands
ARP indirect addressing 5-14
auxiliary-register indirect addressing 5-12
for moving data 8-20
summary of indirect-addressing 5-18
operating modes
selecting by using $\overline{\text{TRST}}$, EMU0, and EMU1 7-5
operations
ACC 6-13
ACC 32-bit 6-15
address register 6-10
AH/AL 6-12
AH/AL byte 6-13
branch 8-4
control 6-18
data move 6-16
math 6-17
multiply 2-26
on memory or register 6-15
program flow 6-16
shift 2-27
special bus 1-12
stack 2-10, 6-10

OR instruction 6-219
ORB instruction 6-223
output modes
  external count C-20
  signal event C-20
OVC (overflow counter) 2-13
  used by SAT instruction 6-254
overflow counter (OVC) 2-13
  used by SAT instruction 6-254
overflow flag (V) 2-15
overflow mode bit (OVM) 2-17
OVM bit 2-17

P

P register 2-7
PAGE0 bit 2-21
PAGE0 direct addressing mode 5-5
  opcode information 5-23
PAGE0 stack addressing mode 5-8
  opcode information 5-23
pages of data memory
  figure 5-4
PAL C-21, C-22, C-24
PC (program counter) 1-10, 2-11, 4-5
phases of pipeline 4-2
pipeline 2-25
  decoupled segments 4-4
  freezes in activity 4-10
  instruction-fetch mechanism 4-4
  operations not protected by 4-16
  phases 4-2
  protection 4-12
  visualizing activity 4-7
  wait states 4-10
pipeline phases 4-2
PM bits 2-14
pointer values
  how to change 8-2
POP instruction 6-224
PREAD instruction 6-233, 8-21
preemptive mode 7-15
preventing an endless loop 8-19
process for handling a DT-DMA request
  figure 7-16
processor
  initializing after reset 8-2
product register (P) 2-7
product shift mode bits (PM) 2-14
program address bus (PAB) 1-11, 4-4
program address generation logic 2-2
program control logic 2-2
program counter (PC) 1-10, 2-11, 4-5
program flow 2-24
program flow operations 6-16
program space
  accessing 1-10
  address map 1-6
program-address counters 4-5
program-read data bus (PRDB) 1-11
protocol
  bus, in emulator system C-4
PUSH IFR instruction 8-16
PUSH instruction 6-236
PWRITE instruction 6-244, 8-21

R

read from program space 6-233
reads and writes
  unprotected 4-16
real-time mode 7-7, 7-9
  figure of execution states 7-10
  using nested interrupts in 8-14
real-time mode versus stop mode
  figure 7-12
real-time operating system interrupt (RTOSINT) 3-5, 7-13
register addressing mode 5-6
  opcode information 5-24
register quick reference A-1
  figures A-3
registers
  accumulator 2-5
  ADDRH 7-22
  ADDRL 7-22
  after reset 3-22
  allow access to emulation registers 6-98
  auxiliary register pointer (ARP) 5-13
  auxiliary registers (AR0–AR5, XAR6, XAR7) 2-10

registers (continued)
conflicts, protection against 4-13
CPU registers (summary) 2-4
data page pointer (DP) 2-8
debug interrupt enable register (DBGIER) 3-7
disallow access to emulation registers 6-99
DMA control register 7-23
end address register (data logging) 7-24
interrupt-control registers (IFR, IER, DBGIER) 2-12
interrupt enable register (IER) 3-7
interrupt flag register (IFR) 3-6
memory-mapped emulation registers, access 6-99
multiplicand (T) 2-7
product register (P) 2-7
program counter (PC) 2-11
quick reference A-1
quick reference figures A-3
stack pointer (SP) 2-9
start address register (data logging) 7-23
status register ST0 2-12, 2-13
status register ST1 2-12, 2-19
T register 2-7
registers after reset 3-22
repeat counter (RPTC) 2-24
repeat instruction (RPT) 6-251
used to move blocks of data 8-20
reserved addresses 1-6, 1-7
reset and interrupt signals 1-5
reset input signal ($\overline{RS}$) 3-22
reset of CPU 3-22
initializing pointer and status bits after 8-2
reset of debug context when aborting interrupt 6-40
RET instruction 6-247
RETE instruction 6-248
return from interrupt and restore registers 6-113
return instructions
IRET (return from interrupt) 6-113
RET (return) 6-247
RETE (return with interrupts enabled) 6-248
return with interrupts enabled 6-248
returns 2-24
ROL instruction 6-249
ROM codes
submitting custom B-1
ROR instruction 6-250
rotate accumulator left 6-249
rotate accumulator right 6-250
RPT instruction 6-251
used to move blocks of data 8-20
RPTC (repeat counter) 2-24
run state 7-7, 7-10
run/stop operation C-10
RUNB_ENABLE
input C-22

S

SAT instruction 6-254
save on stack 6-236
SB instruction 6-257, 8-5
SBBU instruction 6-260
SBRK instruction 6-263
scan path linkers C-16
secondary JTAG scan chain to an SPL C-17
suggested timings C-22
usage C-16
scan paths
TBC emulation connections for JTAG scan paths C-25
selecting device operating modes 7-5
set product shift mode 6-271
SETC instruction 6-264
SETC PAGE0 instruction 8-3
SETC VMAP instruction 8-3
setting key status bits 8-3
setting pointers 8-2
SFR instruction 6-267
shift accumulator right 6-267
shift instructions
ASR (arithmetic shift right) 6-74
LSL (logical shift left) 6-125
LSR (logical shift right) 6-129
SFR (shift accumulator right) 6-267
shift operations 2-27
shifter 1-4, 2-2
shifting values in the accumulator 2-6
sign extend LSByte of ACC half 6-296
signal descriptions
14-pin header 7-4, C-3

signals 1-5
buffered C-10
buffering for emulator connections C-10 to C-13
description, 14-pin header 7-4, C-3
EMU0 7-5
EMU1 7-5
PD (VCC) 7-3
TCK 7-3
TCK_RET 7-3
TDI 7-3
TDO 7-3
timing C-6
TMS 7-3
TRST 7-3, 7-5
sign-extension mode bit (SXM) 2-18
single-instruction state 7-7
slave devices C-4
slave mode 7-5
software interrupts 3-16
software trap 6-303
SPA bit 2-20
special branch operations 8-4
SPM instruction 6-271
stack 2-9
allocating temporary space on 8-26
stack operations 6-10
stack pointer (SP) 1-10, 2-9
align 6-72
indirect addressing modes that use 5-7
unalign 6-207
use as base pointer in array 8-23
stack pointer alignment bit (SPA) 2-20
stack-pointer indirect addressing mode 5-7
opcode information 5-24
start address register (data logging) 7-23
status bits
ARP 2-19
C 2-16
clear 6-84
DBGM 2-22
EALLOW 2-20
IDLESTAT 2-20
INTM 2-23
LOOP 2-20
N 2-15
OVC 2-13
status bits (continued)
OVM 2-17
PAGE0 2-21
PM 2-14
set 6-264
SPA 2-20
SXM 2-18
TC 2-17
V 2-15
VMAP 2-21
Z 2-16
status registers
ST0 2-12, 2-13
quick reference figure A-4
ST1 2-12, 2-19
quick reference figure A-5
stop mode 7-7
figure of execution states 7-8
stop mode versus real-time mode
figure 7-12
SUB instruction 6-273
SUBB instruction 6-280
SUBCU ACC instruction 6-284
SUBL instruction 6-288
submitting ROM codes to TI B-1
subtract from current auxiliary register 6-263
subtract instructions
SBBU (subtract unsigned value plus inverse borrow from ACC) 6-260
SBRK (subtract from current auxiliary register) 6-263
SUB (subtract value from specified location) 6-273
SUBB (subtract short value from specified register) 6-280
SUBCU (conditional subtraction) 6-284
SUBL (subtract long value from accumulator) 6-288
SUBU (subtract unsigned value from accumulator) 6-292
subtract long value from accumulator 6-288
subtract short value from specified register 6-280
subtract unsigned value from accumulator 6-292
subtract value from specified location 6-273
SUBU instruction 6-292
SXM bit 2-18
SXTB instruction 6-296
syntax formats 6-2

T

T register 2-7
T320C2700 core 1-2
target cable C-14
target system emulator connector
  designing C-2
target-system clock C-12
TBIT instruction 6-298
TC bit 2-17
TCK signal 7-4, C-2, C-3, C-4, C-6, C-7, C-13, C-17, C-18, C-25
TDI signal 7-4, C-2 to C-8, C-13, C-18
TDO signal C-4, C-5, C-8, C-19, C-25
temporary space on the stack 8-26
terminology
  debug 7-6
test
  sharing resources 7-28
TEST instruction 6-301
test bit 6-298
test bus controller C-22, C-24
test clock C-12
  diagram C-12
test clock return signal (TCK_RET) 7-3
test for accumulator equal to zero 6-301
test specified bit 6-298
test/control flag bit (TC) 2-17, 8-16
testing and debugging
  signals 1-5
time-critical interrupts
  definition 7-6
  serviced in real-time mode 7-9
timing calculations C-7 to C-9, C-18 to C-26
TMS signal 7-4, C-2 to C-8, C-13, C-17, C-18, C-19, C-25
TMS/TDI inputs C-4
transfer a block of data 8-20
TRAP instruction 3-17, 6-303
$\overline{\text{TRST}}$ signal 7-4, 7-5, C-2, C-3, C-6, C-7, C-13, C-17, C-18, C-25

U

unalign stack pointer instruction (NASP) 6-207, 8-29
unprotected program-space reads and writes 4-16

V

V bit 2-15
vector map bit (VMAP) 2-21

W

wait states
  effects on pipeline 4-10
wait-in-reset mode 7-5
watchpoints 7-18
write to program memory 6-244

X

XAR6 register 2-10
  as circular pointer 8-30
XAR7 indirect addressing mode 5-15
XAR7 register 2-10
  as pointer to program space 5-15
XOR instruction 6-307
XORB instruction 6-310

Z

zero flag bit (Z) 2-16