TMS320C54x DSP Reference Set

Volume 1: CPU and Peripherals

Literature Number: SPRU131E
June 1998

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.

Preface

Read This First

About This Manual

The TMS320C54x is a fixed-point digital signal processor (DSP) in the TMS320 family. This book is the first volume of a 4-volume set and serves as a reference for the TMS320C54x DSP and provides information for developing hardware and software applications using the ’54x. Unless otherwise specified, all references to the ’54x apply to the TMS320C54x, as well as the TMS320LC54x and the TMS320VC54x.

For a summary of updates in this book, see Appendix F, *Summary of Updates in This Document*.

How to Use This Manual

The table below summarizes the ’54x information contained in this book. (For a complete listing, refer to the Table of Contents.)

If you are looking for information about:	Turn to these chapters:
Addressing modes	Chapter 5, *Data Addressing* Chapter 6, *Program Memory Addressing*
Boot loader	Chapter 3, *Memory*
Buffered serial port	Chapter 9, *Serial Ports*
Bus structure	Chapter 2, *Architectural Overview*
Changes in this document	Appendix F, *Summary of Updates in This Document*
Clock generator	Chapter 2, *Architectural Overview* Chapter 8, *On-Chip Peripherals*
CPU architecture	Chapter 2, *Architectural Overview* Chapter 4, *Central Processing Unit*

If you are looking for information about:	Turn to these chapters:
External bus	Chapter 10, *External Bus Operation*
Hold mode	Chapter 10, *External Bus Operation*
Host port interface	Chapter 8, *On-Chip Peripherals*
Interrupts	Chapter 6, *Program Memory Addressing*
Memory	Chapter 2, *Architectural Overview* Chapter 3, *Memory*
On-chip peripherals	Chapter 8, *On-Chip Peripherals*
Overview of the ’54x	Chapter 1, *Introduction*
Parallel I/O Ports	Chapter 2, *Architectural Overview* Chapter 8, *On-Chip Peripherals*
Power-down modes	Chapter 6, *Program Memory Addressing*
Program control	Chapter 6, *Program Memory Addressing*
Pipeline latencies	Chapter 2, *Architectural Overview* Chapter 7, *Pipeline*
Reset	Chapter 6, *Program Memory Addressing*
ROM code submission to TI	Appendix D, *Submitting ROM Codes to TI*
Serial ports	Chapter 10, *Serial Ports*
Status registers	Chapter 4, *Central Processing Unit* Appendix A, *CPU and Peripheral Registers*
Summary of updates in this document	Appendix F, *Summary of Updates in This Document*
TDM serial port	Chapter 10, *Serial Ports*
Timer	Chapter 2, *Architectural Overview* Chapter 8, *On-Chip Peripherals*
Wait-state generator	Chapter 2, *Architectural Overview* Chapter 8, *On-Chip Peripherals*

Notational Conventions

This book uses the following conventions.

- The TMS320C54x DSP can use either of two forms of the instruction set: a mnemonic form or an algebraic form. This book uses the mnemonic form of the instruction set. For information about the mnemonic form of the instruction set, see *TMS320C54x DSP Reference Set, Volume 2: Mnemonic Instruction Set*. For information about the algebraic form of the instruction set, see *TMS320C54x DSP Reference Set, Volume 3: Algebraic Instruction Set*.

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

  Here is a segment of a program listing:

```
STL    A,*AR1+      ;Int_RAM(I)=0
RSBX   INTM         ;Globally enable interrupts
B      MAIN_PG      ;Return to foreground program
```

- Square brackets, [and], identify an optional parameter. If you use an optional parameter, specify the information within the brackets; do not type the brackets themselves.

Information About Cautions

This book contains cautions.

This is an example of a caution statement.

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

The information in a caution is provided for your protection. Please read each caution carefully.

Related Documentation from Texas Instruments

The following books describe the ’54x and related support tools. To obtain a copy of any of these TI documents, call the Texas Instruments Literature Response Center at (800) 477-8924. When ordering, please identify the book by its title and literature number. Many of these documents are located on the internet at http://www.ti.com. Click DSPS Solutions, then click DSP Literature.

TMS320C54x DSP Reference Set is composed of four volumes that can be ordered as a set with literature number SPRU210. To order an individual book, use the document-specific literature number:

TMS320C54x DSP Reference Set, Volume 1: CPU and Peripherals (literature number SPRU131) describes the TMS320C54x 16-bit, fixed-point, general-purpose digital signal processors. Covered are its architecture, internal register structure, data and program addressing, the instruction pipeline, and on-chip peripherals. Also includes development support information, parts lists, and design considerations for using the XDS510 emulator.

TMS320C54x DSP Reference Set, Volume 2: Mnemonic Instruction Set (literature number SPRU172) describes the TMS320C54x digital signal processor mnemonic instructions individually. Also includes a summary of instruction set classes and cycles.

TMS320C54x DSP Reference Set, Volume 3: Algebraic Instruction Set (literature number SPRU179) describes the TMS320C54x digital signal processor algebraic instructions individually. Also includes a summary of instruction set classes and cycles.

TMS320C54x DSP Reference Set, Volume 4: Applications Guide (literature number SPRU173) describes software and hardware applications for the TMS320C54x digital signal processor. Also includes development support information, parts lists, and design considerations for using the XDS510 emulator.

TMS320C54x, TMS320LC54x, TMS320VC54x Fixed-Point Digital Signal Processors (literature number SPRS039) data sheet contains the electrical and timing specifications for these devices, as well as signal descriptions and pinouts for all of the available packages.

TMS320C54x DSKplus User’s Guide (literature number SPRU191) describes the TMS320C54x digital signal processor starter kit (DSK), which allows you to execute custom ’C54x code in real time and debug it line by line. Covered are installation procedures, a description of the debugger and the assembler, customized applications, and initialization routines.

TMS320C54x Assembly Language Tools User's Guide (literature number SPRU102) describes the assembly language tools (assembler, linker, and other tools used to develop assembly language code), assembler directives, macros, common object file format, and symbolic debugging directives for the 'C54x generation of devices.

TMS320C5xx C Source Debugger User's Guide (literature number SPRU099) tells you how to invoke the 'C54x emulator, evaluation module, and simulator versions of the C source debugger interface. This book discusses various aspects of the debugger interface, including window management, command entry, code execution, data management, and breakpoints. It also includes a tutorial that introduces basic debugger functionality.

TMS320C54x Code Generation Tools Getting Started Guide (literature number SPRU147) describes how to install the TMS320C54x assembly language tools and the C compiler for the 'C54x devices. The installation for MS-DOS™, OS/2™, SunOS™, Solaris™, and HP-UX™ 9.0x systems is covered.

TMS320C54x Evaluation Module Technical Reference (literature number SPRU135) describes the 'C54x evaluation module, its features, design details and external interfaces.

TMS320C54x Optimizing C Compiler User's Guide (literature number SPRU103) describes the 'C54x C compiler. This C compiler accepts ANSI standard C source code and produces TMS320 assembly language source code for the 'C54x generation of devices.

TMS320C54x Simulator Getting Started (literature number SPRU137) describes how to install the TMS320C54x simulator and the C source debugger for the 'C54x. The installation for MS-DOS™, PC-DOS™, SunOS™, Solaris™, and HP-UX™ systems is covered.

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

TMS320C548/C549 Bootloader Technical Reference (literature number SPRU288) describes the process the bootloader uses to transfer user code from an external source to the program memory at power up. (Presently available only on the internet.)

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

Technical Articles

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

- ❑ General-Purpose DSP
- ❑ Graphics/Imagery
- ❑ Speech/Voice
- ❑ Control
- ❑ Multimedia
- ❑ Military
- ❑ Telecommunications
- ❑ Automotive
- ❑ Consumer
- ❑ Medical
- ❑ Development Support

In the following list, references appear in alphabetical order according to author. The documents contain beneficial information regarding designs, operations, and applications for signal-processing systems; all of the documents provide additional references. Texas Instruments strongly suggests that you refer to these publications.

***General-Purpose DSP**:*

1) Chassaing, R., Horning, D.W., *"Digital Signal Processing with Fixed and Floating-Point Processors"*, CoED, USA, Volume 1, Number 1, pages 1-4, March 1991.

2) Defatta, David J., Joseph G. Lucas, and William S. Hodgkiss, *Digital Signal Processing: A System Design Approach*, New York: John Wiley, 1988.

3) Erskine, C., and S. Magar, "Architecture and Applications of a Second-Generation Digital Signal Processor," *Proceedings of IEEE International Conference on Acoustics, Speech, and Signal Processing*, USA, 1985.

4) Essig, D., C. Erskine, E. Caudel, and S. Magar, "A Second-Generation Digital Signal Processor," *IEEE Journal of Solid-State Circuits*, USA, Volume SC-21, Number 1, pages 86-91, February 1986.

5) Frantz, G., K. Lin, J. Reimer, and J. Bradley, "The Texas Instruments TMS320C25 Digital Signal Microcomputer," *IEEE Microelectronics*, USA, Volume 6, Number 6, pages 10-28, December 1986.

6) Gass, W., R. Tarrant, T. Richard, B. Pawate, M. Gammel, P. Rajasekaran, R. Wiggins, and C. Covington, "Multiple Digital Signal Processor Environment for Intelligent Signal Processing," *Proceedings of the IEEE, USA*, Volume 75, Number 9, pages 1246-1259, September 1987.

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

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

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

10) Lin, K., G. Frantz, and R. Simar, Jr., "The TMS320 Family of Digital Signal Processors," *Proceedings of the IEEE*, USA, Volume 75, Number 9, pages 1143-1159, September 1987.

11) Lovrich, A., Reimer, J., *"An Advanced Audio Signal Processor"* , Digest of Technical Papers for 1991 International Conference on Consumer Electronics, June 1991.

12) Magar, S., D. Essig, E. Caudel, S. Marshall and R. Peters, "An NMOS Digital Signal Processor with Multiprocessing Capability," *Digest of IEEE International Solid-State Circuits Conference*, USA, February 1985.

13) Oppenheim, Alan V., and R.W. Schafer, *Digital Signal Processing*, Englewood Cliffs, NJ: Prentice-Hall, Inc., 1975 and 1988.

14) Papamichalis, P.E., and C.S. Burrus, "Conversion of Digit-Reversed to Bit-Reversed Order in FFT Algorithms," *Proceedings of ICASSP 89*, USA, pages 984-987, May 1989.

15) Papamichalis, P., and R. Simar, Jr., "The TMS320C30 Floating-Point Digital Signal Processor," *IEEE Micro Magazine*, USA, pages 13-29, December 1988.

16) Papamichalis, P.E., "FFT Implementation on the TMS320C30," *Proceedings of ICASSP 88*, USA, Volume D, page 1399, April 1988.

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

18) Peterson, C., Zervakis, M., Shehadeh, N., *"Adaptive Filter Design and Implementation Using the TMS320C25 Microprocessor"* , Computers in

Education Journal, USA, Volume 3, Number 3, pages 12-16, July-September 1993.

19) Prado, J., and R. Alcantara, "A Fast Square-Rooting Algorithm Using a Digital Signal Processor," *Proceedings of IEEE*, USA, Volume 75, Number 2, pages 262-264, February 1987.

20) Rabiner, L.R. and B. Gold, *Theory and Applications of Digital Signal Processing*, Englewood Cliffs, NJ: Prentice-Hall, Inc., 1975.

21) Simar, Jr., R., and A. Davis, "The Application of High-Level Languages to Single-Chip Digital Signal Processors," *Proceedings of ICASSP 88*, USA, Volume D, page 1678, April 1988.

22) Simar, Jr., R., T. Leigh, P. Koeppen, J. Leach, J. Potts, and D. Blalock, "A 40 MFLOPS Digital Signal Processor: the First Supercomputer on a Chip," *Proceedings of ICASSP 87*, USA, Catalog Number 87CH2396-0, Volume 1, pages 535-538, April 1987.

23) Simar, Jr., R., and J. Reimer, "The TMS320C25: a 100 ns CMOS VLSI Digital Signal Processor," *1986 Workshop on Applications of Signal Processing to Audio and Acoustics*, September 1986.

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

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

***Graphics/Imagery**:*

1) Reimer, J., and A. Lovrich, "Graphics with the TMS32020," *WESCON/85 Conference Record*, USA, 1985.

***Speech/Voice**:*

1) DellaMorte, J., and P. Papamichalis, "Full-Duplex Real-Time Implementation of the FED-STD-1015 LPC-10e Standard V.52 on the TMS320C25," *Proceedings of SPEECH TECH 89*, pages 218-221, May 1989.

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

3) Frantz, G.A., and K.S. Lin, "A Low-Cost Speech System Using the TMS320C17," *Proceedings of SPEECH TECH '87*, pages 25-29, April 1987.

4) Papamichalis, P., and D. Lively, "Implementation of the DOD Standard LPC-10/52E on the TMS320C25," *Proceedings of SPEECH TECH '87*, pages 201-204, April 1987.

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

6) Pawate, B.I., and G.R. Doddington, "Implementation of a Hidden Markov Model-Based Layered Grammar Recognizer," *Proceedings of ICASSP 89*, USA, pages 801-804, May 1989.

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

8) Reimer, J.B. and K.S. Lin, "TMS320 Digital Signal Processors in Speech Applications," *Proceedings of SPEECH TECH '88*, April 1988.

9) Reimer, J.B., M.L. McMahan, and W.W. Anderson, "Speech Recognition for a Low-Cost System Using a DSP," *Digest of Technical Papers for 1987 International Conference on Consumer Electronics*, June 1987.

***Control**:*

1) Ahmed, I., "16-Bit DSP Microcontroller Fits Motion Control System Application," *PCIM*, October 1988.

2) Ahmed, I., "Implementation of Self Tuning Regulators with TMS320 Family of Digital Signal Processors," *MOTORCON '88*, pages 248-262, September 1988.

3) Allen, C. and P. Pillay, *"TMS320 Design for Vector and Current Control of AC Motor Drives"*, Electronics Letters, UK, Volume 28, Number 23, pages 2188-2190, November 1992.

4) Panahi, I. and R. Restle, *"DSPs Redefine Motion Control"*, Motion Control Magazine, December 1993.

5) Lovrich, A., G. Troullinos, and R. Chirayil, "An All-Digital Automatic Gain Control," *Proceedings of ICASSP 88*, USA, Volume D, page 1734, April 1988.

6) Ahmed, I., and S. Meshkat, "Using DSPs in Control," *Control Engineering*, February 1988.

7) Meshkat, S., and I. Ahmed, "Using DSPs in AC Induction Motor Drives," *Control Engineering*, February 1988.

8) Matsui, N. and M. Shigyo, *"Brushless DC Motor Control Without Position and Speed Sensors"* , IEEE Transactions on Industry Applications, USA, Volume 28, Number 1, Part 1, pages 120-127, January-February 1992.

9) Hanselman, H., "LQG-Control of a Highly Resonant Disc Drive Head Positioning Actuator," *IEEE Transactions on Industrial Electronics*, USA, Volume 35, Number 1, pages 100-104, February 1988.

10) Bose, B.K., and P.M. Szczesny, "A Microcomputer-Based Control and Simulation of an Advanced IPM Synchronous Machine Drive System for Electric Vehicle Propulsion," *Proceedings of IECON '87*, Volume 1, pages 454-463, November 1987.

11) Ahmed, I., and S. Lindquist, "Digital Signal Processors: Simplifying High-Performance Control," *Machine Design*, September 1987.

***Multimedia**:*

1) Reimer, J., *"DSP-Based Multimedia Solutions Lead Way Enhancing Audio Compression Performance"* , Dr. Dobbs Journal, December 1993.

2) Reimer, J., G. Benbassat, and W. Bonneau Jr., *"Application Processors: Making PC Multimedia Happen"* , Silicon Valley PC Design Conference, July 1991.

***Military**:*

1) Papamichalis, P., and J. Reimer, "Implementation of the Data Encryption Standard Using the TMS32010," *Digital Signal Processing Applications*, 1986.

***Telecommunications**:*

1) Ahmed, I., and A. Lovrich, "Adaptive Line Enhancer Using the TMS320C25," *Conference Records of Northcon/86*, USA, 14/3/1-10, September/October 1986.

2) Casale, S., R. Russo, and G. Bellina, "Optimal Architectural Solution Using DSP Processors for the Implementation of an ADPCM Transcoder," *Proceedings of GLOBECOM '89*, pages 1267-1273, November 1989.

3) Cole, C., A. Haoui, and P. Winship, "A High-Performance Digital Voice Echo Canceller on a SINGLE TMS32020," *Proceedings of ICASSP 86*, USA, Catalog Number 86CH2243-4, Volume 1, pages 429-432, April 1986.

4) Cole, C., A. Haoui, and P. Winship, "A High-Performance Digital Voice Echo Canceller on a Single TMS32020," *Proceedings of IEEE International Conference on Acoustics, Speech and Signal Processing*, USA, 1986.

5) Lovrich, A., and J. Reimer, "A Multi-Rate Transcoder," *Transactions on Consumer Electronics*, USA, November 1989.

6) Lovrich, A. and J. Reimer, *"A Multi-Rate Transcoder"* , Digest of Technical Papers for 1989 International Conference on Consumer Electronics, June 7-9, 1989.

7) Lu, H., D. Hedberg, and B. Fraenkel, "Implementation of High-Speed Voice-band Data Modems Using the TMS320C25," *Proceedings of ICASSP 87*, USA, Catalog Number 87CH2396-0, Volume 4, pages 1915-1918, April 1987.

8) Mock, P., "Add DTMF Generation and Decoding to DSP- µP Designs," *Electronic Design*, USA, Volume 30, Number 6, pages 205-213, March 1985.

9) Reimer, J., M. McMahan, and M. Arjmand, "ADPCM on a TMS320 DSP Chip," *Proceedings of SPEECH TECH 85*, pages 246-249, April 1985.

10) Troullinos, G., and J. Bradley, "Split-Band Modem Implementation Using the TMS32010 Digital Signal Processor," *Conference Records of Electro/86 and Mini/Micro Northeast*, USA, 14/1/1-21, May 1986.

***Automotive**:*

1) Lin, K., "Trends of Digital Signal Processing in Automotive," *International Congress on Transportation Electronic (CONVERGENCE '88*), October 1988.

***Consumer**:*

1) Frantz, G.A., J.B. Reimer, and R.A. Wotiz, "Julie, The Application of DSP to a Product," *Speech Tech Magazine*, USA, September 1988.

2) Reimer, J.B., and G.A. Frantz, "Customization of a DSP Integrated Circuit for a Customer Product," *Transactions on Consumer Electronics*, USA, August 1988.

3) Reimer, J.B., P.E. Nixon, E.B. Boles, and G.A. Frantz, "Audio Customization of a DSP IC," *Digest of Technical Papers for 1988 International Conference on Consumer Electronics*, June 8-10 1988.

***Medical**:*

1) Knapp and Townshend, "A Real-Time Digital Signal Processing System for an Auditory Prosthesis," *Proceedings of ICASSP 88*, USA, Volume A, page 2493, April 1988.

2) Morris, L.R., and P.B. Barszczewski, "Design and Evolution of a Pocket-Sized DSP Speech Processing System for a Cochlear Implant and Other Hearing Prosthesis Applications," *Proceedings of ICASSP 88*, USA, Volume A, page 2516, April 1988.

***Development Support**:*

1) Mersereau, R., R. Schafer, T. Barnwell, and D. Smith, “A Digital Filter Design Package for PCs and TMS320,” *MIDCON/84 Electronic Show and Convention*, USA, 1984.

2) Simar, Jr., R., and A. Davis, “The Application of High-Level Languages to Single-Chip Digital Signal Processors,” *Proceedings of ICASSP 88*, USA, Volume 3, pages 1678-1681, April 1988.

Trademarks

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

Figures

Tables

Examples

Chapter 1

Introduction

The TMS320C54x devices are fixed-point digital signal processors (DSPs) in the TMS320 family. The ’54x meets the specific needs of real-time embedded applications, such as telecommunications. The ’54x central processing unit (CPU), with its modified Harvard architecture, features minimized power consumption and a high degree of parallelism. Also, the versatile addressing modes and instruction set improve the overall system performance.

Topic **Page**

1.1 TMS320 Family Overview

The TMS320 family consists of fixed-point, floating-point, and multiprocessor digital signal processors (DSPs). The TMS320 architecture is designed specifically for real-time signal processing. The following characteristics make this family the ideal choice for a wide range of processing applications:

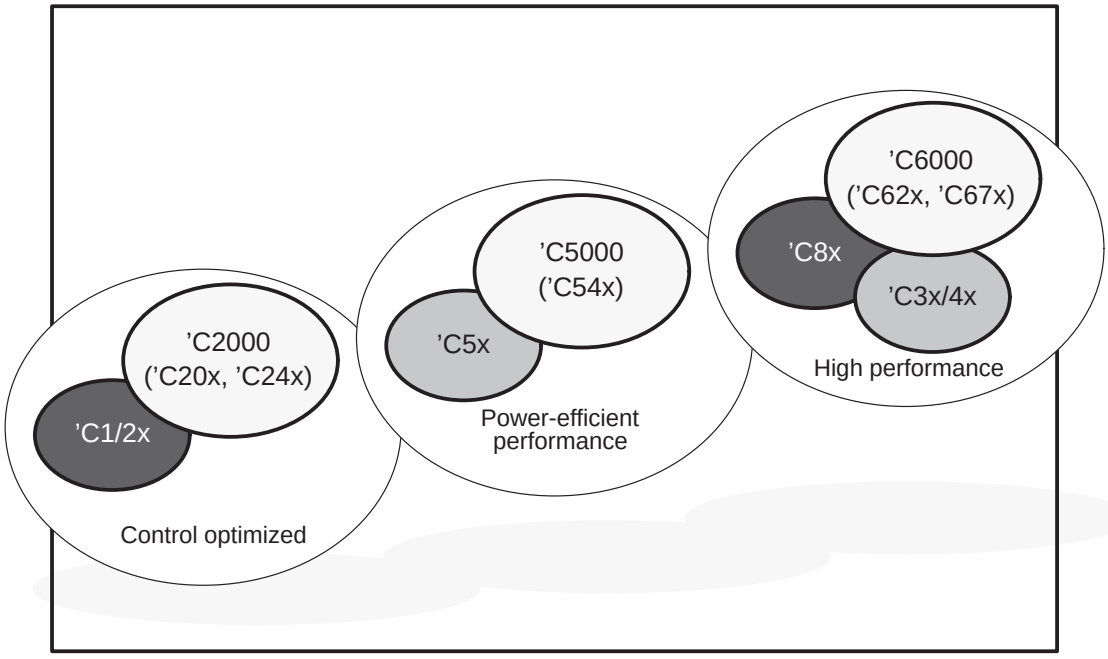
- ❏ Very flexible instruction set
- ❏ Inherent operational flexibility
- ❏ High-speed performance
- ❏ Innovative parallel architecture
- ❏ Cost-effectiveness
- ❏ C-friendly architecture

1.1.1 History, Development, and Advantages of TMS320 DSPs

In 1982, Texas Instruments introduced the TMS32010—the first fixed-point DSP in the TMS320 family. Before the end of the year, *Electronic Products* magazine awarded the TMS32010 the title “Product of the Year”. Today, the TMS320 family consists of these generations: ’C1x, ’C2x, ’C2xx, ’C5x, ’C54x, and ’C6x fixed-point DSPs; ’C3x and ’C4x floating-point DSPs; and ’C8x multiprocessor DSPs.

Devices within a generation of the TMS320 family have the same CPU structure but different on-chip memory and peripheral configurations. Spinoff devices use new combinations of on-chip memory and peripherals to satisfy a wide range of needs in the worldwide electronics market. By integrating memory and peripherals onto a single chip, TMS320 devices reduce system costs and save circuit board space. Figure 1–1 illustrates the performance gains that the TMS320 family has made.

Figure 1–1. Evolution of the TMS320 Family



1.1.2 Typical Applications for the TMS320 Family

Table 1–1 lists some typical applications for the TMS320 family of DSPs. The TMS320 DSPs offer more adaptable approaches to traditional signal-processing problems such as vocoding and filtering than standard microprocessor/microcomputer devices. They also support complex applications that often require multiple operations to be performed simultaneously.

Table 1–1. Typical Applications for the TMS320 DSPs

Automotive	Consumer	Control
Adaptive ride control	Digital radios/TVs	Disk drive control
Antiskid brakes	Educational toys	Engine control
Cellular telephones	Music synthesizers	Laser printer control
Digital radios	Pagers	Motor control
Engine control	Power tools	Robotics control
Navigation and global positioning	Radar detectors	Servo control
Vibration analysis	Solid-state answering machines	
Voice commands		
Anticollision radar		
General-Purpose	**Graphics/Imaging**	**Industrial**
Adaptive filtering	3-D rotation	Numeric control
Convolution	Animation/digital maps	Power-line monitoring
Correlation	Homomorphic processing	Robotics
Digital filtering	Image compression/transmission	Security access
Fast Fourier transforms	Image enhancement	
Hilbert transforms	Pattern recognition	
Waveform generation	Robot vision	
Windowing	Workstations	
Instrumentation	**Medical**	**Military**
Digital filtering	Diagnostic equipment	Image processing
Function generation	Fetal monitoring	Missile guidance
Pattern matching	Hearing aids	Navigation
Phase-locked loops	Patient monitoring	Radar processing
Seismic processing	Prosthetics	Radio frequency modems
Spectrum analysis	Ultrasound equipment	Secure communications
Transient analysis		Sonar processing
Telecommunications		**Voice/Speech**
1200- to 33 600-bps modems	Faxing	Speaker verification
Adaptive equalizers	Line repeaters	Speech enhancement
ADPCM transcoders	Personal communications systems (PCS)	Speech recognition
Cellular telephones	Personal digital assistants (PDA)	Speech synthesis
Channel multiplexing	Speaker phones	Speech vocoding
Data encryption	Spread spectrum communications	Text-to-speech
Digital PBXs	Video conferencing	Voice mail
Digital speech interpolation (DSI)	X.25 packet switching	
DTMF encoding/decoding		
Echo cancellation		

1.2 TMS320C54x Overview

The ’54x has a high degree of operational flexibility and speed. It combines an advanced modified Harvard architecture (with one program memory bus, three data memory buses, and four address buses), a CPU with application-specific hardware logic, on-chip memory, on-chip peripherals, and a highly specialized instruction set. Spinoff devices that combine the ’54x CPU with customized on-chip memory and peripheral configurations have been, and continue to be, developed for specialized areas of the electronics market.

The ’54x devices offer these advantages:

- ❏ Enhanced Harvard architecture built around one program bus, three data buses, and four address buses for increased performance and versatility
- ❏ Advanced CPU design with a high degree of parallelism and application-specific hardware logic for increased performance
- ❏ A highly specialized instruction set for faster algorithms and for optimized high-level language operation
- ❏ Modular architecture design for fast development of spinoff devices
- ❏ Advanced IC processing technology for increased performance and low power consumption
- ❏ Low power consumption and increased radiation hardness because of new static design techniques

1.3 TMS320C54x Key Features

This section lists the key features of the ’54x DSPs.

- CPU
  - Advanced multibus architecture with one program bus, three data buses, and four address buses
  - 40-bit arithmetic logic unit (ALU), including a 40-bit barrel shifter and two independent 40-bit accumulators
  - 17-bit × 17-bit parallel multiplier coupled to a 40-bit dedicated adder for nonpipelined single-cycle multiply/accumulate (MAC) operation
  - Compare, select, store unit (CSSU) for the add/compare selection of the Viterbi operator
  - Exponent encoder to compute the exponent of a 40-bit accumulator value in a single cycle
  - Two address generators, including eight auxiliary registers and two auxiliary register arithmetic units
- Memory
  - 192K words × 16-bit addressable memory space (64K-words program, 64K-words data, and 64K-words I/O), with extended program memory (8M words) in the ’548 and ’549.
  - On-chip configurations as follows (in K words):

Device	Program ROM	Program/Data ROM	DARAM†	SARAM‡
’541	20	8	5	0
’542	2	0	10	0
’543	2	0	10	0
’545	32	16	6	0
’546	32	16	6	0
’548	2	0	8	24
’549	16	16	8	24

† Dual-access RAM
‡ Single-access RAM

- Instruction set
  - Single-instruction repeat and block repeat operations
  - Block memory move instructions for better program and data management
  - Instructions with a 32-bit long operand
  - Instructions with 2- or 3-operand simultaneous reads
  - Arithmetic instructions with parallel store and parallel load
  - Conditional-store instructions
  - Fast return from interrupt
- On-chip peripherals
  - Software-programmable wait-state generator
  - Programmable bank switching
  - On-chip phase-locked loop (PLL) clock generator with internal oscillator or external clock source. With the external clock source, there are several multiplier values available from one of the following device options:

Option 1	Option 2	Option 3
1.0	1.0	Software-programmable PLL†
1.5	4.0	
2.0	4.5	
3.0	5.0	

† The ’541B, ’545A, ’546A, ’548, and ’549 are designated as LP-type devices. These devices have a software-programmable PLL and two additional saturation modes. The software-programmable PLL is described in subsection 8.4.2, *Software-Programmable PLL*, on page 8-19. The saturation modes are described in section 4.1.2, *Processor Mode Status Register (PMST)*, on page 4-6.

Each device offers selection of clock modes from one option list only.

  - External bus-off control to disable the external data bus, address bus, and control signals
  - Data bus with a bus holder feature
  - Programmable timer

- Ports:

Device	Host Port Interface	Serial Ports		
		Synchronous	Buffered	Time-Division Multiplexed
’541	0	2	0	0
’542	1	0	1	1
’543	0	0	1	1
’545	1	1	1	0
’546	0	1	1	0
’548	1	0	2	1
’549	1	0	2	1

- Speed: 25/20/15/12.5/10-ns† execution time for a single-cycle, fixed-point instruction (40 MIPS/50 MIPS/66 MIPS/80 MIPS/100 MIPS):

Device	Power Supply	Speed	Package
’541	5 V	25 ns	100-pin TQFP
	3 V / 3.3 V	25 ns/20 ns	100-pin TQFP
’541B	3 V / 3.3 V	15 ns	100-pin TQFP
’542	5 V	25 ns	144-pin TQFP
	3 V / 3.3 V	25 ns/20 ns	128-pin/144-pin TQFP
’543	3 V / 3.3 V	25 ns/20 ns	100-pin TQFP
’545	3 V / 3.3 V	25 ns/20 ns	128-pin TQFP
’545A	3 V / 3.3 V	15 ns	128-pin TQFP
’546	3 V / 3.3 V	25 ns/20 ns	100-pin TQFP
’546A	3 V / 3.3 V	15 ns	100-pin TQFP
’548	3.3 V	20 ns/15 ns	144-pin TQFP
’549	3.3 V	15 ns/12.5 ns	144-pin TQFP/144-pin Micro Star™ BGA
’VC549	3.3 V (2.5 core)	10 ns	144-pin TQFP/144-pin Micro Star™ BGA

- Power
  - Power consumption control with IDLE 1, IDLE 2, and IDLE 3 instructions for power-down modes
  - Control to disable the CLKOUT signal
- Emulation: IEEE Standard 1149.1 boundary scan logic interfaced to on-chip scan-based emulation logic

Chapter 2

Architectural Overview

This chapter provides an overview of the architectural structure of the ’54x, which comprises the central processing unit (CPU), memory, and on-chip peripherals.

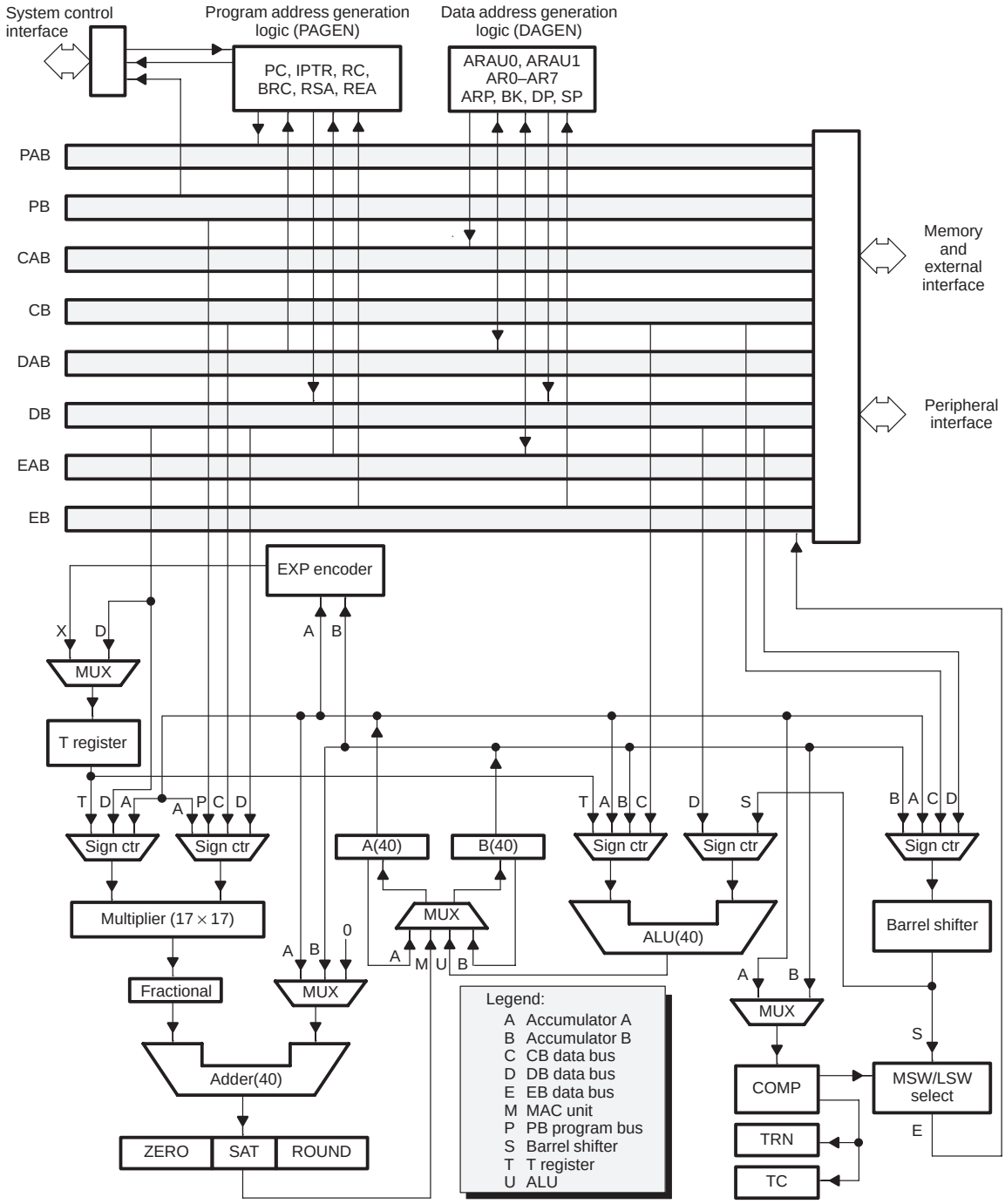
The ’54x DSPs use an advanced modified Harvard architecture that maximizes processing power with eight buses.

Separate program and data spaces allow simultaneous access to program instructions and data, providing a high degree of parallelism. For example, three reads and one write can be performed in a single cycle. Instructions with parallel store and application-specific instructions fully utilize this architecture. In addition, data can be transferred between data and program spaces. Such parallelism supports a powerful set of arithmetic, logic, and bit-manipulation operations that can all be performed in a single machine cycle. Also, the ’54x includes the control mechanisms to manage interrupts, repeated operations, and function calling.

Figure 2–1 is a ’54x functional block diagram, which includes the principal blocks and bus structure.

Topic **Page**

Figure 2–1. Block Diagram of TMS320C54x Internal Hardware



2.1 Bus Structure

The ’54x architecture is built around eight major 16-bit buses (four program/data buses and four address buses):

- The program bus (PB) carries the instruction code and immediate operands from program memory.
- Three data buses (CB, DB, and EB) interconnect to various elements, such as the CPU, data address generation logic, program address generation logic, on-chip peripherals, and data memory.
  - The CB and DB carry the operands that are read from data memory.
  - The EB carries the data to be written to memory.
- Four address buses (PAB, CAB, DAB, and EAB) carry the addresses needed for instruction execution.

The ’54x can generate up to two data-memory addresses per cycle using the two auxiliary register arithmetic units (ARAU0 and ARAU1).

The PB can carry data operands stored in program space (for instance, a coefficient table) to the multiplier and adder for multiply/accumulate operations or to a destination in data space for data move instructions (MVPD and READA). This capability, in conjunction with the feature of dual-operand read, supports the execution of single-cycle, 3-operand instructions such as the FIRS instruction.

The ’54x also has an on-chip bidirectional bus for accessing on-chip peripherals; this bus is connected to DB and EB through the bus exchanger in the CPU interface. Accesses that use this bus can require two or more cycles for reads and writes, depending on the peripheral’s structure.

Table 2–1 summarizes the buses used by various types of accesses.

Table 2–1. Bus Usage for Read and Write Accesses

	Address Bus				Data Bus			
Access Type	**PAB**	**CAB**	**DAB**	**EAB**	**PB**	**CB**	**DB**	**EB**
Program read	√				√			
Program write	√							√
Data single read			√				√	
Data dual read		√	√			√	√	
Data long (32-bit) read		√(hw)	√(lw)			√(hw)	√(lw)	
Data single write				√				√
Data read/data write			√	√			√	√
Dual read/coefficient read	√	√	√		√	√	√	
Peripheral read			√				√	
Peripheral write				√				√

Legend: hw = high 16-bit word
lw = low 16-bit word

2.2 Internal Memory Organization

The ’54x memory is organized into three individually selectable spaces: program, data, and I/O space. All ’54x devices contain both random-access memory (RAM) and read-only memory (ROM). Among the devices, two types of RAM are represented: dual-access RAM (DARAM) and single-access RAM (SARAM). Table 2–2 shows how much ROM, DARAM, and SARAM are available on the different ’54x devices. The ’54x also has 26 CPU registers plus peripheral registers that are mapped in data-memory space. The ’54x memory types and features are introduced in the subsections following this paragraph. For details about configuring and using the various memory blocks, see Chapter 3, *Memory.*

Table 2–2. Program and Data Memory on the TMS320C54x Devices

Memory Type	’541	’542	’543	’545	’546	’548	’549
ROM:	28K	2K	2K	48K	48K	2K	16K
Program	20K	2K	2K	32K	32K	2K	16K
Program/data	8K	0	0	16K	16K	0	16K
DARAM†	5K	10K	10K	6K	6K	8K	8K
SARAM†	0	0	0	0	0	24K	24K

† You can configure the dual-access RAM (DARAM) and single-access RAM (SARAM) as data memory or program/data memory.

2.2.1 On-Chip ROM

The on-chip ROM is part of the program memory space and, in some cases, part of the data memory space. The amount of on-chip ROM available on each device varies, as shown in Table 2–2.

On devices with a small amount of ROM (2K words), the ROM contains a boot-loader that is useful for booting to faster on-chip or external RAM. For boot-loading details on all ’54x devices except the ’548 and ’549, see *TMS320C54x DSP Reference Set, Volume 4: Applications Guide.* For boot-loading details on the ’548 and ’549, see *TMS320C548/549 Bootloader Technical Reference.*

On devices with larger amounts of ROM, a portion of the ROM may be mapped into both data and program space. The larger ROMs are also custom ROMs: you provide the code or data to be programmed into the ROM in object file format, and Texas Instruments generates the appropriate process mask to program the ROM. For details on submitting ROM codes to Texas Instruments, see Appendix D, *Submitting ROM Codes to TI.*

2.2.2 On-Chip Dual-Access RAM (DARAM)

The DARAM is composed of several blocks. Because each DARAM block can be accessed twice per machine cycle, the central processing unit (CPU) and peripherals such as the buffered serial port (BSP) and host port interface (HPI) can read from and write to a DARAM memory address in the same cycle. The DARAM is always mapped in data space and is primarily intended to store data values. It can also be mapped into program space and used to store program code.

2.2.3 On-Chip Single-Access RAM (SARAM)

The SARAM is composed of several blocks. Each block is accessible once per machine cycle for either a read or a write. The SARAM is always mapped in data space and is primarily intended to store data values. It can also be mapped into program space and used to store program code.

2.2.4 On-Chip Memory Security

The ’54x maskable memory security option protects the contents of on-chip memories. When you designate this option, no externally originating instruction can access the on-chip memory spaces.

2.2.5 Memory-Mapped Registers

The data memory space contains memory-mapped registers for the CPU and the on-chip peripherals. These registers are located on data page 0, simplifying access to them. The memory-mapped access provides a convenient way to save and restore the registers for context switches and to transfer information between the accumulators and the other registers.

2.3 Central Processing Unit (CPU)

The ’54x CPU is common to all the ’54x devices. The ’54x CPU contains:

- 40-bit arithmetic logic unit (ALU)
- Two 40-bit accumulators
- Barrel shifter
- 17 × 17-bit multiplier
- 40-bit adder
- Compare, select, and store unit (CSSU)
- Data address generation unit
- Program address generation unit

2.3.1 Arithmetic Logic Unit (ALU)

The ’54x performs 2s-complement arithmetic with a 40-bit arithmetic logic unit (ALU) and two 40-bit accumulators (accumulators A and B). The ALU can also perform Boolean operations. The ALU uses these inputs:

- 16-bit immediate value
- 16-bit word from data memory
- 16-bit value in the temporary register, T
- Two 16-bit words from data memory
- 32-bit word from data memory
- 40-bit word from either accumulator

The ALU can also function as two 16-bit ALUs and perform two 16-bit operations simultaneously. See Section 4.2, *Arithmetic Logic Unit (ALU)*, on page 4-11, for more details about ALU operation.

2.3.2 Accumulators

Accumulators A and B (see Figure 2–1 on page 2-2) store the output from the ALU or the multiplier/adder block. They can also provide a second input to the ALU; accumulator A can be an input to the multiplier/adder. Each accumulator is divided into three parts:

- Guard bits (bits 39–32)
- High-order word (bits 31–16)
- Low-order word (bits 15–0)

Instructions are provided for storing the guard bits, for storing the high- and the low-order accumulator words in data memory, and for transferring 32-bit accumulator words in or out of data memory. Also, either of the accumulators can be used as temporary storage for the other. See Section 4.3, *Accumulators A and B*, on page 4-15, for more details about the features of these accumulators.

2.3.3 Barrel Shifter

The ’54x barrel shifter has a 40-bit input connected to the accumulators or to data memory (using CB or DB), and a 40-bit output connected to the ALU or to data memory (using EB). The barrel shifter can produce a left shift of 0 to 31 bits and a right shift of 0 to 16 bits on the input data. The shift requirements are defined in the shift count field of the instruction, the shift count field (ASM) of status register ST1, or in the temporary register T (when it is designated as a shift count register).

The barrel shifter and the exponent encoder normalize the values in an accumulator in a single cycle. The LSBs of the output are filled with 0s, and the MSBs can be either zero filled or sign extended, depending on the state of the sign-extension mode bit (SXM) in ST1. Additional shift capabilities enable the processor to perform numerical scaling, bit extraction, extended arithmetic, and overflow prevention operations. See Section 4.4, *Barrel Shifter*, on page 4-19, for more details about the function and use of the shifter. See Section 4.7, *Exponent Encoder*, on page 4-29, for more information about the encoder’s accumulator-normalizing function.

2.3.4 Multiplier/Adder Unit

The multiplier/adder unit performs 17 $\times$ 17-bit 2s-complement multiplication with a 40-bit addition in a single instruction cycle. The multiplier/adder block consists of several elements: a multiplier, an adder, signed/unsigned input control logic, fractional control logic, a zero detector, a rounder (2s complement), overflow/saturation logic, and a 16-bit temporary storage register (T). The multiplier has two inputs: one input is selected from T, a data-memory operand, or accumulator A; the other is selected from program memory, data memory, accumulator A, or an immediate value.

The fast, on-chip multiplier allows the ’54x to perform operations efficiently such as convolution, correlation, and filtering. In addition, the multiplier and ALU together execute multiply/accumulate (MAC) computations and ALU operations in parallel in a single instruction cycle. This function is used in determining the Euclidian distance and in implementing symmetrical and LMS filters, which are required for complex DSP algorithms. See Section 4.5, *Multiplier/Adder Unit*, on page 4-21, for more details about the multiplier/adder unit.

2.3.5 Compare, Select, and Store Unit (CSSU)

The compare, select, and store unit (CSSU) performs maximum comparisons between the accumulator's high and low word, allows both the test/control flag bit (TC) in status register ST0 and the transition register (TRN) to keep their transition histories, and selects the larger word in the accumulator to store into data memory. The CSSU also accelerates Viterbi-type butterfly computations with optimized on-chip hardware. See Section 4.6, *Compare, Select, and Store Unit (CSSU)*, on page 4-26, for more details about this unit.

2.4 Data Addressing

The '54x offers seven basic data addressing modes:

- ❏ Immediate addressing uses the instruction to encode a fixed value.
- ❏ Absolute addressing uses the instruction to encode a fixed address.
- ❏ Accumulator addressing uses accumulator A to access a location in program memory as data.
- ❏ Direct addressing uses seven bits of the instruction to encode the lower seven bits of an address. The seven bits are used with the data page pointer (DP) or the stack pointer (SP) to determine the actual memory address.
- ❏ Indirect addressing uses the auxiliary registers to access memory.
- ❏ Memory-mapped register addressing uses the memory-mapped registers without modifying either the current DP value or the current SP value.
- ❏ Stack addressing manages adding and removing items from the system stack.

During the execution of instructions using direct, indirect, or memory-mapped register addressing, the data-address generation logic (DAGEN) computes the addresses of data-memory operands. For a detailed discussion of the data addressing modes, see Chapter 5, *Data Addressing.*

2.5 Program Memory Addressing

Program memory is usually addressed on a ’54x device with the program counter (PC). With some instructions, however, absolute addressing may be used to access data items that have been stored in program memory. (Absolute addressing is described in Chapter 5, *Data Addressing*.)

The PC, which is used to fetch individual instructions, is loaded by the program-address generation logic (PAGEN). Typically, the PAGEN increments the PC as sequential instructions are fetched. However, the PAGEN may load the PC with a non-sequential value as a result of some instructions or other operations. Operations that cause a discontinuity include branches, calls, returns, conditional operations, single-instruction repeats, multiple-instruction repeats, reset, and interrupts. For calls and interrupts, the current PC is saved onto the stack, which is referenced by the stack pointer (SP). When the called function or interrupt service routine is finished, the PC value that was saved is restored from the stack via a return instruction.

For a detailed discussion of the hardware and software factors in program address generation, see Chapter 6, *Program Memory Addressing.*

2.6 Pipeline Operation

An instruction pipeline consists of a sequence of operations that occur during the execution of an instruction. The ’54x pipeline has six levels: *prefetch*, *fetch*, *decode*, *access*, *read*, and *execute*. At each of the levels, an independent operation occurs. Because these operations are independent, from one to six instructions can be active in any given cycle, each instruction at a different stage of completion. Typically, the pipeline is full with a sequential set of instructions, each at one of the six stages. When a PC discontinuity occurs, such as during a branch, call, or return, one or more stages of the pipeline may be temporarily unused. For more details about the pipeline operation, see Chapter 7, *Pipeline*.

2.7 On-Chip Peripherals

All the ’54x devices have the same CPU, but different on-chip peripherals are connected to their CPUs. The ’54x devices have these on-chip peripheral options:

- General-purpose I/O pins ($\overline{\text{BIO}}$ and XF)
- Software-programmable wait-state generator
- Programmable bank-switching logic
- Host port interface (HPI)
- Hardware timer
- Clock generator
- Serial ports
  - Synchronous serial ports
  - Buffered serial ports
  - Time-division multiplexed (TDM) serial ports

For more information about these peripherals, see Chapter 8, *On-Chip Peripherals,* Chapter 9, *Serial Ports*, and Chapter 10, *External Bus Operation*.

2.7.1 General-Purpose I/O Pins

Each ’54x device has two general-purpose I/O pins: $\overline{\text{BIO}}$ and XF. $\overline{\text{BIO}}$ is an input pin that can be used to monitor the status of external devices. XF is a software-controlled output pin that allows you to signal external devices. See Section 8.2, *General-Purpose I/O*, on page 8-12, for more details about $\overline{\text{BIO}}$ and XF.

2.7.2 Software-Programmable Wait-State Generator

The software-programmable wait-state generator extends external bus cycles up to seven machine cycles (14 machine cycles in the ’549) to interface with slower off-chip memory and I/O devices. The software wait-state generator is incorporated without any external hardware. For off-chip memory accesses, from zero to seven wait states can be specified within the software wait-state register (SWWSR) for each 32K-word block of program and data memory, and for the 64K-word block of I/O space. See section 10.3.1, *Wait-State Generator*, on page 10-5, for more details.

2.7.3 Programmable Bank-Switching Logic

The programmable bank-switching logic can automatically insert one cycle when an access crosses memory bank boundaries inside program memory or data memory. One cycle can also be inserted when an access crosses from program memory to data memory. This extra cycle prevents bus contention by allowing memory devices to release the bus before other devices start driving the bus. The size of memory bank for bank switching is defined by the bank switching control register (BSCR). See subsection 10.3.2, *Bank-Switching Logic*, on page 10-8, for more details.

2.7.4 Host Port Interface

The host port interface (HPI) is an 8-bit parallel port that provides an interface to a host processor. Information is exchanged between the ’54x and the host processor through ’54x on-chip memory that is accessible to both the host processor and the ’54x. Table 2–3 identifies the HPI-equipped ’54x devices. See Section 8.5, *Host Port Interface*, on page 8-28, for more details about HPI operation.

Table 2–3. Host Port Interfaces on the TMS320C54x Devices

On-Chip Peripheral	’541	’542	’543	’545	’546	’548	’549
Host port interface	0	1	0	1	0	1	1

2.7.5 Hardware Timer

The ’54x features a 16-bit timing circuit with a 4-bit prescaler. The timer counter is decremented by 1 at every CLKOUT cycle. Each time the counter decrements to 0, a timer interrupt is generated. The timer can be stopped, restarted, reset, or disabled by specific status bits. See Section 8.3, *Timer*, on page 8-14, for more details.

2.7.6 Clock Generator

The clock generator consists of an internal oscillator and a phase-locked loop (PLL) circuit. The clock generator can be driven internally by a crystal resonator with the internal oscillator or externally by a clock source. The PLL circuit can generate an internal CPU clock by multiplying the clock source by a specific factor; thus, you should use a clock source with a lower frequency than that of the CPU. For more details about the generator, see Section 8.4, *Clock Generator*, on page 8-18.

2.8 Serial Ports

The serial ports on the ’54x vary by device, but three types of serial ports are represented: synchronous, buffered, and time-division multiplexed (TDM). See Table 2–4 for the number of each type on the various ’54x devices. The subsections following this paragraph provide an introduction to the three types of serial ports. For more details about these ports, turn to Chapter 9, *Serial Ports*.

Table 2–4. Serial Port Interfaces on the TMS320C54x Devices

Serial Ports	’541	’542	’543	’545	’546	’548	’549
Synchronous	2	0	0	1	1	0	0
Buffered	0	1	1	1	1	2	2
TDM	0	1	1	0	0	1	1

2.8.1 Synchronous Serial I/O Ports

The synchronous serial ports are high-speed, full-duplexed serial ports that provide direct communication with serial devices such as codecs, analog-to-digital (A/D) converters, and other serial systems. When more than one synchronous serial port resides on a ’54x, these ports are identical but independent. Each synchronous serial port can operate at up to one-fourth the machine cycle rate (CLKOUT). The synchronous serial port transmitter and receiver are double buffered and individually controlled by maskable external interrupt signals. Data is framed either as bytes or as words.

2.8.2 Buffered Serial Ports

A buffered serial port (BSP) is a synchronous serial port that is enhanced with an autobuffering unit and is clocked at the full CLKOUT rate. It is full-duplexed and double-buffered to offer flexible data stream length. The autobuffering unit supports high-speed transfers and reduces the overhead of servicing interrupts.

2.8.3 TDM Serial Ports

A time-division multiplexed (TDM) serial port is a synchronous serial port that is enhanced to allow time-division multiplexing of the data. It can be configured for either synchronous operations or for TDM operations and is commonly used in multiprocessor applications.

2.9 External Bus Interface

The ’54x can address up to 64K words of data memory, 64K words of program memory (8M words in the ’548 and ’549), and up to 64K words of 16-bit parallel I/O ports. Accesses to either external memory or I/O ports take place through the external interface. Individual space-select signals, $\overline{DS}$, $\overline{PS}$, and $\overline{IS}$, allow the selection of physically separate spaces.

The interface’s external ready input signal and software-generated wait states allow the processor to interface with memory and I/O devices of many different speeds. The interface’s hold modes allow an external device to take control of the ’54x buses; in this way, an external device can access the resources in the program, data, and I/O spaces.

External memory can be accessed by most ’54x instructions. However, accessing I/O ports requires the use of special instructions: PORTR and PORTW.

See Chapter 10, *External Bus Operation*, for more details about interfacing the ’54x to external devices.

2.10 IEEE Standard 1149.1 Scanning Logic

The IEEE Standard 1149.1 scanning-logic circuitry is used for emulation and testing purposes only. This logic provides the boundary scan to and from the interfacing devices. Also, it can be used to test pin-to-pin continuity as well as to perform operational tests on devices peripheral to the ’54x. The IEEE Standard 1149.1 scanning logic is interfaced to internal scanning-logic circuitry that has access to all of the on-chip resources. Thus, the ’54x can perform on-board emulation using the IEEE Standard 1149.1 serial scan pins and the emulation-dedicated pins. See Appendix B, *Design Considerations for Using XDS510 Emulator,* for more information.

Chapter 3

Memory

This chapter describes the ’54x memory configuration and operation. In general, the ’54x devices have a total memory space of 192K 16-bit words. This space is divided into three specific memory segments: 64K words of program, 64K words of data, and 64K words of I/O. In some devices, such as the ’548 and ’549, the memory structure has been modified through overlay and paging schemes. The parallel nature of the ’54x architecture and the dual-access capability of the on-chip RAM allow the ’54x to perform four concurrent memory operations in any given machine cycle: an instruction fetch, two-operand reads, and an operand write.

There are several advantages of operating from on-chip memory:

- Higher performance because no wait states are required
- Lower cost than external memory
- Lower power than external memory

The main advantage of operating from off-chip memory is the ability to access a larger memory space.

Topic **Page**

3.1 Memory Space

The ’54x’s memory is organized into three individually selectable spaces: program, data, and I/O. Within any of these spaces, RAM, ROM, EPROM, EEPROM, or memory-mapped peripherals can reside either on- or off-chip. Together, these three spaces provide a total address range of 192K words (except in the ’548 and ’549).

The program memory space contains the instructions to execute, as well as tables used in execution. The data-memory space stores data used by instructions. The I/O memory space interfaces to external memory-mapped peripherals and can also serve as extra data storage space.

Depending on the chip version, several on-chip memory types are available on the ’54x: dual-access RAM (DARAM), single-access RAM (SARAM), and ROM. The RAMs are always mapped into data space, but may also be mapped into program space. The ROM may be activated and mapped into program space; it can also be mapped, in part, into data space.

Three features of the ’54x allow flexibility in enabling or disabling on-chip memories in the program and data spaces:

- MP/$\overline{\text{MC}}$ bit. If this bit is set to 0, the on-chip ROM is mapped into program space. If this bit is set to 1, the on-chip ROM is not mapped into program space.
- OVLY bit. If this bit is set to 1, the on-chip RAMs are mapped into program and data space. If this bit is set to 0, the on-chip RAMs are mapped only into data space.
- DROM bit. If this bit is set to 1, a part of the on-chip ROM is mapped into data space. If this bit is set to 0, the on-chip ROM is not mapped into data space. The use of DROM is independent of the state of MP/$\overline{\text{MC}}$.

The MP/$\overline{\text{MC}}$, OVLY, and DROM bits are located in the processor mode status register (PMST). For more details, see Section 4.1, *CPU Status and Control Registers*, on page 4-2.

Figure 3–1 through Figure 3–4 show the ’54x device’s data and program memory maps and how the maps are affected by the MP/$\overline{\text{MC}}$, OVLY, and DROM bits.

Figure 3–1. Memory Maps for the TMS320C541

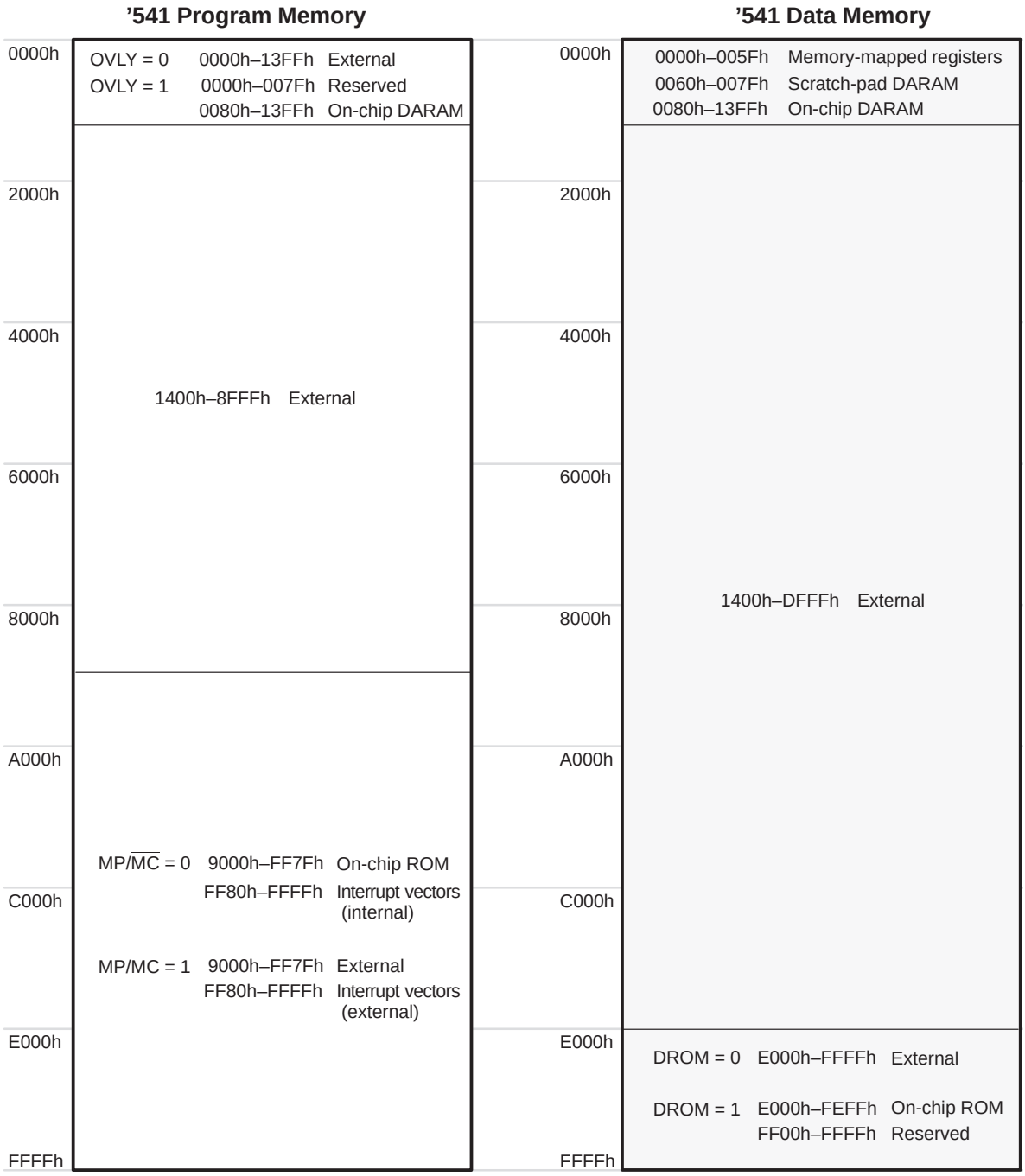


Figure 3–2. Memory Maps for the TMS320C542 and TMS320C543

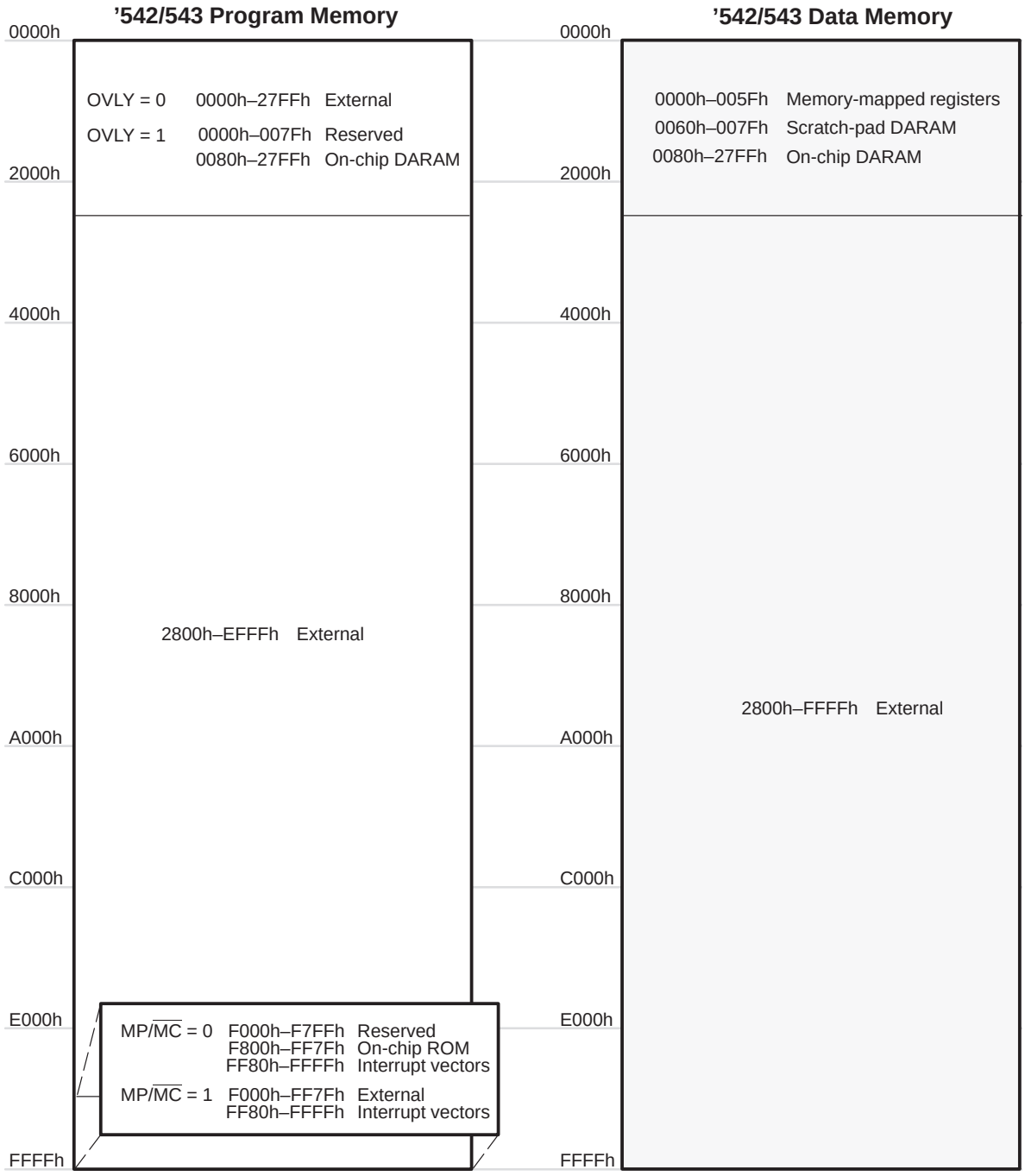


Figure 3–3. Memory Maps for the TMS320C545 and TMS320C546

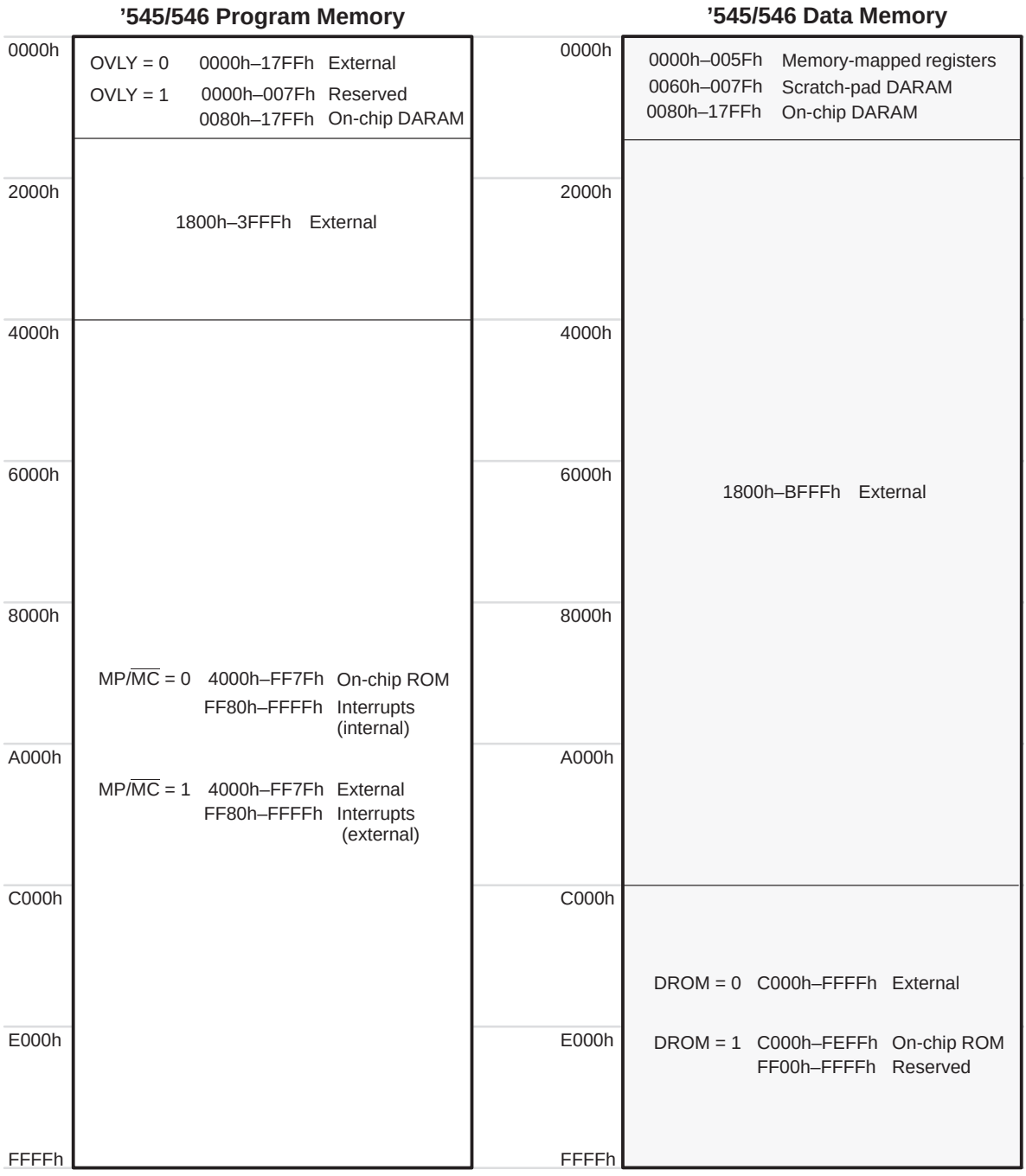


Figure 3–4. Memory Maps for the TMS320C548

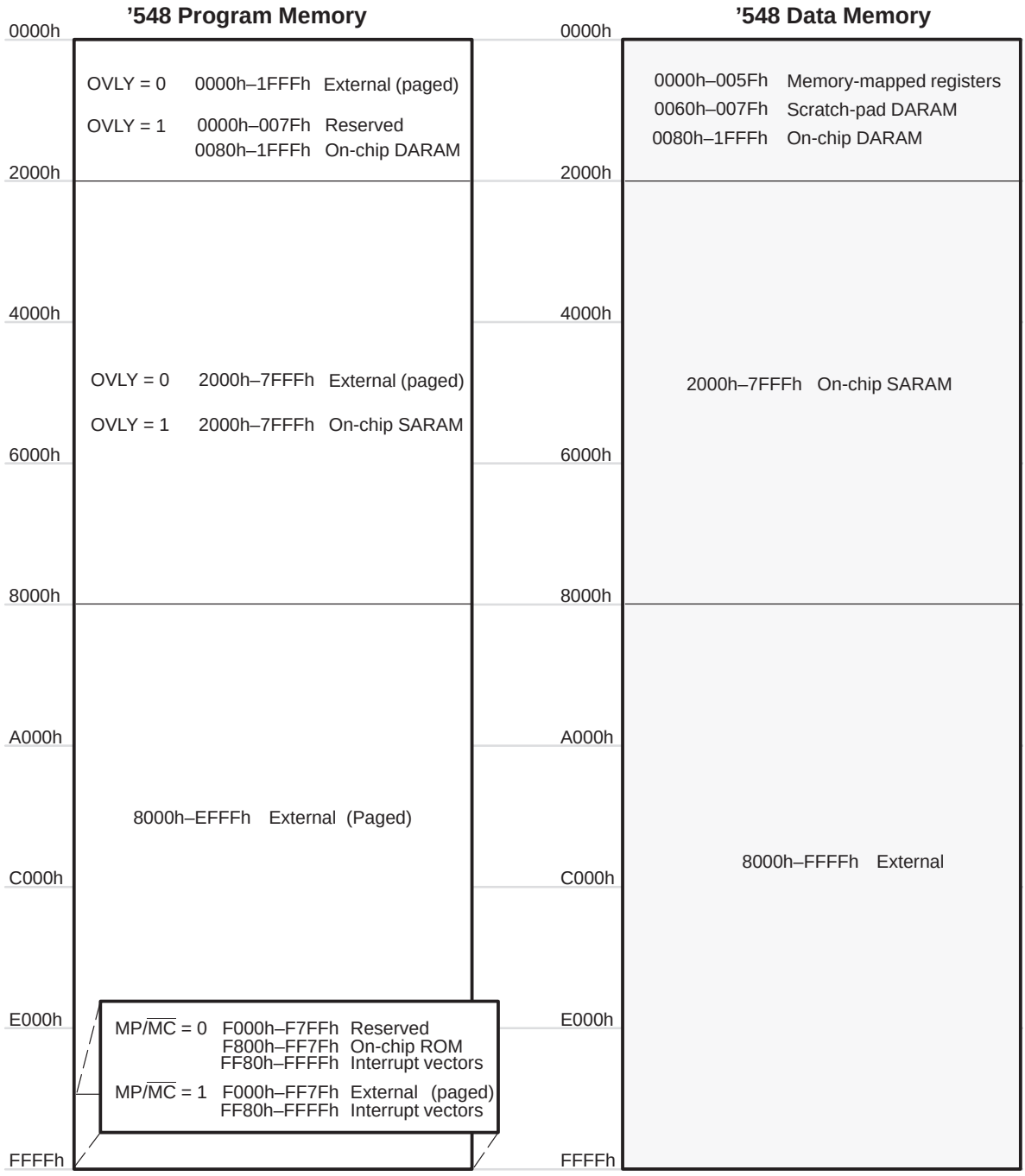
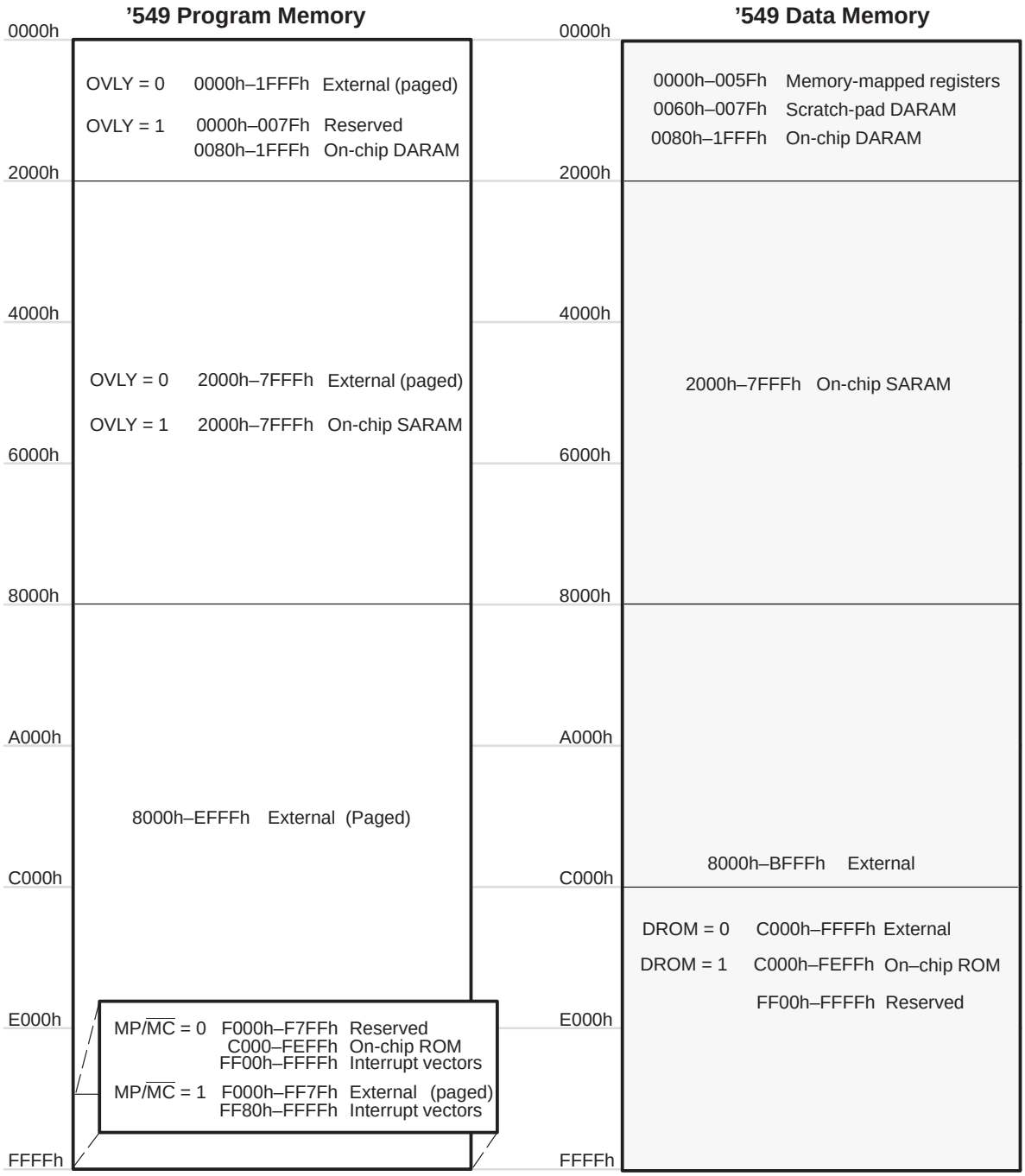


Figure 3–5. Memory Maps for the TMS320C549



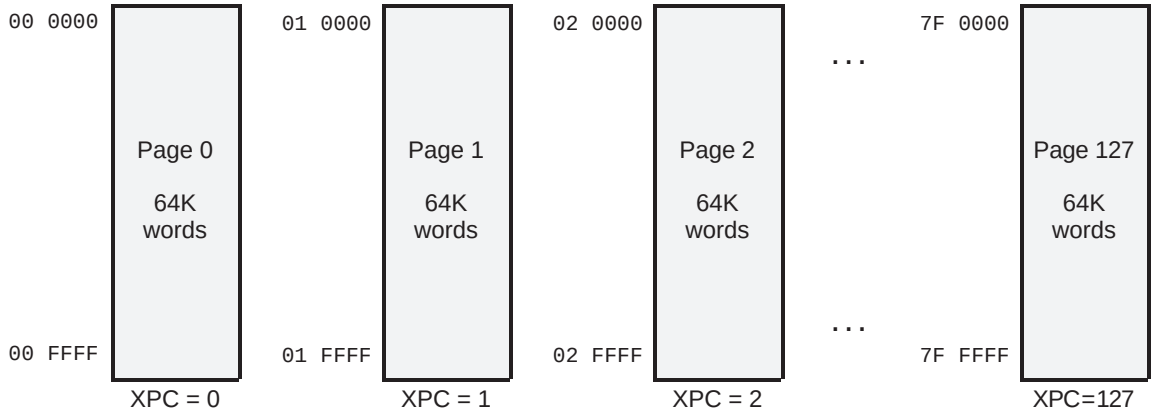
3.1.1 Extended Program Memory (Available on TMS320C548/549)

The ’548 and ’549 use a paged extended memory scheme in program space to allow access of up to 8192K words of program memory. To implement this scheme, the ’548 and ’549 include several additional features:

- 23 address lines, instead of 16
- An extra memory-mapped register, the program counter extension register (XPC)
- Six extra instructions for addressing extended program space

Program memory in the ’548 and ’549 is organized into 128 pages that are each 64K words in length, as shown in Figure 3–6.

Figure 3–6. Extended Program Memory With On-Chip RAM Not Mapped in Program Space (OVLY = 0)

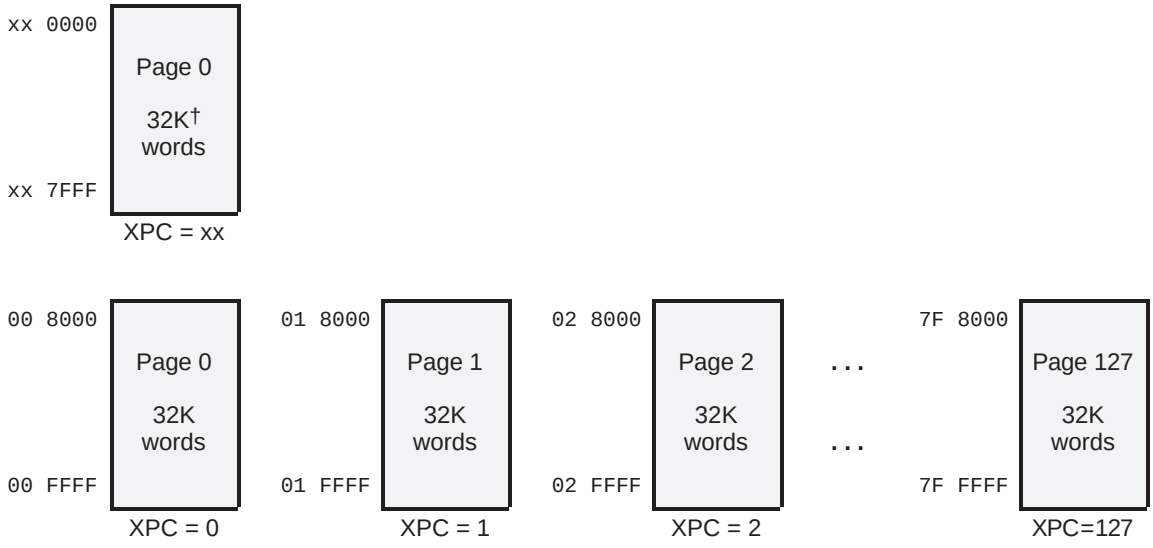


When the on-chip RAM is enabled in program space, each page of program memory is made up of two parts: a common block of 32K words maximum and a unique block of 32K words. The common block is shared by all pages and each unique block is accessible only through its assigned page. Figure 3–7 shows the common and unique blocks.

If the on-chip ROM is enabled (MP/$\overline{\text{MC}}$ = 0), it is enabled only on page 0. It is not mapped to any other page in program memory.

The value of the XPC register defines the page selection. This register is memory-mapped into data space to address 001Eh. At a hardware reset, the XPC is initialized to 0.

Figure 3–7. Extended Program Memory With On-Chip RAM Mapped in Program Space and Data Space (OVLY = 1)



Note: When the on-chip RAM is enabled in program space, all accesses to the region xx 0000 – xx 7FFF, regardless of page number, are mapped to the on-chip RAM at 00 0000 – 00 7FFF.

† See Figure 3–4 on page 3-6 for more information about this on-chip memory region.

To facilitate page switching through software, the ’548 and ’549 have six special instructions that affect the XPC:

- FB[D] – Far branch
- FBACC[D] – Far branch to the location specified by the value in accumulator A or accumulator B
- FCALA[D] – Far call to the location specified by the value in accumulator A or accumulator B
- FCALL[D] – Far call
- FRET[D] – Far return
- FRETE[D] – Far return with interrupts enabled

In addition to these new instructions, two ’54x instructions are extended in the ’548 and ’549 to use 23 bits:

- READA – Read program memory addressed by accumulator A and store in data memory
- WRITA – Write data to program memory addressed by accumulator A

All other instructions do not modify the XPC and access only memory within the current page.

3.2 Program Memory

The external program memory on the ’54x devices (except on the ’548 and ’549) addresses up to 64K 16-bit words. ’54x devices have on-chip ROM, dual-access RAM (DARAM), and single-access RAM (SARAM) that can be mapped by software into the program space. When the cells are mapped into program space, the device automatically accesses them when addresses fall within their bounds. When the program address generation unit (PAGEN) generates an address outside of the bounds of on-chip memory, the device automatically generates an external access. (For more information about program address generation, see Chapter 6, *Program Memory Addressing*.) Table 3–1 shows the on-chip program memory available on the various ’54x devices.

Table 3–1. On-Chip Program Memory Available on the TMS320C54x Devices

Device	ROM (MP/$\overline{MC}$ = 0)	DARAM (OVLY = 1)	SARAM (OVLY = 1)
‘541	28K	5K	–
‘542	2K	10K	–
‘543	2K	10K	–
‘545	48K	6K	–
‘546	48K	6K	–
‘548	2K	8K	24K
’549	16K	8K	24K

3.2.1 Program Memory Configurability

The MP/$\overline{MC}$ and OVLY bits determine which on-chip memories are enabled in program space.

At reset, the logic level present on the MP/$\overline{MC}$ pin is transferred to the MP/$\overline{MC}$ bit in the PMST register (see Section 4.1, *CPU Status and Control Registers*, on page 4-2). The MP/$\overline{MC}$ bit determines whether to enable the on-chip ROM. If MP/$\overline{MC}$ = 1, the device is configured as a microprocessor, and the on-chip ROM is not enabled. If MP/$\overline{MC}$ pin = 0, the device is configured as a microcomputer, and the on-chip ROM is enabled. The MP/$\overline{MC}$ pin is sampled only at reset; however, you can disable or enable the on-chip ROM through software by setting or clearing the MP/$\overline{MC}$ bit in the PMST register.

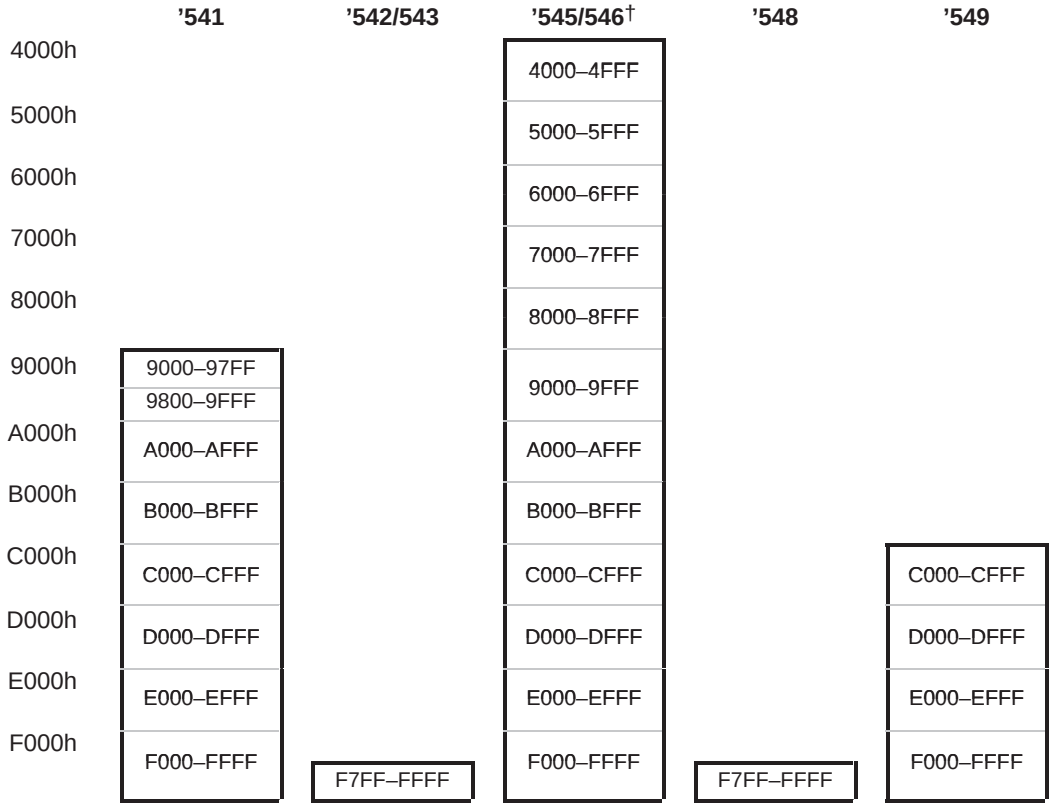
At reset, the RAM—DARAM and SARAM (if present)—is not addressable in program space. You can make the RAM addressable in program space by setting the OVLY bit in the PMST register. If OVLY = 0, then the RAM is mapped only to data space. If OVLY = 1, then the RAM is mapped into both program space and data space.

Figure 3–1 through Figure 3–4 (pages 3-3 through 3-6) show the program memory configurations on the individual ’54x devices.

3.2.2 On-Chip ROM Organization

The on-chip ROM is subdivided and organized in blocks to enhance performance. For example, the block organization enables you to fetch an instruction from one block of ROM without sacrificing data accesses that come from a different block of ROM. Figure 3–8 shows the way the ROM is organized in blocks for each ’54x device. The gray lines in the figure indicate block boundaries.

Figure 3–8. On-Chip ROM Block Organization



† ROM is organized in 8K blocks on LP versions of these devices.

3.2.3 Program Memory Address Map and On-Chip ROM Contents

At device reset, the reset, interrupt, and trap vectors are mapped to address FF80h in program space. However, these vectors can be remapped to the beginning of any 128-word page in program space after device reset. This feature facilitates moving the vector table out of the boot ROM and then removing the ROM from the memory map. For details on remapping the vectors, see subsection 6.10.9, *Remapping Interrupt-Vector Addresses*, on page 6-35.

Note:

In the on-chip ROM, 128 words are reserved for device-testing purposes. Application code written to be implemented in on-chip ROM must reserve these 128 words at addresses FF00h–FF7Fh in program space.

3.2.4 On-Chip ROM Code Contents and Mapping

'54x devices have either large (16K, 24K, 28K, or 48K words) on-chip ROM or 2K words of on-chip ROM. A large on-chip ROM can be programmed with your code, while the content of a 2K-word on-chip ROM is defined by Texas Instruments. On '54x devices with on-chip bootloader ROM, the 2K words (at F800h to FFFFh) may contain one or more of the following, depending on the specific device:

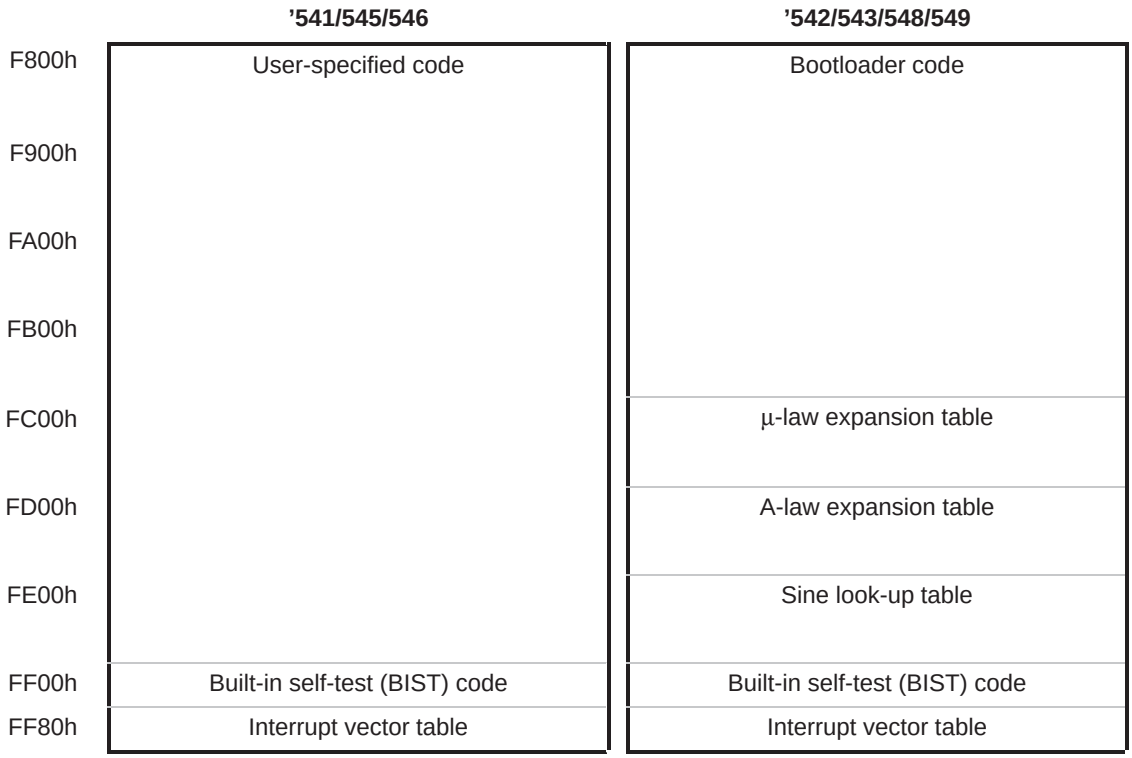
- A bootloader program that boots from the serial ports, external memory, an I/O port, or the host port interface (if present)
- A 256-word μ-law expansion table
- A 256-word A-law expansion table
- A 256-word sine look-up table
- An interrupt vector table

Figure 3–9 shows which of these items are on a particular '54x device and shows the addresses of each of the items. The address range for the code, F800h–FFFFh, is mapped to the on-chip ROM if the MP/$\overline{\text{MC}}$ bit is 0.

Note:

You can submit code to Texas Instruments in object file format to program into the '54x on-chip ROM. See Appendix D, *Submitting ROM Codes to TI*, for details on how to submit ROM code to Texas Instruments.

Figure 3–9. On-Chip ROM Program Memory Map (High Addresses)



3.3 Data Memory

The data memory on the ’54x contains up to 64K 16-bit words. A number of the ’54x devices have on-chip ROM that can be mapped by software into the data space (DROM) in addition to any dual- and single-access RAM (DARAM and SARAM). Table 3–2 shows the on-chip data memory available on various ’54x devices.

Table 3–2. On-Chip Data Memory Available on the TMS320C54x Devices

Device	Program/Data ROM (DROM = 1)	DARAM	SARAM
’541	8K	5K	–
’542	–	10K	–
’543	–	10K	–
’545	16K	6K	–
’546	16K	6K	–
’548	–	8K	24K
’549	16K	8K	24K

Accesses to the RAM and the data ROM (when it is enabled) are made when addresses fall within the bounds of the corresponding on-chip memories. When the data-address generation logic (DAGEN) generates an address outside of the bounds of on-chip memory, the device automatically generates an external access. (For more information about the generation of data addresses, see Chapter 5, *Data Addressing*.)

3.3.1 Data Memory Configurability

Data memory can reside both on- and off-chip. The on-chip DARAM is mapped into data memory space. For some ’54x devices, you can map a portion of the on-chip ROM (the amount shown in Table 3–2) into data space by setting the DROM bit located in the PMST register (see Section 4.1, *CPU Status and Control Registers*, on page 4-2). This portion of on-chip ROM is enabled both in the data space (DROM bit) and in the program space (MP/$\overline{\text{MC}}$ bit), allowing an instruction to use the ROM area as a data ROM residing in data space. At reset, the processor clears the DROM bit to 0.

The data ROM is accessed in a single cycle by an instruction using single data-memory operand addressing, including an instruction with a 32-bit long word operand. In the dual-memory operand addressing, the access requires two cycles if both operands reside in the same block; if the operands reside in different blocks, the access requires a single cycle. For the address boundaries of the ROM blocks, see subsection 3.2.2, *On-Chip ROM Organization*, on page 3-11.

Figure 3–1 through Figure 3–4 (pages 3-3 through 3-6) show the data memory configurations on the individual ’54x devices.

3.3.2 On-Chip RAM Organization

On-chip RAM is subdivided and organized in blocks to enhance performance. For example, the block organization enables you to fetch two operands from one block of DARAM and write to another block of DARAM in the same cycle. Figure 3–10 on page 3-16 shows the RAM block organization for each ’54x device. The gray lines in the figure indicate block boundaries.

Figure 3–11 on page 3-17 shows the organization of the first 1K of DARAM on all ’54x devices. This portion of data memory includes the memory-mapped CPU and peripheral registers, 32 words of scratch-pad DARAM, and 896 words of DARAM. The gray lines in the figure indicate memory-page boundaries used with direct memory addressing (see Section 5.4, *Direct Addressing*, on page 5-7), and DP refers to the data page pointer in ST0 (see Section 4.1, *CPU Status and Control Registers*, on page 4-2).

Figure 3–10. On-Chip RAM Block Organization

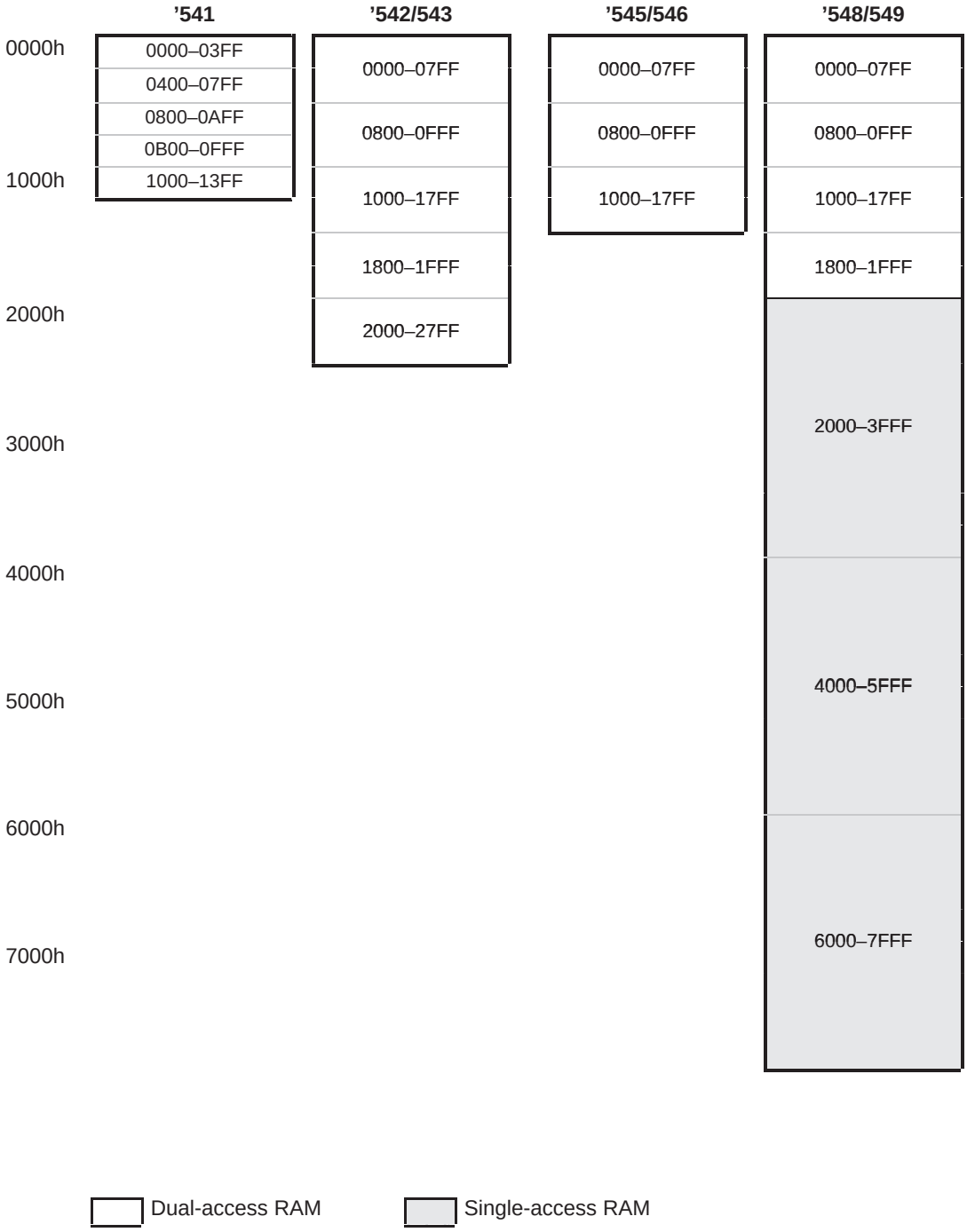
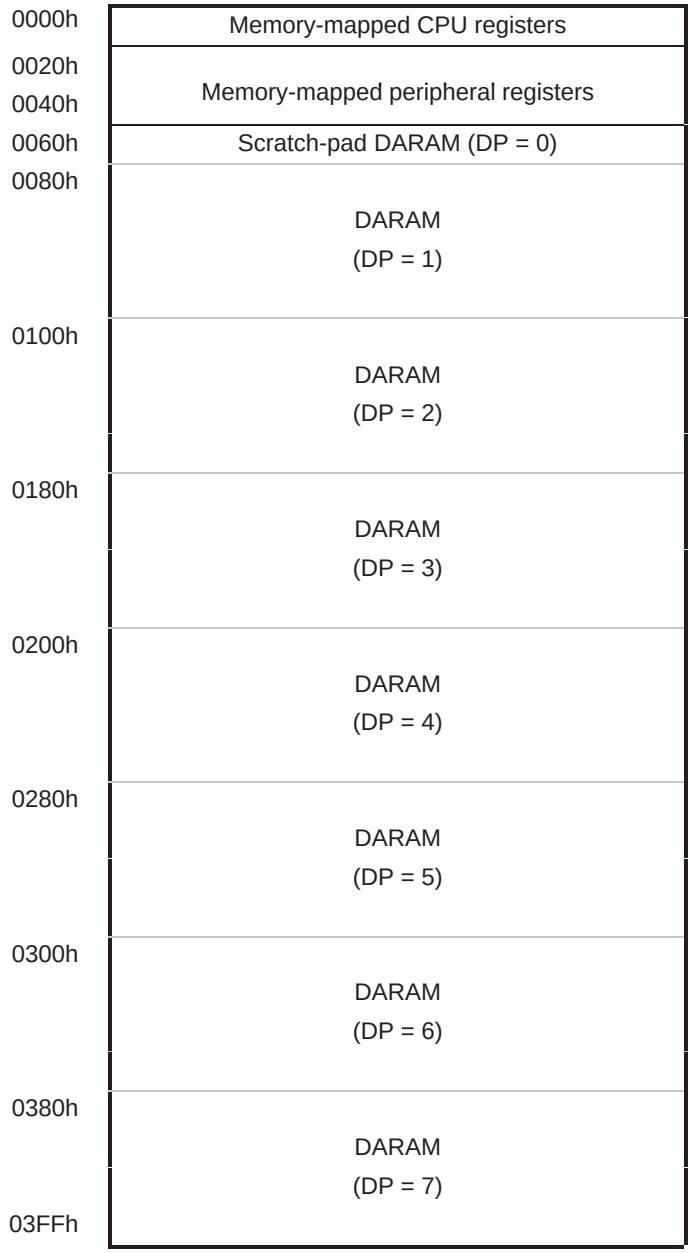


Figure 3–11. Low Addresses of On-Chip Data Memory



3.3.3 Memory-Mapped Registers

The 64K words of data memory space include the device's memory-mapped registers, which reside in data page 0 (data addresses 0000h–007Fh, as shown in Figure 3–11 on page 3-17). Data page 0 consists of the following:

- ❑ The CPU registers (26 total) are accessible with no wait states; see Table 3–3 on page 3-19.
- ❑ The peripheral registers are used as control and data registers in peripheral circuits. These registers reside within addresses 0020h–005F and reside on a dedicated peripheral bus structure called the TIBUS. They require at least two cycles when accessed. The number of cycles required depends on the peripherals built into the '54x devices. For a list of peripherals on a particular '54x device, see Section 8.1, *Peripheral Memory-Mapped Registers*, on page 8-2.
- ❑ The scratch-pad RAM block (60h–7Fh in data memory) includes 32 words of DARAM for variable storage that helps avoid fragmenting the large RAM block.

3.3.4 CPU Memory-Mapped Registers

Table 3–3 lists the CPU memory-mapped registers. This subsection gives a brief summary of one or more of the registers.

3.3.4.1 Interrupt Registers (IMR, IFR)

The interrupt mask register (IMR) individually masks off specific interrupts at required times. The interrupt flag register (IFR) indicates the current status of the interrupts. Interrupts are described in detail in Section 6.10, *Interrupts*, on page 6-26.

3.3.4.2 Status Registers (ST0, ST1)

The status registers ST0 and ST1 contain the status of the various conditions and modes for the '54x devices. ST0 contains the flags (OVA, OVB, C, and TC) produced by arithmetic operations and bit manipulations, in addition to the DP and the ARP fields. ST1 reflects the status of modes and instructions executed by the processor. See Section 4.1, *CPU Status and Control Registers*, on page 4-2 for detailed information.

Table 3–3. CPU Memory-Mapped Registers

Address	Name	Description
0	IMR	Interrupt mask register
1	IFR	Interrupt flag register
2–5	–	Reserved for testing
6	ST0	Status register 0
7	ST1	Status register 1
8	AL	Accumulator A low word (bits 15–0)
9	AH	Accumulator A high word (bits 31–16)
A	AG	Accumulator A guard bits (bits 39–32)
B	BL	Accumulator B low word (bits 15–0)
C	BH	Accumulator B high word (bits 31–16)
D	BG	Accumulator B guard bits (bits 39–32)
E	T	Temporary register
F	TRN	Transition register
10	AR0	Auxiliary register 0
11	AR1	Auxiliary register 1
12	AR2	Auxiliary register 2
13	AR3	Auxiliary register 3
14	AR4	Auxiliary register 4
15	AR5	Auxiliary register 5
16	AR6	Auxiliary register 6
17	AR7	Auxiliary register 7
18	SP	Stack pointer
19	BK	Circular-buffer size register
1A	BRC	Block-repeat counter
1B	RSA	Block-repeat start address
1C	REA	Block-repeat end address
1D	PMST	Processor mode status register
1E	XPC	Program counter extension register (’548 and ’549)
1E–1F	–	Reserved

3.3.4.3 Accumulators (A, B)

The ’54x devices have two 40-bit accumulators: accumulator A and accumulator B. Each accumulator is memory-mapped and partitioned into accumulator low word (AL, BL), accumulator high word (AH, BH), and accumulator guard bits (AG, BG). See Section 4.3, *Accumulators A and B*, on page 4-15 for more details about these accumulator features.

3.3.4.4 Temporary Register (T)

The temporary (T) register has many uses. For example, it may hold:

- One of the multiplicands for multiply and multiply/accumulate instructions (For more details about the T register and the processes of multiplication, see Section 4.5, *Multiplier/Adder Unit*, on page 4-21.)
- A dynamic (execution-time programmable) shift count for instructions with shift operation such as the ADD, LD, and SUB instructions
- A dynamic bit address for the BITT instruction
- Branch metrics used by the DADST and DSADT instructions for ACS operation of Viterbi decoding

In addition, the EXP instruction stores the exponent value computed into T register, and then the NORM instruction uses the T register value to normalize the number.

3.3.4.5 Transition Register (TRN)

The 16-bit transition (TRN) register holds the transition decision for the path to new metrics to perform the Viterbi algorithm. The CMPS (compare select max and store) instruction updates the contents of TRN register on the basis of the comparison between the accumulator high word and the accumulator low word.

3.3.4.6 Auxiliary Registers (AR0–AR7)

The eight 16-bit auxiliary registers (AR0–AR7) can be accessed by the CPU and modified by the auxiliary register arithmetic units (ARAUs). The primary function of the auxiliary registers is to generate 16-bit addresses for data space. However, these registers can also act as general-purpose registers or counters. For information about the role the auxiliary registers play in data-memory addressing, see Section 5.5, *Indirect Addressing*, on page 5-10.

3.3.4.7 Stack-Pointer Register (SP)

The 16-bit stack-pointer register (SP) contains the address of the top of the system stack. The SP always points to the last element pushed onto the stack. The stack is manipulated by interrupts, traps, calls, returns, and the PSHD, PSHM, POPD, and POPM instructions. Pushes and pops of the stack predecrement and postincrement, respectively, the 16-bit value in the stack pointer.

3.3.4.8 Circular-Buffer Size Register (BK)

The ARAUs use16-bit circular-buffer size register (BK) in circular addressing to specify the data block size. For information on BK and circular addressing, see subsection 5.5.3.4, *Circular Address Modifications*, on page 5-15.

3.3.4.9 Block-Repeat Registers (BRC, RSA, REA)

The 16-bit block-repeat counter (BRC) register specifies the number of times a block of code is to repeat when a block repeat is performed. The 16-bit block-repeat start address (RSA) register contains the starting address of the block of program memory to be repeated. The 16-bit block-repeat end address (REA) register contains the ending address of the block of program memory to be repeated. For more information about repeating multiple instructions and the BRC, RSA, and REA, see Section 6.8, *Repeating a Block of Instructions*, on page 6-23.

3.3.4.10 Processor Mode Status Register (PMST)

The processor mode status register (PMST) controls memory configurations of the ’54x devices. The PMST is described in detail in Section 4.1, *CPU Status and Control Registers*, on page 4-2.

3.3.4.11 Program Counter Extension Register (XPC, Available on ’548/549)

The program counter extension register (XPC) contains the upper 7 bits of the current program memory address. See subsection 3.1.1, *Extended Program Memory* on page 3-8, for more information about extended memory.

3.4 I/O Memory

The ’54x devices offer an I/O memory space in addition to the program- and data-memory spaces. The I/O memory space is a 64K-word address space (0000h–FFFFh) and exists only external to the device. Two instructions, PORTR and PORTW, are used to access this space. Read timings vary from those of the program- and data-memory spaces to facilitate access to individual I/O-mapped devices rather than to memories. For details of external bus operation and control for I/O accesses, see Chapter 10, *External Bus Operation*.

3.5 Program and Data Security

The ’54x has two staged security options: on-chip ROM security and ROM/RAM security. See Table 3–4 for a summary of the security feature.

Table 3–4. Data Security

	Affect On-Chip Memory Accesses	
Security	**ROM**	**RAM**
ROM	Affected for both program and data accesses. Instructions fetched from on-chip ROM may read on-chip ROM for program and data accesses. Instructions fetched from on-chip RAM or external memory will be prohibited from accessing on-chip ROM and will read invalid data (0FFFFh) on the bus for program and data accesses.	Unaffected by this security option.
ROM/RAM	Affected for both program and data accesses. Instructions fetched from on-chip ROM and on-chip RAM may read on-chip ROM for program and data accesses. Instructions fetched from external memory will be prohibited from accessing on-chip ROM and will read invalid data (0FFFFh) on the bus for program and data accesses.	Affected for both program and data accesses. Instructions fetched from on-chip ROM and on-chip RAM may read on-chip RAM for program and data accesses. Instructions fetched from external memory will be prohibited from accessing on-chip RAM and will read invalid data (0FFFFh) on the bus for program and data accesses.

Chapter 4

Central Processing Unit

This chapter describes the ’54x central processing unit (CPU) operations. The CPU can perform high-speed arithmetic operations within one instruction cycle because of its parallel architectural design.

The following CPU functional components are discussed in this chapter:

- 40-bit arithmetic logic unit (ALU)
- Two 40-bit accumulator registers
- Barrel shifter supporting a –16 to 31 shift range
- Multiply/accumulate block
- 16-bit temporary register (T)
- 16-bit transition register (TRN)
- Compare, select, and store unit (CSSU)
- Exponent encoder

The CPU registers are memory-mapped, enabling quick saves and restores.

Topic **Page**

4.1 CPU Status and Control Registers

The ’54x has three status and control registers:

- ❏ Status register 0 (ST0)
- ❏ Status register 1 (ST1)
- ❏ Processor mode status register (PMST)

ST0 and ST1 contain the status of various conditions and modes; PMST contains memory-setup status and control information. Because these registers are memory-mapped, they can be stored into and loaded from data memory; the status of the processor can be saved and restored for subroutines and interrupt service routines (ISRs).

4.1.1 Status Registers (ST0 and ST1)

The individual bits of the ST0 and ST1 registers can be set or cleared with the SSBX and RSBX instructions. For example, the sign-extension mode is set with SSBX 1, SXM, or reset with RSBX 1, SXM. The ARP, DP, and ASM bit fields can be loaded using the LD instruction with a short-immediate operand. The ASM and DP fields can be also loaded with data-memory values by using the LD instruction.

The ST0 bits are shown in Figure 4–1 and described in Table 4–1. The ST1 bits are shown in Figure 4–2 and described in Table 4–2 on page 4-4.

Figure 4–1. Status Register 0 (ST0) Diagram

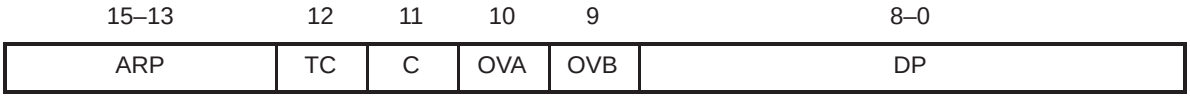


Table 4–1. Status Register 0 (ST0) Bit Summary

Bit	Name	Reset Value	Function
15 – 13	ARP	0	Auxiliary register pointer. This 3-bit field selects the auxiliary register to use in the compatibility mode of indirect single-operand addressing (see section 5.5, *Indirect Addressing*, page 5-10). ARP must always be set to zero when the DSP is in standard mode (CMPT = 0).

Table 4–1. Status Register 0 (ST0) Bit Summary (Continued)

Bit	Name	Reset Value	Function
12	TC	1	Test/control flag. TC stores the results of the arithmetic logic unit (ALU) test bit operations. TC is affected by the BIT, BITF, BITT, CMPM, CMPR, CMPS, and SFTC instructions. The status (set or cleared) of TC determines if the conditional branch, call, execute, and return instructions execute. TC = 1 if the following conditions are true: ❑ A bit tested by BIT or BITT is a 1. ❑ A compare condition tested by CMPM, CMPR, or CMPS exists between a data-memory value and an immediate operand, AR0 and another auxiliary register, or an accumulator high word and an accumulator low word. ❑ Bit 31 and bit 30 of an accumulator tested by SFTC have different values from each other.
11	C	1	Carry is set to 1 if the result of an addition generates a carry; it is cleared to 0 if the result of a subtraction generates a borrow. Otherwise, it is reset after an addition and it is set after a subtraction, except for an ADD or SUB with a 16-bit shift. In these cases, the ADD can only set and the SUB only reset the carry bit, but they cannot affect it otherwise. Carry and borrow are defined at the 32nd bit position and are operated at the ALU level only. The shift and rotate instructions (ROR, ROL, SFTA, and SFTL), and the MIN, MAX, ABS, and NEG instructions also affect this bit.
10	OVA	0	Overflow flag for accumulator A. OVA is set to 1 when an overflow occurs in either the ALU or the multiplier's adder and the destination for the result is accumulator A. Once an overflow occurs, OVA remains set until either a reset, a BC[D], a CC[D], an RC[D], or an XC instruction is executed using the AOV and ANOV conditions. The RSBX instruction can also clear this bit.
9	OVB	0	Overflow flag for accumulator B. OVB is set to 1 when an overflow occurs in either the ALU or the multiplier's adder and the destination for the result is accumulator B. Once an overflow occurs, OVB remains set until either a reset, a BC[D], a CC[D], an RC[D], or an XC instruction is executed using the BOV and BNOV conditions. This RSBX instruction can also clear this bit.
8 – 0	DP	0	Data-memory page pointer. This 9-bit field is concatenated with the seven LSBs of an instruction word to form a direct-memory address of 16 bits for single data-memory operand addressing. This operation is done if the compiler mode bit in ST1 (CPL) = 0. The DP field can be loaded by the LD instruction with a short-immediate operand or from data memory.

Figure 4–2. Status Register 1 (ST1) Diagram

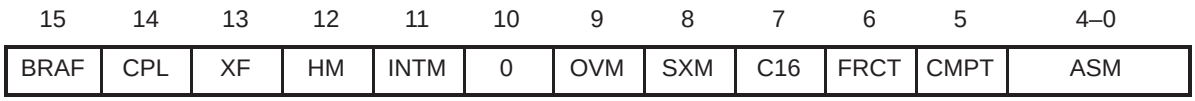


Table 4–2. Status Register 1 (ST1) Bit Summary

Bit	Name	Reset Value	Function
15	BRAF	0	Block-repeat active flag. BRAF indicates whether a block repeat is currently active. BRAF = 0 The block repeat is deactivated. BRAF is cleared when the block-repeat counter (BRC) decrements below 0. BRAF = 1 The block repeat is active. BRAF is automatically set when an RPTB instruction is executed.
14	CPL	0	Compiler mode. CPL indicates which pointer is used in relative direct addressing: CPL = 0 The relative direct-addressing mode using the data page pointer (DP) is selected. CPL = 1 The relative direct-addressing mode using the stack pointer (SP) is selected.
13	XF	1	XF status. XF indicates the status of the external flag (XF) pin, which is a general-purpose output pin. The SSBX instruction can set XF and the RSBX instruction can reset XF.
12	HM	0	Hold mode. HM indicates whether the processor continues internal execution when acknowledging an active $\overline{\text{HOLD}}$ signal: HM = 0 The processor continues execution from internal program memory but places its external interface in the high-impedance state. HM = 1 The processor halts internal execution.
11	INTM	1	Interrupt mode. INTM globally masks or enables all interrupts. INTM = 0 All unmasked interrupts are enabled. INTM = 1 All maskable interrupts are disabled. The SSBX instruction sets INTM and the RSBX instruction resets INTM. INTM is set to 1 by reset or when a maskable interrupt trap is taken (INTR or external interrupts). INTM is cleared to 0 when a RETE or RETF instruction (return from interrupt) is executed. INTM does not affect the nonmaskable interrupts ($\overline{\text{RS}}$ and $\overline{\text{NMI}}$). INTM cannot be set by memory-write operations.
10		0	Always read as 0.

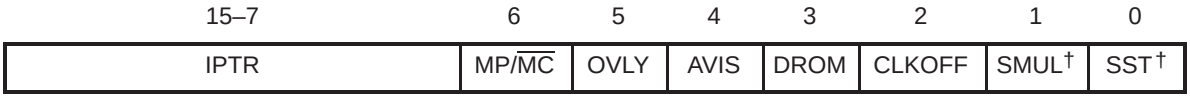
Table 4–2. Status Register 1 (ST1) Bit Summary (Continued)

Bit	Name	Reset Value	Function
9	OVM	0	Overflow mode. OVM determines what is loaded into the destination accumulator when an overflow occurs: OVM = 0 An overflowed result from either the ALU or the multiplier's adder overflows normally in the destination accumulator. OVM = 1 The destination accumulator is set to either the most positive value (00 7FFF FFFFh) or the most negative value (FF 8000 0000h) upon encountering an overflow. The SSBX and RSBX instructions set and reset OVM, respectively.
8	SXM	1	Sign-extension mode. SXM determines whether sign extension is performed: SXM = 0 Sign extension is suppressed. SXM = 1 Data is sign extended before being used by the ALU. SXM does not affect the definitions of certain instructions: the ADDS, LDU, and SUBS instructions suppress sign extension regardless of SXM value. The SSBX and RSBX instructions set and reset SXM, respectively.
7	C16	0	Dual 16-Bit/double-precision arithmetic mode. C16 determines the arithmetic mode of the ALU's operation: C16 = 0 The ALU operates in double-precision arithmetic mode. C16 = 1 The ALU operates in dual 16-bit arithmetic mode.
6	FRCT	0	Fractional mode. When FRCT is 1, the multiplier output is left-shifted by one bit to compensate for an extra sign bit.
5	CMPT	0	Compatibility mode. CMPT determines the compatibility mode for the ARP: CMPT = 0 ARP is not updated in indirect addressing mode with a single data-memory operand. ARP must always be set to 0 when the DSP is in this mode. CMPT = 1 ARP is updated in indirect addressing mode with a single data-memory operand, except when the instruction is selecting auxiliary register 0 (AR0).
4 – 0	ASM	0	Accumulator shift mode. The 5-bit ASM field specifies a shift value within a –16 through 15 range and is coded as a 2s-complement value. Instructions with a parallel store, as well as STH, STL, ADD, SUB, and LD, use this shift capability. ASM can be loaded from data memory or by the LD instruction using a short-immediate operand.

4.1.2 Processor Mode Status Register (PMST)

The PMST register is loaded with memory-mapped register instructions such as STM. The PMST bits are shown in Figure 4–3 and described in Table 4–3.

Figure 4–3. Processor Mode Status Register (PMST) Diagram



† Only on the LP devices; reserved bits on all other devices

Table 4–3. Processor Mode Status Register (PMST) Bit Summary

Bit	Name	Reset Value	Function
15 – 7	IPTR	1FFh	Interrupt vector pointer. The 9-bit IPTR field points to the 128-word program page where the interrupt vectors reside. You can remap the interrupt vectors to RAM for boot-loaded operations. At reset, these bits are all set to 1; the reset vector always resides at address FF80h in program-memory space. The RESET instruction does not affect this field.
6	MP/$\overline{MC}$	MP/$\overline{MC}$ pin	Microprocessor/microcomputer mode. MP/$\overline{MC}$ enables/disables the on-chip ROM to be addressable in program memory space. MP/$\overline{MC}$ = 0 The on-chip ROM is enabled and addressable. MP/$\overline{MC}$ = 1 The on-chip ROM is not available. MP/$\overline{MC}$ is set to the value corresponding to the logic level on the MP/$\overline{MC}$ pin when sampled at reset. This pin is not sampled again until the next reset. The RESET instruction does not affect this bit. This bit can also be set or cleared by software.
5	OVLY	0	RAM overlay. OVLY enables on-chip dual-access data RAM blocks to be mapped into program space. The values for the OVLY bit are: OVLY = 0 The on-chip RAM is addressable in data space but not in program space. OVLY = 1 The on-chip RAM is mapped into program space and data space. Data page 0 (addresses 0h to 7Fh), however, is not mapped into program space.

† Only on the LP devices; reserved bits on all other devices

Table 4–3. Processor Mode Status Register (PMST) Bit Summary (Continued)

Bit	Name	Reset Value	Function
4	AVIS	0	Address visibility mode. AVIS enables/disables the internal program address to be visible at the address pins. AVIS = 0 — The external address lines do not change with the internal program address. Control and data lines are not affected and the address bus is driven with the last address on the bus. AVIS = 1 — This mode allows the internal program address to appear at the pins of the ’54x so that the internal program address can be traced. Also, it allows the interrupt vector to be decoded in conjunction with $\overline{\text{IACK}}$ when the interrupt vectors reside in on-chip memory.
3	DROM	0	Data ROM. DROM enables on-chip ROM to be mapped into data space. The values for the DROM bit are: DROM = 0 — The on-chip ROM is not mapped into data space. DROM = 1 — A portion of the on-chip ROM is mapped into data space. See Chapter 3, *Memory*, for details.
2	CLKOFF	0	CLOCKOUT off. When the CLKOFF bit is 1, the output of CLKOUT is disabled and remains at a high level.
1	SMUL†	N/A	Saturation on multiplication. When SMUL = 1, saturation of a multiplication result occurs before performing the accumulation in a MAC or MAS instruction. The SMUL bit applies only when OVM = 1 and FRCT = 1. SMUL bit allows the MAC and MAS operations to be consistent with MAC and MAS basic operation defined in ETSI GSM specifications (GSM specs 6.06, 6.10, 6.53). The effect is that the result of 8000h × 8000h is saturated to 7FF FFFFh in fractional mode, before performing subsequent addition/subtraction required by a MAC or MAS instruction. In this mode, the MAC instruction is equivalent to MPY + ADD when OVM=1. If the mode is not set and OVM = 1, the result of the multiplication is not saturated before performing the addition/subtraction, only the results of the MAC and MAS instructions are saturated. See Example 4–1 on page 4-9 for examples of saturation on multiplication operations.

† Only on the LP devices; reserved bits on all other devices

Table 4–3. Processor Mode Status Register (PMST) Bit Summary (Continued)

Bit	Name	Reset Value	Function
0	SST†	N/A	Saturation on store. When SST 1, saturation of the data from the accumulator is enabled before storing in memory. The saturation is performed after the shift operation. Saturation on store takes place with the following instructions: STH, STL, STLM, DST, ST||ADD, ST||LD, ST||MACR[R], ST||MAS[R], ST||MPY, and ST||SUB. The following steps are performed when using saturate on store: 1) A 40-bit data value is shifted (right or left) depending on the instruction. The shift is the same as described in the SFTA instruction and depends on the SXM bit. 2) The 40-bit data value is saturated to a 32-bit value; the saturation depends on the SXM bit (the number is always assumed to be positive). If SXM = 0, the following 32-bit value is generated: ■ 7FFF FFFFh if the value is greater than 7FFF FFFFh If SXM = 1, the following 32-bit value is generated: ■ 7FFF FFFFh if the value is greater than 7FFF FFFFh ■ 8000 0000h if the value is less than 8000 0000h 3) The data is stored in memory depending upon instruction. 4) The accumulator contents remain unchanged during the operation. See Example 4–2 on page 4-10 for examples of saturation on store operations.

† Only on the LP devices; reserved bits on all other devices

Example 4–1. Use of SMUL Bit

```
MAC    *AR1+, A      ;SMUL=1, FRCT=1, OVM=1, SXM=1
```

Before Instruction		After Instruction	
A	FF FFFF FFFFh	A	00 7FFF FFFEh
T	8000h	T	8000h
AR1	100h	AR1	101h
Data Memory		Data Memory	
100h	8000h	100h	8000h

```
MAC    *AR1+, A      ;SMUL=0, FRCT=1, OVM=1, SXM=1
```

Before Instruction		After Instruction	
A	FF FFFF FFFFh	A	00 7FFF FFFFh
T	8000h	T	8000h
AR1	101h	AR1	102h
Data Memory		Data Memory	
100h	8000h	100h	8000h

Example 4–2. Use of SST Bit

```
STH A, –4, *AR1+ ;SXM=1, SST=1
```

	Before Instruction		After Instruction
A	7F FFFF 0000h	A	7F FFFF 0000h
AR1	100h	AR1	101h
Data Memory		Data Memory	
100h	5555h	100h	7FFFh

```
STL A, –4, *AR1+ ;SXM=1, SST=1
```

	Before Instruction		After Instruction
A	7F FFFF 0000h	A	7F FFFF 0000h
AR1	101h	AR1	102h
Data Memory		Data Memory	
101h	0 AAAAh	101h	0 FFFFh

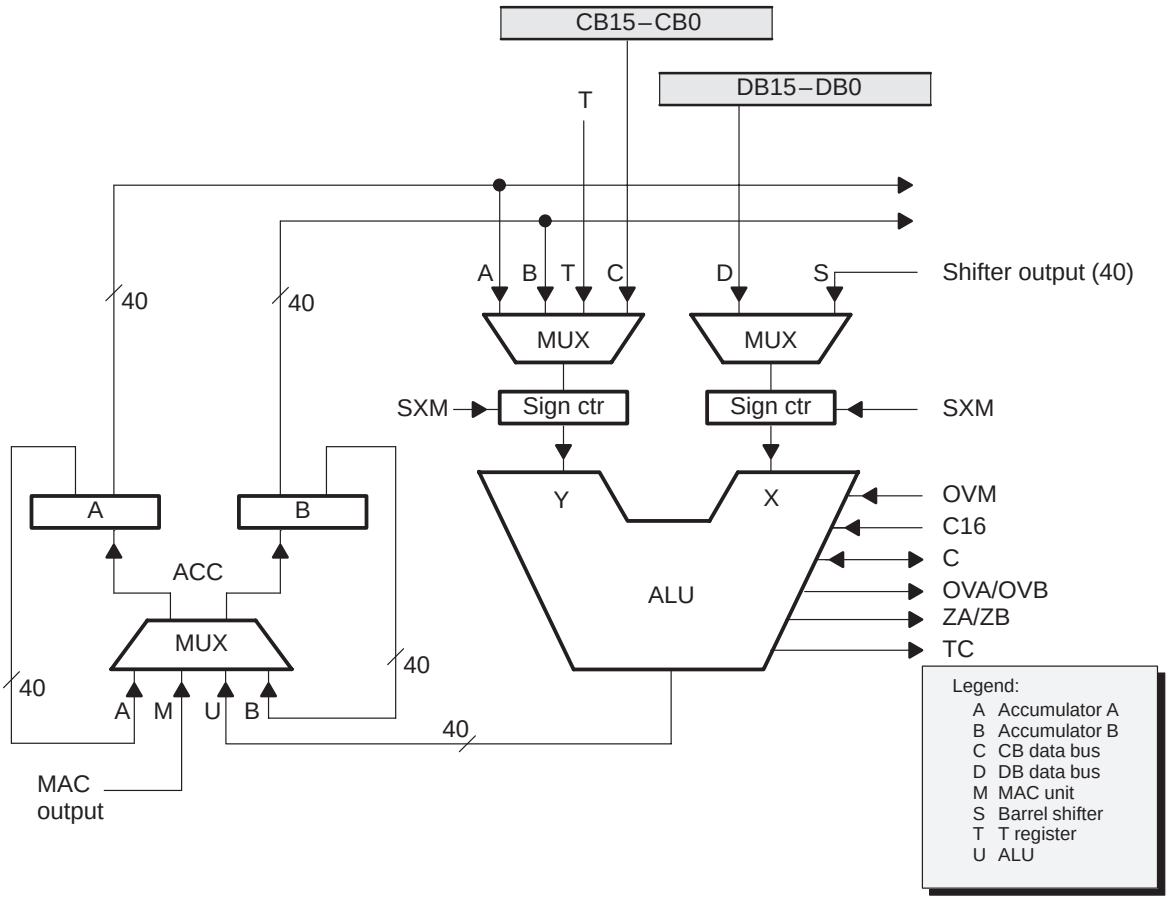
```
DST B, *AR3–  ;SXM=0, SST=1
```

	Before Instruction		After Instruction
B	8F FFFF 0000h	B	8F FFFF 0000h
AR3	103h	AR3	101h
Data Memory		Data Memory	
102h	1234h	102h	0 FFFFh
103h	5678h	103h	0 FFFFh

4.2 Arithmetic Logic Unit (ALU)

The 40-bit ALU, shown in Figure 4–4, implements a wide range of arithmetic and logical functions, most of which execute in a single clock cycle. After an operation is performed in the ALU, the result is usually transferred to a destination accumulator (accumulator A or B). Instructions that perform memory-to-memory operations (ADDM, ANDM, ORM, and XORM) are exceptions.

Figure 4–4. ALU Functional Diagram



4.2.1 ALU Input

ALU input takes several forms from several sources.

The X input source to the ALU is either of two values:

- The shifter output (a 32-bit or 16-bit data-memory operand or a shifted accumulator value)

- A data-memory operand from data bus DB

The Y input source to the ALU is any of three values:

- The value in one of the accumulators (A or B)
- A data-memory operand from data bus CB
- The value in the T register

When a 16-bit data-memory operand is fed through data bus CB or DB, the 40-bit ALU input is constructed in one of two ways:

- If bits 15 through 0 contain the data-memory operand, bits 39 through 16 are zero filled (SXM = 0) or sign-extended (SXM = 1).
- If bits 31 through 16 contain the data-memory operand, bits 15 through 0 are zero filled, and bits 39 through 32 are either zero filled (SXM = 0) or sign extended (SXM = 1).

Table 4–4 shows how the ALU inputs are obtained for the ADD instructions, depending on the type of syntax used. The ADD instructions execute in one cycle, except for cases 4, 7, and 8 that use two words and execute in two cycles.

Table 4–4. ALU Input Selection for ADD Instructions

Case	Instruction Syntax		Words	A	B	DB	CB	Shift
1	ADD	*AR1, A	1	√				√
2	ADD	*AR3, TS, A	1	√				√
3	ADD	*AR2, 16, B, A	1		√			√
4	ADD	*AR1, 8, B, A	2		√			√
5	ADD	*AR2, 8, B	1		√			√
6	ADD	*AR2, *AR3, A	1			√	√	
7	ADD	#1234h, 6, A, B	2	√				√
8	ADD	#1234h, 16, A, B	2	√				√
9	ADD	A, 12, B	1		√			√
10	ADD	B, ASM, A	1	√				√
11	DADD	*AR2, A, B	1	√				√

4.2.2 Overflow Handling

The ALU saturation logic prevents a result from overflowing by keeping the result at a maximum (or minimum) value. This feature is useful for filter calculations. The logic is enabled when the overflow mode bit (OVM) in status register ST1 is set.

When a result overflows:

- If OVM = 0, the accumulators are loaded with the ALU result without modification.
- If OVM = 1, the accumulators are loaded with either the most positive 32-bit value (00 7FFF FFFFh) or the most negative 32-bit value (0FF 8000 0000h), depending on the direction of the overflow.
- The overflow flag (OVA/OVB) in status register ST0 is set for the destination accumulator and remains set until one of the following occurs:
  - A reset is performed.
  - A conditional instruction (such as a branch, a return, a call, or an execute) is executed on an overflow condition.
  - The overflow flag (OVA/OVB) is cleared.

Note:

You can saturate the accumulator by using the SAT instruction, regardless of the value of OVM.

4.2.3 The Carry Bit

The ALU has an associated carry bit (C) that is affected by most arithmetic ALU instructions, including rotate and shift operations. The carry bit supports efficient computation of extended-precision arithmetic operations. The carry bit is not affected by loading the accumulator, performing logical operations, or executing other nonarithmetic or control instructions, so it can be used for overflow management.

Two conditional operands, C and NC, enable branching, calling, returning, and conditionally executing according to the status (set or cleared) of the carry bit. Also, the RSBX and SSBX instructions can be used to load the carry bit. The carry bit is set on a hardware reset.

4.2.4 Dual 16-Bit Mode

For arithmetic operations, the ALU can operate in a special dual 16-bit arithmetic mode that performs two 16-bit operations (for instance, two additions or two subtractions) in one cycle. You can select this mode by setting the C16 field of ST1. This mode is especially useful for the Viterbi add/compare/select operation (see Section 4.6, *Compare, Select, and Store Unit (CSSU)*, on page 4-26).

4.3 Accumulators A and B

Accumulator A and accumulator B can be configured as the destination registers for either the multiplier/adder unit or the ALU. In addition, they are used for MIN and MAX instructions or for the parallel instruction LD||MAC, in which one accumulator loads data and the other performs computations.

Each accumulator is split into three parts, as shown in Figure 4–5 and Figure 4–6.

Figure 4–5. Accumulator A

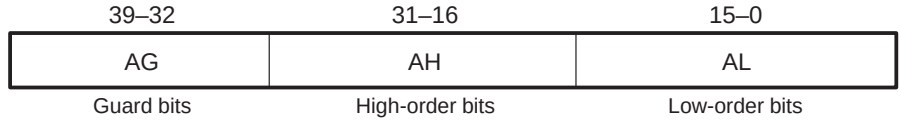
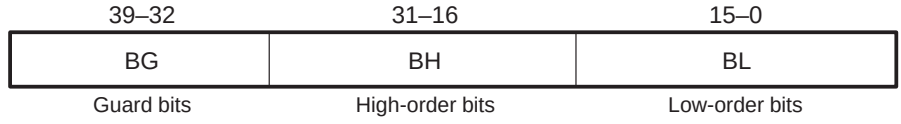


Figure 4–6. Accumulator B



The guard bits are used as a headmargin for computations. Headmargins allow you to prevent some overflow in iterative computations such as autocorrelation.

AG, BG, AH, BH, AL, and BL are memory-mapped registers that can be pushed onto and popped from the stack for context saves and restores by using PSHM and POPM instructions. These registers can also be used by other instructions that use memory-mapped registers (MMR) for page 0 addressing. The only difference between accumulators A and B is that bits 32–16 of A can be used as an input to the multiplier in the multiplier/adder unit.

4.3.1 Storing Accumulator Contents

You can store accumulator contents in data memory by using the STH, STL, STLM, and SACCD instructions or by using parallel-store instructions. To store the 16 MSBs of the accumulator in memory with a shift, use the STH, SACCD, and parallel-store instructions. For right-shift operations, bits from AG and BG shift into AH and BH. For left-shift operations, bits from AL and BL shift into AH and BH, respectively.

To store the 16 LSBs of the accumulator in memory with a shift, use the STL instruction. For right-shift operations, bits from AH and BH shift into AL and BL,

respectively, and the LSBs are lost. For left-shift operations, the bits in AL and BL are filled with zeros. Since shift operations are performed in the shifter, the contents of the accumulator remain unchanged. Example 4–3 shows the result of accumulator store operations with shift; it assumes that accumulator A = 0FF 4321 1234h.

Example 4–3. Accumulator Store With Shift

```
STH  A,8,TEMP    ; TEMP = 2112h
STH  A,-8,TEMP   ; TEMP = FF43h
STL  A,8,TEMP    ; TEMP = 3400h
STL  A,-8,TEMP   ; TEMP = 2112h
```

4.3.2 Accumulator Shift and Rotate Operations

The following instructions shift or rotate the contents of the accumulator through the carry bit:

- SFTA (shift arithmetically)
- SFTL (shift logically)
- SFTC (shift conditionally)
- ROL (rotate accumulator left)
- ROR (rotate accumulator right)
- ROLTC (rotate accumulator left with TC)

In SFTA and SFTL, the shift count is defined as $-16 \leq \text{SHIFT} \leq 15$. SFTA is affected by the SXM bit. When SXM = 1 and SHIFT is a negative value, SFTA performs an arithmetic right shift and maintains the sign of the accumulator. When SXM = 0, the MSBs of the accumulator are zero filled. SFTL is not affected by the SXM bit; it performs the shift operation for bits 31–0, shifting 0s into the MSBs or LSBs, depending on the direction of the shift.

SFTC performs a 1-bit left shift when both bits 31 and 30 are 1 or both are 0. This normalizes 32 bits of the accumulator by eliminating the most significant nonsign bit.

ROL rotates each bit of the accumulator to the left by one bit, shifts the value of the carry bit into the LSB of the accumulator, shifts the value of the MSB of the accumulator into the carry bit, and clears the accumulator's guard bits.

ROR rotates each bit of the accumulator to the right by one bit, shifts the value of the carry bit into the MSB of the accumulator, shifts the value of the LSB of the accumulator into the carry bit, and clears the accumulator's guard bits.

The ROLTC instruction (rotate accumulator left with TC) rotates the accumulator to the left and shifts the test control (TC) bit into the LSB of the accumulator.

4.3.3 Saturation Upon Accumulator Store (Available on LP Devices)

The SST bit in PMST determines whether or not the data in an accumulator is saturated before storing it in memory. The saturation is performed after the shift operation. Saturation on store is available with ten instructions:

- ❏ STH
- ❏ STL
- ❏ STLM
- ❏ DST
- ❏ ST || ADD
- ❏ ST || LD
- ❏ ST || MAC[R]
- ❏ ST || MAS[R]
- ❏ ST || MPY
- ❏ ST || SUB

The following steps are performed when saturating upon accumulator store:

1) The 40-bit data value is shifted (right or left) depending on the instruction. The shift is the same as described in the SFTA instruction and depends on the value of the SXM bit.

2) The 40-bit value is saturated to a 32-bit value. The saturation depends on the value of the SXM bit (the number is always assumed to be positive):

   - SXM = 0. 7FFF FFFFh is generated if the 40-bit value is greater than or equal to 7FFF FFFFh.
   - SXM = 1. 7FFF FFFFh is generated if the 40-bit value is greater than 7FFF FFFFh. 8000 0000h is generated if the 40-bit value is less than 8000 0000h.

3) The data is stored in memory depending on the instruction (either 16-bit LSB, 16-bit MSB, or 32-bit data).

The accumulator remains unchanged during this process.

4.3.4 Application-Specific Instructions

Each accumulator is dedicated to specific operations in application-specific instructions with parallel operations. These include symmetrical FIR filter operations using the FIRS instruction, adaptive filter operations using the LMS instruction, Euclidean distance calculations using the SQDST instruction, and other parallel operations:

- ❏ FIRS performs operations for symmetric FIR filters by using multiply/accumulates (MACs) in parallel with additions.
- ❏ LMS performs a MAC and a parallel add with rounding to efficiently update the coefficients in an FIR filter.
- ❏ SQDST performs a MAC and a subtract in parallel to calculate Euclidean distance.

FIRS multiplies accumulator A(32–16) with a program-memory value addressed by a program-memory address and adds the result to the value in accumulator B. At the same time, it adds the memory operands Xmem and Ymem, shifts the result left 16 bits, and loads this value into accumulator A.

In the LMS instruction, accumulator B stores the interim results of the input sequence convolution and filter coefficients; accumulator A updates the filter coefficients. Accumulator A can also be used as an input for MAC, which contributes to single-cycle execution of instructions with parallel operations.

The SQDST instruction computes the square of the distance between two vectors. Accumulator A(32–16) is squared and the product is added to accumulator B. The result is stored in accumulator B. At the same time, Ymem is subtracted from Xmem and the difference is stored in accumulator A. The value that is squared is the value of the accumulator before the subtraction, *Ymem – Xmem*, is executed.

4.4 Barrel Shifter

The barrel shifter is used for scaling operations such as:

- Prescaling an input data-memory operand or the accumulator value before an ALU operation
- Performing a logical or arithmetic shift of the accumulator value
- Normalizing the accumulator
- Postscaling the accumulator before storing the accumulator value into data memory

The 40-bit shifter (see Figure 4–7 on page 4-20) is connected as follows:

- The input is connected to:
  - DB for a 16-bit data input operand
  - DB and CB for a 32-bit data input operand
  - Either one of the two 40-bit accumulators
- The output is connected to:
  - One of the ALU inputs
  - The EB bus through the MSW/LSW write select unit

The SXM bit controls signed/unsigned extension of the data operands; when the bit is set, sign extension is performed. Some instructions, such as LDU, ADDS, and SUBS operate with unsigned memory operands and do not perform sign extension, regardless of the SXM value.

The shift count determines how many bits to shift. Positive shift values correspond to left shifts, whereas negative values correspond to right shifts. The shift count is specified as a 2s-complement value in several ways, depending on the instruction type. An immediate operand, the accumulator shift mode (ASM) field of ST1, or T can be used to define the shift count:

- A 4 or 5-bit immediate value specified in the operand of an instruction represents a shift count value in the –16 to 15 range. For example:

```
ADD    A,-4,B     ; Add accumulator A (right-shifted
                  ; 4 bits) to accumulator B
                  ; (one word, one cycle).
SFTL   A,+8       ; Shift (logical) accumulator A eight
                  ; bits left (one word, one cycle)
```

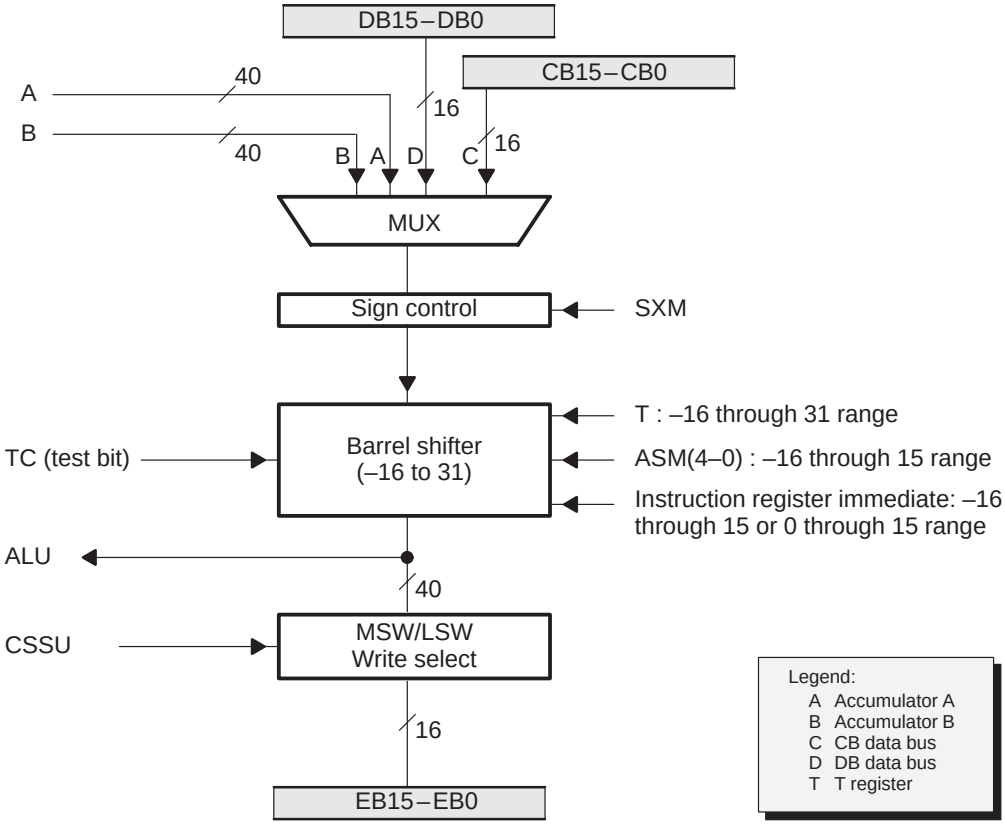
- The ASM value represents a shift count value in the –16 to 15 range and can be loaded by the LD instruction (with an immediate operand or with a data-memory operand). For example:

```
ADD    A, ASM, B     ; Add accumulator A to accumulator B
                     ; with a shift specified by ASM
```

- The six LSBs of T represent a shift count value in the –16 to 31 range. For example:

```
NORM   A            ; Normalize accumulator A (T
                    ; contains the exponent value)
```

Figure 4–7. Barrel Shifter Functional Diagram



4.5 Multiplier/Adder Unit

The '54x CPU has a 17-bit × 17-bit hardware multiplier coupled to a 40-bit dedicated adder. This multiplier/adder unit provides multiply and accumulate (MAC) capability in one pipeline phase cycle. The multiplier/adder unit is shown in Figure 4–8 on page 4-22.

The multiplier can perform signed, unsigned, and signed/unsigned multiplication with the following constraints:

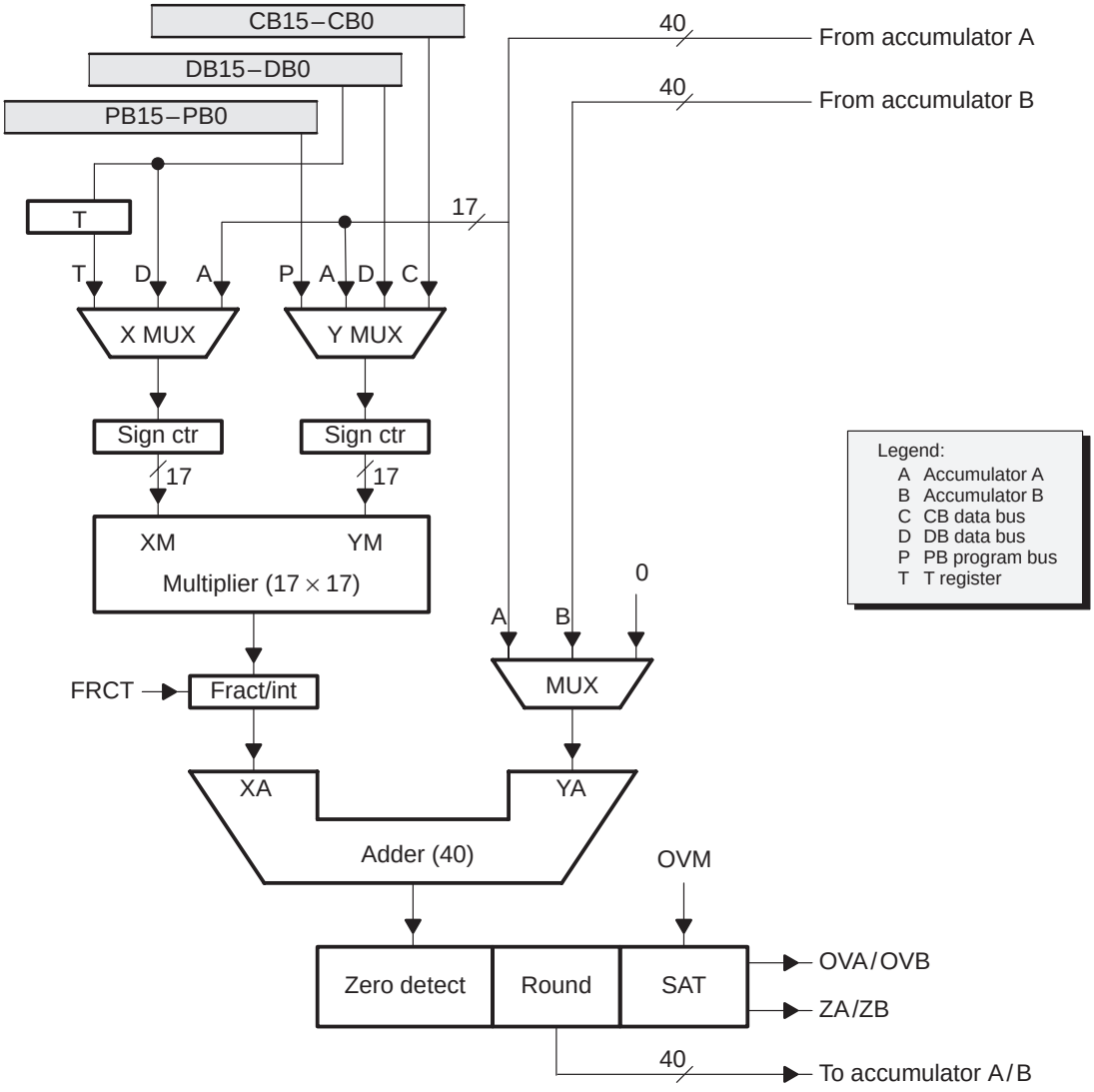
- ❏ For signed multiplication, each 16-bit memory operand is assumed to be a 17-bit word with sign extension.
- ❏ For unsigned multiplication, a 0 is added to the MSB (bit 16) in each input operand.
- ❏ For signed/unsigned multiplication, one of the operands is sign extended, and the other is extended with a 0 in the MSB (zero filled).

The multiplier output can be shifted left by one bit to compensate for the extra sign bit generated by multiplying two 16-bit 2s-complement numbers in fractional mode. (Fractional mode is selected when the FRCT bit = 1 in ST1.)

The adder in the multiplier/adder unit contains a zero detector, a rounder (2s complement), and overflow/saturation logic. Rounding consists of adding 2^{15} to the result and then clearing the lower 16 bits of the destination accumulator. Rounding is performed in some multiply, MAC, and multiply/subtract (MAS) instructions when the suffix R is included with the instruction. The LMS instruction also rounds to minimize quantization errors in updated coefficients.

The adder's inputs come from the multiplier's output and from one of the accumulators. Once any multiply operation is performed in the unit, the result is transferred to a destination accumulator (A or B).

Figure 4–8. Multiplier/Adder Functional Diagram



4.5.1 Multiplier Input Sources

This subsection lists sources for multiplier inputs and discusses how multiplier inputs can be selected for various instructions.

The XM input source to the multiplier is any of the following values:

- ❏ The temporary register (T)
- ❏ A data-memory operand from data bus DB
- ❏ Accumulator A bits 32 – 16

The YM input source to the multiplier is any of the following values:

- A data-memory operand from data bus DB
- A data-memory operand from data bus CB
- A program-memory operand from program bus PB
- Accumulator A bits 32 – 16

Table 4–5 shows how the multiplier inputs are obtained for several instructions. There are a total of nine combinations of multiplier inputs that are actually used.

For instructions using T as one input, the second input may be obtained as an immediate value or from data memory via a data bus (DB), or from accumulator A.

For instructions using single data-memory operand addressing, one operand is fed into the multiplier via DB. The second operand may come from T, as an immediate value or from program memory via PB, or from accumulator A.

For instructions using dual data-memory operand addressing, DB and CB carry the data into the multiplier.

The last two cases are used with the FIRS instruction and the SQUR and SQDST instructions. The FIRS instruction obtains inputs from PB and accumulator A. The SQUR and SQDST obtain both inputs from accumulator A.

Table 4–5. Multiplier Input Selection for Several Instructions

			X Multiplexer			Y Multiplexer			
Case	**Instruction Type**		**T**	**DB**	**A**	**PB**	**CB**	**DB**	**A**
1	MPY	#1234h, A	√					√	
2	MPY[R]	*AR2, A	√					√	
3	MPYA	B	√						√
4	MACP	*AR2, pmad, A		√		√			
5	MPY	*AR2, *AR3, B		√			√		
6	SQUR	*AR2, B		√				√	
7	MPYA	*AR2		√					√
8	FIRS	*AR2, *AR3, pmad			√	√			
9	SQUR	A, B			√				√

T provides one operand for multiply and multiply/accumulate instructions; the other memory operand is a single data-memory operand. T also provides an operand for multiply instructions with parallel load or parallel store, such as LD||MAC, LD||MAS, ST||MAC, ST||MAS, and ST||MPY. T can be loaded explicitly by instructions that support a memory-mapped register addressing mode or implicitly during multiply operations.

Since bits A(32–16) can be an input to the multiplier, some sequences that require storing the result of one computation in memory and feeding this result to the multiplier can be made faster. For some application-specific instructions (FIRS, SQDST, ABDST, and POLY), the contents of accumulator A can be computed by the ALU and then input to the multiplier without any overhead.

4.5.2 Multiply/Accumulate (MAC) Instructions

MAC instructions use the multiplier's computational bandwidth to simultaneously process two operands. Multiple arithmetic operations can be performed in a single cycle by the multiplier/adder unit.

In the MAC, MAS, and MACSU instructions with dual data-memory operand addressing, data can be transferred to the multiplier during each cycle via CB and DB and multiplied and added in a single cycle. Data addresses for these operands are generated by ARAU0 and ARAU1, the auxiliary register arithmetic units. For information about ARAU0 and ARAU1, see subsection 5.5.2, *ARAU and Address-Generation Operation*, on page 5-11.

In the MACD and MACP instructions, data can be transferred to the multiplier during each cycle via DB and PB. DB retrieves data from data memory, and PB retrieves coefficients from program memory. When MACD and MACP are used with repeat instructions (RPT and RPTZ), they perform single-cycle MAC operations with sequential access of data and coefficients. Data addresses are generated by ARAU0 and the program address register (PAR). The data-memory address is updated by ARAU0 according to a single data-memory operand in the indirect addressing mode; the program-memory address is incremented by PAGEN.

The repeated MACD instruction supports filtering constructs (weighted running average). While the sum-of-products is executed, the sample data is shifted in memory to make room for the next sample and to throw away the oldest sample. MAC and MACP instructions with circular addressing can also support filter implementation. The FIRS instruction implements an efficient symmetric structure for the FIR filter when circular addressing is used.

The MPYU and MACSU instructions facilitate extended-precision arithmetic operations. The MPYU instruction performs an unsigned multiplication. The unsigned contents of T are multiplied by the unsigned contents of the addressed data-memory location, and the result is placed in the specified accumulator. The MACSU instruction performs a signed/unsigned multiplication and addition. The unsigned contents of one data-memory location are multiplied by the signed contents of another data-memory location, and the result is added to the accumulator. This operation allows operands greater than 16 bits to be broken down into 16-bit words and then processed separately to generate products that are larger than 32 bits.

The square/add (SQURA) and square/subtract (SQURS) instructions pass the same data value to both inputs of the multiplier to square the value. The result is added to (SQURA) or subtracted from (SQURS) the accumulator at the adder level. The SQUR instruction squares a data-memory value or the contents of accumulator A.

4.5.3 MAC and MAS Saturation Upon Multiplication (Available on LP Devices)

When saturate-on-multiply is set (SMUL = 1), the MAC instruction is equivalent to MPY + ADD when OVM = 1. The effect is that the multiplication, 8000h $\times$ 8000h, is saturated to 7FFF FFFFh in fractional mode before performing the subsequent addition (MAC) or subtraction (MAS).

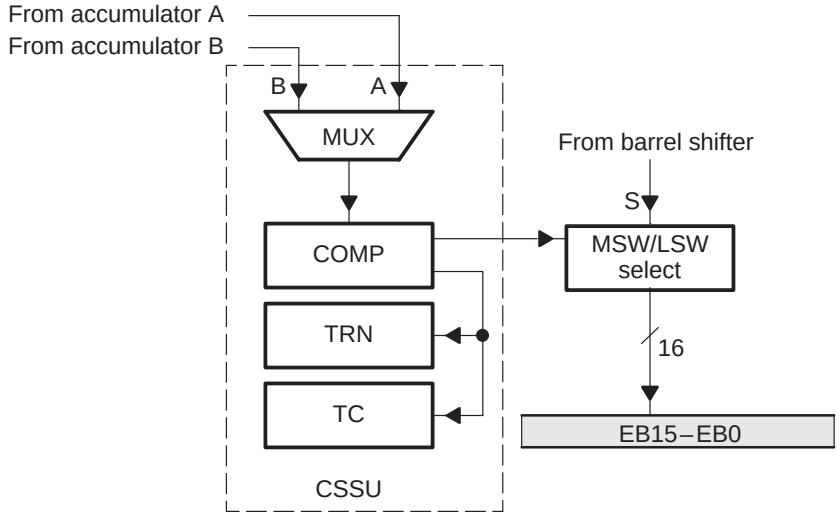
When saturate-on-multiply is not set (SMUL = 0), only the end results of MAC and MAS are saturated.

When OVM = 1 and FRCT = 1, the SMUL bit in PMST determines whether or not the result of a multiplication is saturated before the accumulation is performed in MAC and MAS instructions. This feature allows the MAC and MAS operations to be consistent with the MAC and MAS basic operation defined in ETSI GSM specifications (GSM specifications 6.06, 6.10, and 6.53).

4.6 Compare, Select, and Store Unit (CSSU)

The compare, select, and store unit (CSSU) is an application-specific hardware unit dedicated to add/compare/select (ACS) operations of the Viterbi operator. Figure 4–9 shows the CSSU, which is used with the ALU to perform fast ACS operations.

Figure 4–9. Compare, Select, and Store Unit (CSSU)

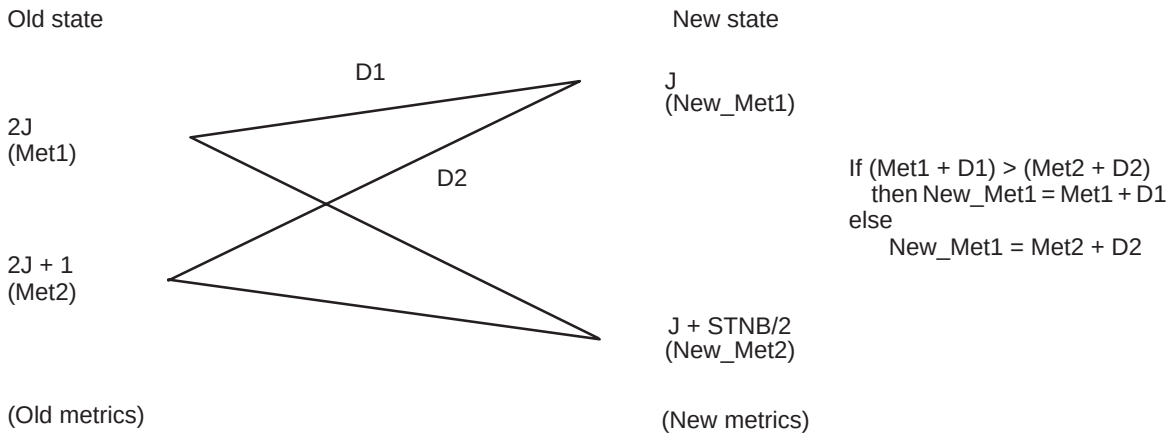


The CSSU allows the ’54x to support various Viterbi butterfly algorithms used in equalizers and channel decoders.

The add function of the Viterbi operator (see Figure 4–10) is performed by the ALU. This function consists of a double addition function (Met1 ± D1 and Met2 ± D2). Double addition is completed in one machine cycle if the ALU is configured for dual 16-bit mode by setting the C16 bit in ST1. With the ALU configured in dual 16-bit mode, all the long-word (32-bit) instructions become dual 16-bit arithmetic instructions.

T is connected to the ALU input (as a dual 16-bit operand) and is used as local storage in order to minimize memory access. Table 4–6 shows the instructions that perform dual 16-bit ALU operations.

Figure 4–10. Viterbi Operator



Legend:

	STNB	Number of states
	Met	Path metrics
	D	Branch metrics

Table 4–6. ALU Operations in Dual 16-Bit Mode

Instruction	Function (Dual 16-Bit Mode)
DADD Lmem, src [, dst]	src(31–16) + Lmem(31–16) → dst(39–16) src(15–0) + Lmem(15–0) → dst(15–0)
DADST Lmem, dst	Lmem(31–16) + T → dst(39–16) Lmem(15–0) – T → dst(15–0)
DRSUB Lmem, src	Lmem(31–16) – src(31–16) → src(39–16) Lmem(15–0) – src(15–0) → src(15–0)
DSADT Lmem, dst	Lmem(31–16) – T → dst(39–16) Lmem(15–0) + T → dst(15–0)
DSUB Lmem, src	src(31–16) – Lmem(31–16) → src(39–16) src(15–0) – Lmem(15–0) → src(15–0)
DSUBT Lmem, dst	Lmem(31–16) – T → dst(39–16) Lmem(15–0) – T → dst(15–0)

Legend:

→	Is stored to
Lmem	Long (32-bit) data-memory value
src	Source accumulator (A or B)
dst	Destination accumulator (A or B)
x(n–m)	Read as bits n through m of x

The CSSU implements the compare and select operation via the CMPS instruction, a comparator, and the 16-bit transition register (TRN). This operation compares two 16-bit parts of the specified accumulator and shifts the decision into bit 0 of TRN. This decision is also stored in the TC bit of ST0. Based on the decision, the corresponding 16-bit part of the accumulator is stored in data memory. Example 4–4 shows the compare and select operation executed by the CMPS instruction.

Example 4–4. CMPS Instruction Operation

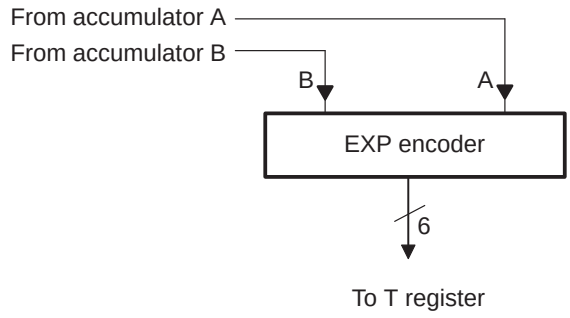
```
CMPS   B,*AR3    ;if (B(31-16)>B(15-0)) then
                 ;B(31-16)->(*AR3); TRN<<1; 0->TRN(0);
                 ;0->TC}
                 ;else B(15-0)->(*AR3); TRN<<1;
                 ;1->TRN(0); 1->TC;
```

TRN contains information of the path transition decisions to new states. This information can be used for a back-tracking routine that finds the optimal path, which results in decoding the code.

4.7 Exponent Encoder

The exponent encoder is an application-specific hardware device dedicated to supporting the EXP instruction in a single cycle (see Figure 4–11). With the EXP instruction, the exponent value in the accumulator can be stored in T as a 2s-complement value within a –8 through 31 range. The exponent is defined as the number of leading redundant bits – 8, which corresponds to the number of shifts required in the accumulator to eliminate nonsignificant sign bits. This operation results in a negative value when the accumulator value exceeds 32 bits.

Figure 4–11. Exponent Encoder



The EXP and NORM instructions use the exponent encoder to normalize the accumulator's contents efficiently. NORM supports shifting the accumulator value by the number of bits specified in T in a single cycle. A negative value in T produces a right shift of the accumulator's contents, which normalizes any value beyond the 32-bit range of the accumulator. Example 4–5 demonstrates the normalization of accumulator A.

Example 4–5. Normalization of Accumulator A

```
;Normalize accumulator A
  EXP   A            ; (the number of leading bits – 8)-> T.
  ST    T, EXPONENT  ; Store the exponent (T) into data
                     ; memory
  NORM  A            ; Normalize accumulator A, (A)<<(T)
```

Chapter 5

Data Addressing

The ’54x offers seven basic addressing modes:

- *Immediate addressing* uses the instruction to encode a fixed value.
- *Absolute addressing* uses the instruction to encode a fixed address.
- *Accumulator addressing* uses an accumulator to access a location in program memory as data.
- *Direct addressing* uses seven bits of the instruction to encode an offset relative to DP or to SP. The offset plus DP or SP determine the actual address in data memory.
- *Indirect addressing* uses the auxiliary registers to access memory.
- *Memory-mapped register addressing* modifies the memory-mapped registers without affecting either the current DP value or the current SP value.
- *Stack addressing* manages adding and removing items from the system stack.

Topic **Page**

5.1 Immediate Addressing

In immediate addressing, the instruction syntax contains the specific value of the operand. Two types of values can be encoded in an instruction:

- *Short immediate values* can be 3, 5, 8, or 9 bits in length.
- *16-bit immediate values* are always 16 bits in length.

Immediate values can be encoded in 1-word or 2-word instructions. The 3-, 5-, 8-, or 9-bit values are encoded into 1-word instructions; 16-bit values are encoded into 2-word instructions.

The length of the immediate value encoded in an instruction depends on the type of instruction used. Table 5–1 lists the ’54x instructions that can encode immediate values in their instruction word(s). The table also gives the bit value that can be encoded in the instruction.

Table 5–1. Instructions That Allow Immediate Addressing

3- and 5-Bit Constants	8-Bit Constant	9-Bit Constant	16-Bit Constant	
LD	FRAME	LD	ADD	ORM
	LD		ADDM	RPT
	RPT		AND	RPTZ
			ANDM	ST
			BITF	STM
			CMPM	SUB
			LD	XOR
			MAC	XORM
			OR	

The syntax for immediate addressing uses a number sign (#) immediately preceding the value or symbol to indicate that it is an immediate value. For example, to load accumulator A with the value 80 in hexadecimal, you would write:

```
LD #80h, A
```

Figure 5–1 and Figure 5–2 use the RPT instruction to show how an immediate value is encoded in instructions that use immediate addressing. The opcode in the instruction is encoded in the high half of the instruction: bits 8–15 of a 1-word encoding; bits 0–15 of the high word of a 2-word encoding. The value of the constant is in the rest of the space.

Figure 5–1. RPT Instruction With Short-Immediate Addressing

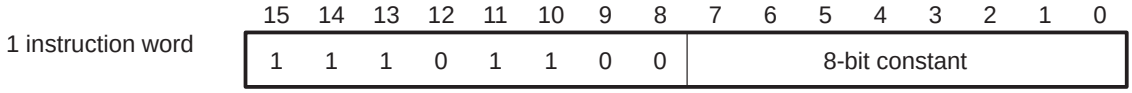
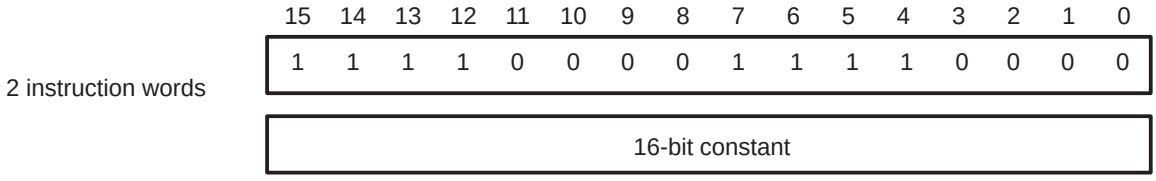


Figure 5–2. RPT Instruction With 16-Bit-Immediate Addressing



5.2 Absolute Addressing

There are four types of absolute addressing:

- *dmad* addressing:
  - MVDK *Smem*, *dmad*
  - MVDM *dmad*, *MMR*
  - MVKD *dmad*, *Smem*
  - MVMD *MMR*, *dmad*
- *pmad* addressing:
  - FIRS *Xmem*, *Ymem*, *pmad*
  - MACD *Smem*, *pmad*, *src*
  - MACP *Smem*, *pmad*, *src*
  - MVDP *Smem*, *pmad*
  - MVPD *pmad*, *Smem*
- *PA* addressing:
  - PORTR *PA*, *Smem*
  - PORTW *Smem*, *PA*
- *(*lk*) addressing is used with all instructions that support the use of a single data-memory (Smem) operand.

Absolute addresses are always encoded with a length of 16 bits, so instructions that encode absolute addresses are always at least two words in length.

5.2.1 *dmad* Addressing

Data-memory address (*dmad*) addressing uses a specific value to specify an address in data space.

The syntax for dmad addressing uses a symbol or a number to specify an address in data space. For example, to copy the value contained at the address labeled SAMPLE in data space to the memory location in data space pointed to by AR5, you would write:

```
MVKD   SAMPLE, *AR5
```

In this example, the address referenced by SAMPLE is the dmad value.

5.2.2 *pmad* Addressing

Program-memory address (*pmad*) addressing uses a specific value to specify an address in program space.

The syntax for pmad addressing uses a symbol or a number to specify an address in program space. For example, to copy a word in the program-memory location labeled TABLE to a data-memory location specified by AR7, you would write:

```
MVPD   TABLE, *AR7–
```

In this example, the address referenced by TABLE is the pmad value.

5.2.3 *PA* Addressing

Port address (*PA*) addressing uses a specific value to specify an external I/O port address.

The syntax for PA addressing uses a symbol or a number to specify the port address. For example, to copy a value from the I/O port at port address FIFO to a data-memory location pointed to by AR5, you would write:

```
PORTR FIFO, *AR5
```

In the example, FIFO refers to the port address.

5.2.4 *(*lk*) Addressing

*(*lk*) addressing uses a specific value to specify an address in data space.

The syntax for *(lk) addressing uses a symbol or a number to specify an address in data space. For example, to load accumulator A with the value contained in address BUFFER in data space, you would write:

```
LD *(BUFFER),A
```

The syntax for *(lk) addressing allows all instructions that use Smem addressing to access any location in data space without changing the DP or initializing an AR. When this form of absolute addressing is used, the length of the instruction is extended by one word. For example, a 1-word instruction would become a 2-word instruction or a 2-word instruction would become a 3-word instruction. The addition of one word to an instruction affects its usability in delay slots.

Note:

Instructions using the *(lk) form of absolute addressing cannot be used with repeat single instructions (RPT, RPTZ).

5.3 Accumulator Addressing

Accumulator addressing uses the accumulator as an address. This addressing mode is used to address program memory as data.

Two instructions allow you to use the accumulator as an address:

- READA *Smem*
- WRITA *Smem*

READA transfers a word from a program-memory location specified by accumulator A to a data-memory location specified by the single data-memory (Smem) operand of the instruction.

WRITA transfers a word from a data-memory location specified by the Smem operand of the instruction to a program-memory location specified by accumulator A.

In repeat mode, an increment may be used to increment accumulator A.

Note:

In most ’54x devices, the program-memory location is specified by the lower 16 bits of accumulator A. However, because the ’548 and ’549 have 23 address lines, the program-memory location in a ’548 or ’549 device is specified by the lower 23 bits of accumulator A. See subsection 3.1.1, *Extended Program Memory*, on page 3-8.

5.4 Direct Addressing

In direct addressing mode, the instruction contains the lower seven bits of the data-memory address (dma). The 7-bit dma is an address offset that is combined with a base address, with the data-page pointer (DP), or with the stack pointer (SP) to form a 16-bit data-memory address. Using this form of addressing, you can access any of 128 locations in random order without changing the DP or the SP.

Note:

Direct addressing is not the only method of offset addressing. However, the advantage of this mode is that it encodes each instruction and address into a single word.

Either DP or SP can be combined with the dma offset to generate the actual address. The compiler mode bit (CPL), located in status register ST1, selects which method is used to generate the address:

- When CPL = 0, the dma field is concatenated with the 9-bit DP field to form the 16-bit data-memory address.
- When CPL = 1, the dma field is added (positive offset) to SP to form the 16-bit data-memory address.

The syntax for direct addressing uses a symbol or a number to specify the offset value. For example, to add the contents of the memory location SAMPLE to accumulator B, provided that the correct base address is in DP (CPL = 0) or SP (CPL = 1), you would write:

```
ADD    SAMPLE, B
```

The lower seven bits of the address of *SAMPLE* are stored in the instruction word.

Figure 5–3 shows the opcode format for instructions that use direct addressing. Table 5–2 describes the bits of the direct-addressing instruction. Figure 5–4 illustrates how the 16-bit data address is formed.

Figure 5–3. Direct-Addressing Instruction Format

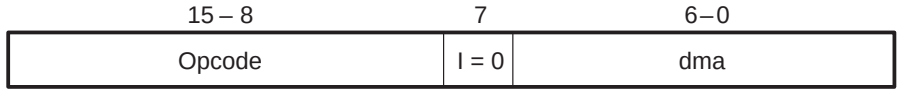
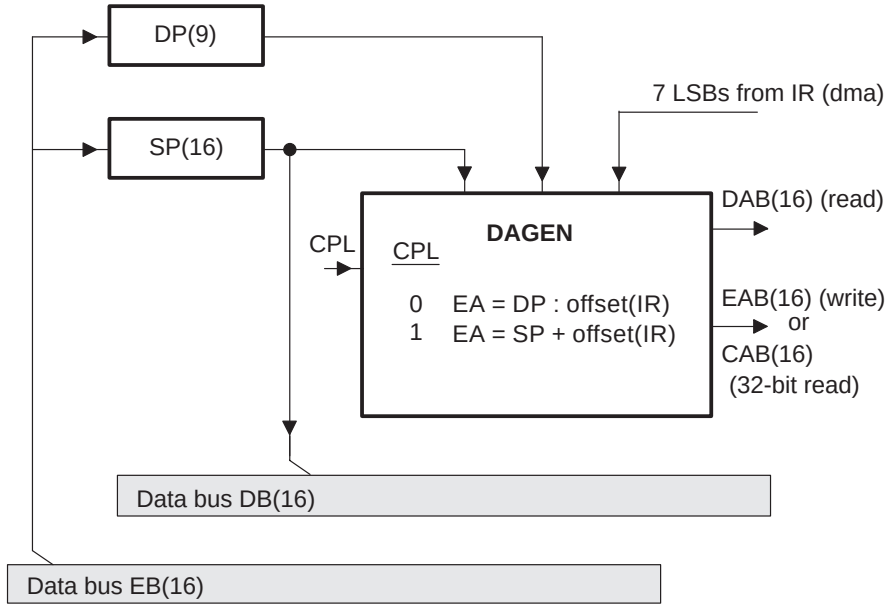


Table 5–2. Direct-Addressing Instruction Bit Summary

Bit	Name	Function
15 – 8	Opcode	This eight-bit field contains the operation code for the instruction.
7	I	I = 0, the addressing mode used by the instruction is the direct addressing mode.
6–0	dma	This seven-bit field contains the data-memory address offset for the instruction.

Figure 5–4. Direct Addressing Block Diagram

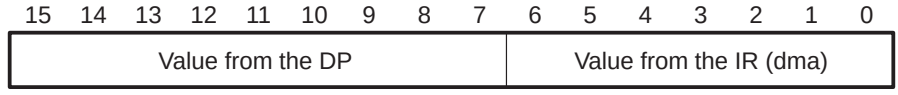


Legend: EA Effective address
IR Instruction register

5.4.1 DP-Referenced Direct Addressing

In DP-referenced direct addressing, the 7-bit dma in the instruction register is concatenated with the 9-bit DP to form the address. Figure 5–5 shows how the two values make up the resulting address.

Figure 5–5. DP-Referenced Direct Address



DP-referenced direct addressing divides memory into 512 pages, because the DP's range is from 0 to 511 ($2^9 - 1$). Each page has 128 addressable locations, because the dma ranges from 0 to 127 ($2^7 - 1$).

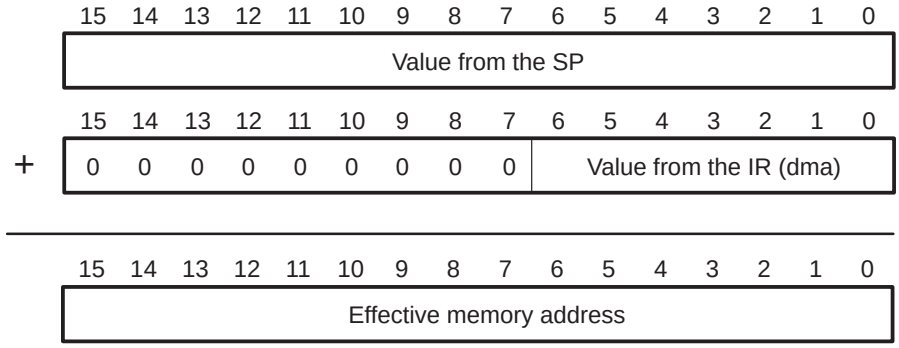
In other words, the DP points to one of 512 possible 128-word data-memory pages; the dma points to the specific location within that page. The only difference between an access to location 0 on page 1 and to location 0 on page 2 is the value of the DP.

The DP is loaded by the LD instruction.

5.4.2 SP-Referenced Direct Addressing

In SP-referenced direct addressing, the 7-bit dma in the instruction register is added as a positive offset to the SP to form the effective 16-bit data-memory address. Figure 5–6 shows how the two values combine to form the resulting address.

Figure 5–6. SP-Referenced Direct Address



The SP points to any address in memory. The dma points to the specific location on the page, allowing you to access a contiguous 128-word ($2^7 - 1$) block in memory from any base address.

SP can also add or remove items from the stack. See Section 5.7, *Stack Addressing*, for more information.

5.5 Indirect Addressing

In indirect addressing, any location in the 64K-word data space can be accessed via a 16-bit address contained in an auxiliary register. The ’54x has eight 16-bit auxiliary registers (AR0–AR7). Indirect addressing is used mainly when there is a need to step through sequential locations in memory in fixed-size steps.

When memory is addressed with indirect addressing, the auxiliary register and the address can be optionally modified by a decrement, an increment, an offset, or an index. Special modes offer circular and bit-reversed addressing. A circular buffer size register (BK) is used with circular addressing. The AR0 register is used for indexed and bit-reversed addressing modes in addition to being used to point to memory as the other auxiliary registers do.

Indirect addressing is flexible enough not only to read or write a single 16-bit data operand from memory with one instruction, but also to access two data-memory locations with one instruction. Accesses of two data-memory locations include reads of two independent memory locations, reads and writes of two consecutive memory locations, and a read of one memory location combined with a write to a memory location.

5.5.1 Single-Operand Addressing

Figure 5–7 shows the indirect-addressing instruction format for a single data-memory (Smem) operand. Table 5–3 describes the bits of the instruction.

Figure 5–7. Indirect-Addressing Instruction Format for a Single Data-Memory Operand

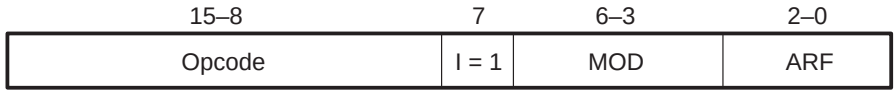


Table 5–3. Indirect-Addressing Instruction Bit Summary – Single Data-Memory Operand

Bit	Name	Function
15 – 8	Opcode	This eight-bit field contains the operation code for the instruction.
7	I	I = 1, the addressing mode used by the instruction is the indirect addressing mode.
6–3	MOD	This 4-bit modification field defines the type of indirect addressing. Subsection 5.5.3, *Single-Operand Address Modifications*, on page 5-13, describes the 16 ways to specify addressing types with the MOD field.

Table 5–3. Indirect-Addressing Instruction Bit Summary – Single Data-Memory Operand (Continued)

Bit	Name	Function
2–0	ARF	This 3-bit auxiliary register field defines the auxiliary register used for addressing. ARF depends on the compatibility mode bit (CMPT) in status register ST1: CMPT = 0 Standard mode. In standard mode, ARF always specifies the auxiliary register, regardless of the value in ARP. ARP is not updated. ARP must always be set to zero when the DSP is in this mode. CMPT = 1 Compatibility mode. In compatibility mode, ARP selects the auxiliary register if ARF = 0. Otherwise, ARF selects the auxiliary register and the ARF value is loaded into ARP when the access is completed. *AR0 in the assembly instruction indicates the auxiliary register selected by ARP in compatibility mode.

Note:

In some cases, two data operands can be fetched at once. This requires a different instruction format. Subsection 5.5.4, *Dual-Operand Address Modifications*, on page 5-19, describes this format.

5.5.2 ARAU and Address-Generation Operation

Two auxiliary register arithmetic units (ARAU0 and ARAU1) operate on the contents of the auxiliary registers. The ARAUs perform unsigned, 16-bit auxiliary register arithmetic operations. Some addresses can be obtained by premodifying the auxiliary register.

The auxiliary registers can be:

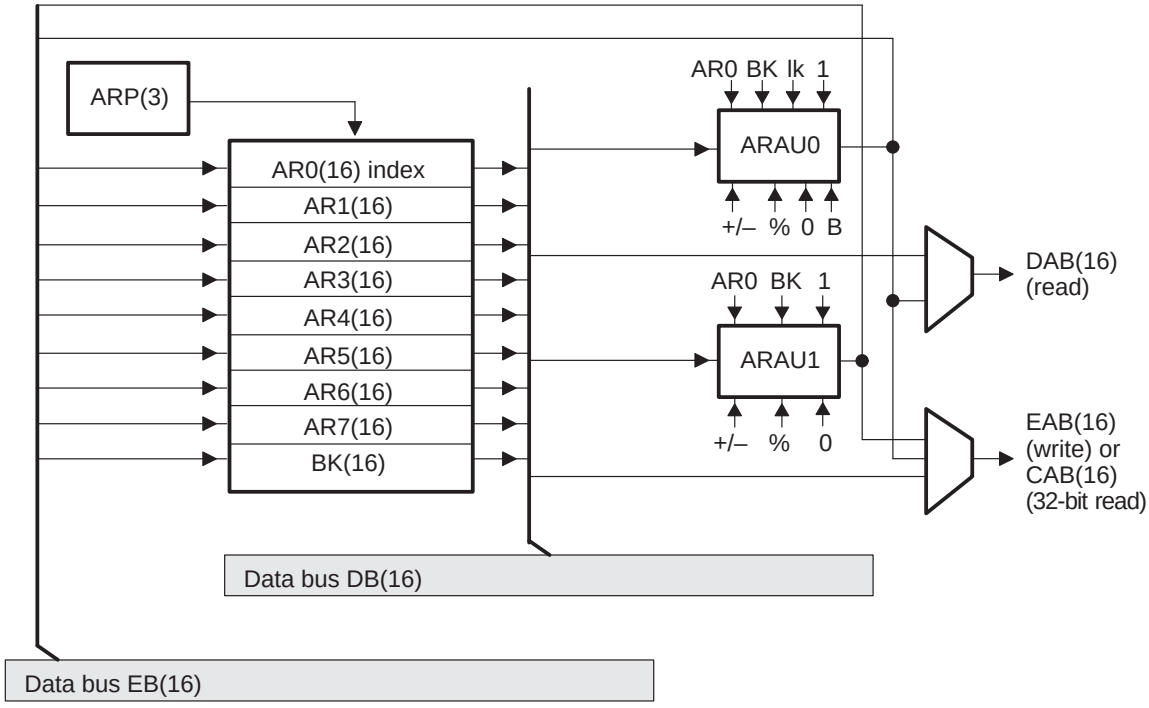
- Loaded with an immediate value using the STM instruction
- Loaded via the data bus by writing to the memory-mapped auxiliary registers
- Modified by the indirect addressing field of any instruction that supports indirect addressing
- Modified by the modify auxiliary register (MAR) instruction
- Used as loop counters using the BANZ[D] instruction

Note:

Typically, STM or MVDK is used to load auxiliary registers. Both of these instructions allow the next instruction to use the new value in the register. Other instructions that load a new value into an AR produce a pipeline latency. For further information on the pipeline and possible pipeline conflicts, see Chapter, 7 *Pipeline.*

Figure 5–8 shows the ARAUs used to generate an address in the indirect addressing mode using a single data-memory operand. As the figure shows, the main components used for address generation in indirect addressing are the auxiliary register arithmetic units (ARAU0 and ARAU1) and the auxiliary registers (AR0–AR7).

Figure 5–8. Indirect Addressing Block Diagram for a Single Data-Memory Operand



5.5.3 Single-Operand Address Modifications

You can modify the addresses you use in instructions before or after they are accessed, or you can leave them unchanged. You can modify them by incrementing or decrementing the address by 1, adding a 16-bit offset, or indexing with the value in AR0. These three types of action combined with taking the action either before or after the access, plus the ways of leaving the address unchanged make a total of 16 addressing types, each assigned to a value of MOD, the 4-bit modification field in the encoding of an instruction using indirect addressing.

Table 5–4 lists the types of single data-memory operand addressing, along with the value of MOD, the assembler syntax, and the function for each type.

Table 5–4. Indirect Addressing Types With a Single Data-Memory Operand

MOD Field	Operand Syntax	Function	Description†
0000 (0)	*ARx	addr = ARx	ARx contains the data-memory address.
0001 (1)	*ARx–	addr = ARx ARx = ARx – 1	After access, the address in ARx is decremented.‡
0010 (2)	*ARx+	addr = ARx ARx = ARx + 1	After access, the address in ARx is incremented.‡
0011 (3)	*+ARx	addr = ARx + 1 ARx = ARx + 1	The address in ARx is incremented before its use.‡§#
0100 (4)	*ARx–0B	addr = ARx ARx = B(ARx – AR0)	After access, AR0 is subtracted from ARx with reverse carry (rc) propagation.
0101 (5)	*ARx–0	addr = ARx ARx = ARx – AR0	After access, AR0 is subtracted from ARx.
0110 (6)	*ARx+0	addr = ARx ARx = ARx + AR0	After access, AR0 is added to ARx.
0111 (7)	*ARx+0B	addr = ARx ARx = B(ARx + AR0)	After access, AR0 is added to ARx with reverse carry (rc) propagation.
1000 (8)	*ARx–%	addr = ARx ARx = circ(ARx – 1)	After access, the address in ARx is decremented with circular addressing.‡
1001 (9)	*ARx–0%	addr = ARx ARx = circ(ARx – AR0)	After access, AR0 is subtracted from ARx with circular addressing.

† ARx is used as the data-memory address unless otherwise specified.
‡ Increment/decrement value is 1 for 16-bit word access and 2 for 32-bit word access.
§ This mode is not allowed in memory-mapped register addressing.
¶ This mode is discussed in greater detail in subsection 5.2.4, *(*lk*) *Addressing*, on page 5-5.
# This mode is allowed only for write accesses.

Table 5–4. Indirect Addressing Types With a Single Data-Memory Operand (Continued)

MOD Field	Operand Syntax	Function	Description†
1010 (10)	*ARx+%	addr = ARx ARx = circ(ARx + 1)	After access, the address in ARx is incremented with circular addressing.‡
1011 (11)	*ARx+0%	addr = ARx ARx = circ(ARx + AR0)	After access, AR0 is added to ARx with circular addressing.
1100 (12)	*ARx(lk)	addr = ARx + lk ARx = ARx	The sum of ARx and the 16-bit long offset (lk) is used as the data-memory address. ARx is not updated.
1101 (13)	*+ARx(lk)	addr = ARx + lk ARx = ARx + lk	The address in ARx is incremented before its use and added to the signed 16-bit long offset (lk). It is then used as the data-memory address.§
1110 (14)	*+ARx(lk)%	addr = circ(ARx + lk) ARx = circ(ARx + lk)	The address in ARx is incremented before its use and added to a signed 16-bit long offset (lk) with circular addressing. It is then used as the data-memory address.§
1111 (15)	*(lk)	addr = lk	An unsigned 16-bit long offset (lk) is used as the absolute address of data memory (absolute addressing).§¶

† ARx is used as the data-memory address unless otherwise specified.
‡ Increment/decrement value is 1 for 16-bit word access and 2 for 32-bit word access.
§ This mode is not allowed in memory-mapped register addressing.
¶ This mode is discussed in greater detail in subsection 5.2.4, *(*lk*) *Addressing*, on page 5-5.
# This mode is allowed only for write accesses.

5.5.3.1 Increment/Decrement Address Modifications (MOD = 0, 1, 2, or 3)

While an AR is being used, you can modify the AR by incrementing or decrementing its value.

The syntaxes for using the AR without modification, postdecrementing the AR by 1, postincrementing the AR by 1, and preincrementing the AR by 1 are shown in Table 5–4 for MOD = 0, 1, 2, and 3, respectively.

Preincrementing (*+ARx) is supported only in instructions that access operands in a write operation.

5.5.3.2 Offset Address Modifications (MOD = 12 or 13)

Offset addressing is a type of indirect addressing in which a predetermined offset, or step size, is added to the contents of an auxiliary register. There are two options for offset addressing. In both cases, a 16-bit long offset, which is part of the instruction, is added to the value in the auxiliary register, and the result is used to address a location in data memory. In the first case, the auxiliary register is not updated. In the second case, the auxiliary register is updated with the new address.

This type of addressing is useful in accessing a specific element of an array or structure, especially when the auxiliary register is not updated. When the auxiliary register is updated, this type of addressing is especially useful for stepping through an array in fixed-size steps.

The syntaxes for offset addressing of an AR without and with updating the AR using offset addressing are shown in Table 5–4 in MOD 12 and 13, respectively.

Notes:

1) Instructions using offset addressing cannot be repeated using the repeat single instruction.

2) Premodification by a 16-bit word offset (*+ARx(lk)) uses an extra cycle because the instruction code has two or three words. The last word is the offset.

5.5.3.3 Indexed Address Modifications (MOD = 5 or 6)

Indexed addressing is a type of indirect addressing in which the contents of AR0 are added to, or subtracted from, any other auxiliary register, ARx. Indexed addressing differs from offset addressing in that the index or step size can be determined during code execution. Because the index is determined during code execution, you can easily make adjustments to the step size. Indexed addressing also offers an advantage over offset addressing: it does not require an additional word for the instruction.

The syntaxes for subtracting AR0 from ARx and for adding AR0 to ARx are shown in Table 5–4 for MOD = 5 and 6, respectively.

5.5.3.4 Circular Address Modifications (MOD = 8, 9, 10, 11, or 14)

Many algorithms, such as convolution, correlation, and FIR filters, require the implementation of a circular buffer in memory. In these algorithms, a circular buffer is a sliding window containing the most recent data. As new data comes in, the buffer overwrites the oldest data. The key to the implementation of a circular buffer is the implementation of circular addressing.

The circular-buffer size register (BK) specifies the size of the circular buffer. A circular buffer of size R must start on a N-bit boundary (that is, the N LSBs of the base address of the circular buffer must be 0), where N is the smallest integer that satisfies $2^N > R$. The value R must be loaded into BK. For example, a 31-word circular buffer must start at an address whose five LSBs are 0 (that is, XXXX XXXX XXX0 0000_2), and the value 31 must be loaded into BK. As a second example, a 32-word circular buffer must start at an address whose six LSBs are 0 (that is, XXXX XXXX XX00 0000_2), and the value 32 must be loaded into BK. In some applications, however, it may be possible to use bit-reversed addressing to place a 2^N buffer on a 2^N boundary and offer the effect of circular addressing.

The *effective base address* (EFB) of the circular buffer is determined by zeroing the N LSBs of a user-selected auxiliary register (ARx). The *end of buffer address* (EOB) of the circular buffer is determined by replacing the N LSBs of ARx with the N LSBs of BK. The *index* of the circular buffer is simply the N LSBs of ARx and the *step* is the quantity being added to or subtracted from the auxiliary register. Follow these three rules when you use circular addressing:

- ❏ Place the first (lowest) address of the circular buffer on a 2^N boundary where 2^N is larger than the circular buffer size.
- ❏ Use a step less than or equal to the circular buffer size.
- ❏ The first time the circular queue is addressed, the auxiliary register must point to an element in the circular queue.

The algorithm for circular addressing is as follows:

```
If 0 ≤ index + step < BK:
   index = index + step.
Else if index + step ≥ BK:
   index = index + step – BK.
Else if index + step < 0:
   index = index + step + BK.
```

Circular addressing can be used for single data-memory or dual data-memory operands. When BK is zero, the circular modifier results in no circular address modification. This is especially useful when a dual operand must perform an address modification equivalent to ARx+0.

Figure 5–9 illustrates the relationships among BK, the auxiliary register (ARx), the bottom of the circular buffer, the top of the circular buffer, and the index into the circular buffer.

Figure 5–10 shows how the circular buffer is implemented and illustrates the relationship between the generated values and the elements in the circular buffer.

Figure 5–9. Circular Addressing Block Diagram

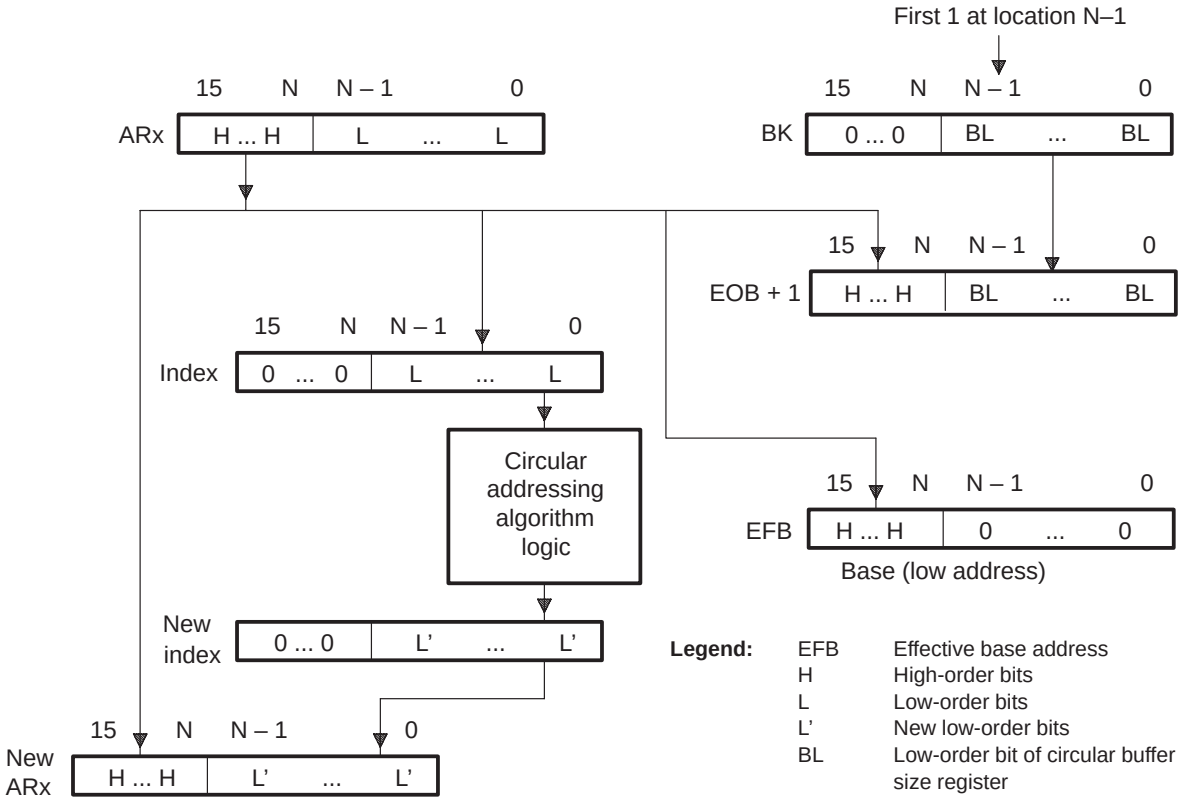
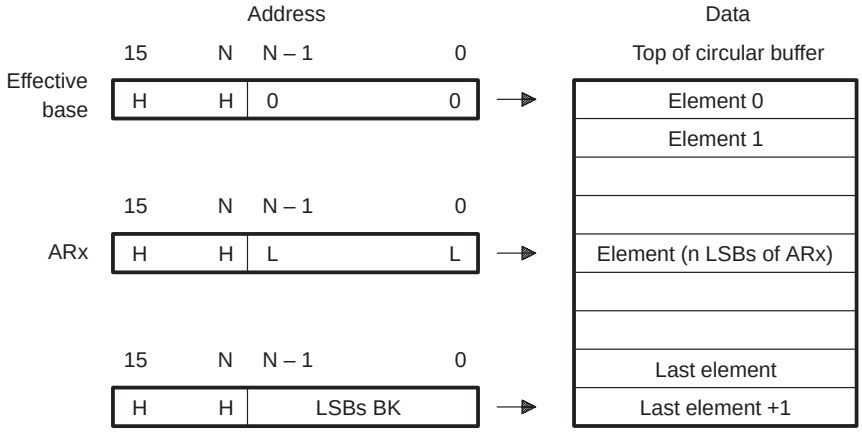


Figure 5–10. Circular Buffer Implementation



Circular addressing typically uses a decrement or an increment by one (MOD = 8 and 10) or a decrement or an increment by an index (MOD = 9 and 11). Premodification by a 16-bit word offset (*+ARx(lk)%) requires an extra code word so that the instruction code has two or three words. The last word is the offset. An instruction using indirect-offset addressing cannot be repeated using a single repeat operation.

The syntaxes for each of the five types of circular addressing are shown in Table 5–4 for MOD = 8, 9, 10, 11, and 14.

5.5.3.5 Bit-Reversed Address Modifications (MOD = 4 or 7)

Bit-reversed addressing enhances execution speed and program memory for FFT algorithms that use a variety of radixes. In this addressing mode, AR0 specifies one half of the size of the FFT. The value contained in AR0 must be equal to $2N^{-1}$, where N is an integer, and the FFT size is 2N. An auxiliary register points to the physical location of a data value. When you add AR0 to the auxiliary register using bit-reversed addressing, the address is generated in a bit-reversed fashion, with the carry bit propagating from left to right, instead of the normal right to left.

The syntaxes for each of the two bit-reversed addressing modes are shown in Table 5–4 for MOD 4 and 7, respectively.

Assume that the auxiliary registers are eight bits long, that AR2 represents the base address of the data in memory (01100000_2), and that AR0 contains the value 00001000_2. Example 5–1 shows a sequence of modifications of AR2 and the resulting values of AR2.

Example 5–1. Sequence of Auxiliary Registers Modifications in Bit-Reversed Addressing

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

Table 5–5 shows the relationship of the bit pattern of the index steps and the four LSBs of AR2, which contain the bit-reversed address.

See the *TMS320C54x DSP Reference Set, Volume 4: Applications Guide* for an application of the bit-reversed addressing mode.

Table 5–5. Bit-Reversed Addresses

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

5.5.4 Dual-Operand Address Modifications

Dual data-memory operand addressing is used for instructions that perform two reads or a single read and a parallel store (indicated by two vertical bars, ||) at the same time. These instructions are all one word long and operate in indirect addressing mode only. Two data-memory operands are represented by Xmem and Ymem:

- Xmem is a *read* operand with access through the D bus. Store instructions, for example STH and STL with shift operation, change Xmem to a *write* operand.
- Ymem is used as a *read* operand in instructions with dual reads (accessed through the C bus) or as a *write* operand in instructions with a parallel store (accessed through the E bus).

If the source operand and the destination operand point to the same location, in instructions with a parallel store (for example, ST||LD), the source is read before writing to the destination. If a dual-operand instruction (for example, ADD) points to the same auxiliary register with different addressing modes specified for both operands, the mode defined by the Xmod field is used for addressing.

Figure 5–11 shows the indirect-addressing instruction format for a dual data-memory operand. Table 5–6 describes the bits of the instruction.

Because only two bits are available for selecting each auxiliary register in this mode, only four of the auxiliary registers can be used, AR2 – AR5. Table 5–7 shows which Xar or Yar value selects which auxiliary registers.

Figure 5–11. Indirect-Addressing Instruction Format for Dual Data-Memory Operands

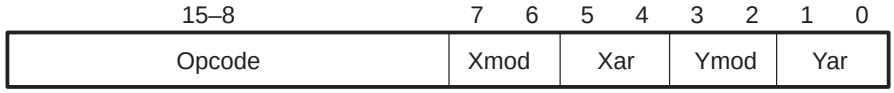


Table 5–6. Indirect-Addressing Instruction Bit Summary – Dual Data-Memory Operands

Bit	Name	Function
15 – 8	Opcode	This eight-bit field contains the operation code for the instruction.
7–6	Xmod	This 2-bit field defines the type of indirect addressing mode used for accessing the Xmem operand.
5–4	Xar	The 2-bit Xmem auxiliary register selection field defines the auxiliary register that contains the address of Xmem.
3–2	Ymod	This 2-bit field defines the type of indirect addressing mode used for accessing the Ymem operand.
1–0	Yar	The 2-bit Ymem auxiliary register selection field defines the auxiliary register that contains the address of Ymem.

Table 5–7. Auxiliary Registers Selected by Xar and Yar Field of Instruction

Xar or Yar Field	Auxiliary Register
00	AR2
01	AR3
10	AR4
11	AR5

Figure 5–12 shows how an address is generated using dual data-memory operand addressing.

Dual data-memory operand addressing uses four auxiliary registers (AR2–AR5). The ARAUs, together with these registers, provide the capability to access two operands in a single cycle.

Figure 5–12. Indirect Addressing Block Diagram for Dual Data-Memory Operands

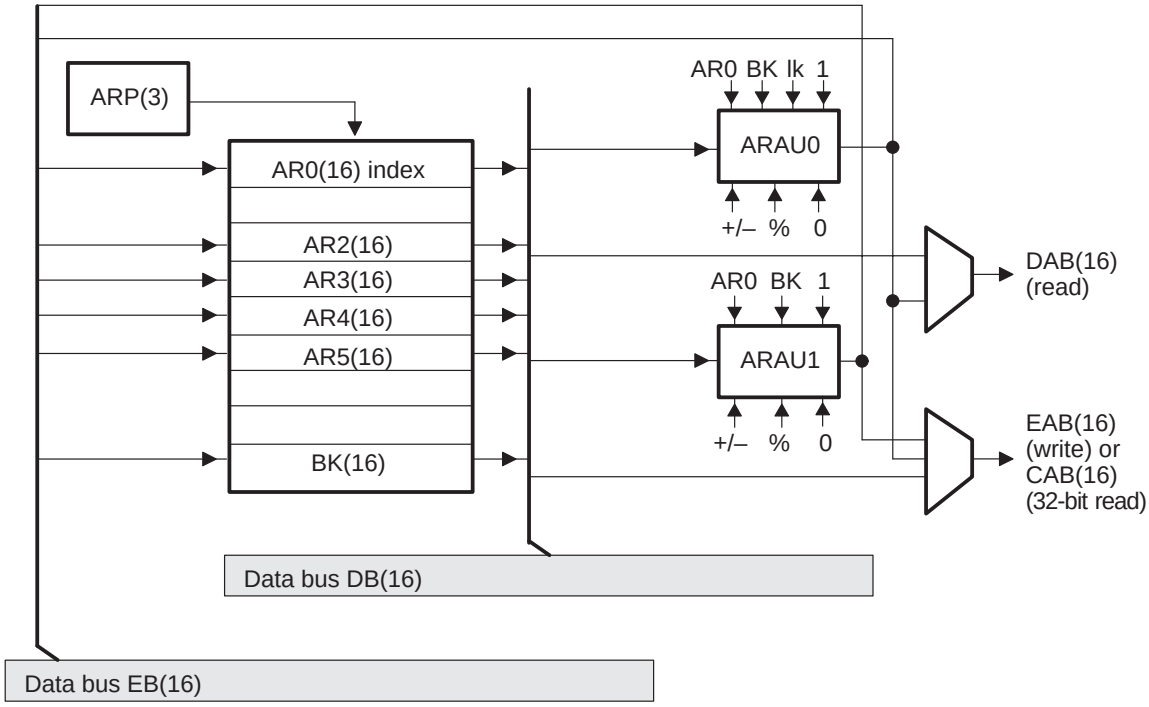


Table 5–8 lists the types of dual data-memory operand addressing, along with the value of the modification field (either Xmod or Ymod), the assembler syntax, and the function for each type.

Table 5–8. Indirect Addressing Types With Dual Data-Memory Operands

Xmod or Ymod Field	Operand Syntax	Function	Description†
00 (0)	*ARx	addr = ARx	ARx is the data-memory address.
01 (1)	*ARx–	addr = ARx ARx = ARx – 1	After access, the address in ARx is decremented.
10 (2)	*ARx+	addr = ARx ARx = ARx + 1	After access, the address in ARx is incremented.
11 (3)	*ARx+0%	addr = ARx ARx = circ(ARx + AR0)	After access, AR0 is added to ARx using circular addressing.‡

† ARx is used as the data-memory address unless otherwise specified.
‡ The size of the circular buffer is specified in circular-buffer size register (BK)

In each case, the content of the auxiliary register is used as the data-memory operand. After using the address in the auxiliary register, the ARAUs perform the specified mathematical operation. By disabling circular modifications, it is possible to perform indexed addressing or the equivalent of *ARx+0. Clearing the BK to 0 disables circular modification.

In instructions that perform dual-operand reads, if the auxiliary register specified by the Yar field accesses one of the memory-mapped registers, the value read will not represent the contents of the register.

See the *TMS320C54x DSP Reference Set, Volume 4: Applications Guide* for examples of dual-operand indirect addressing.

5.5.4.1 Dual-Operand Increment/Decrement Address Modifications (Xmod or Ymod = 0, 1, or 2)

You can modify the AR by incrementing or decrementing its value. When Xmod or Ymod = 0, ARx is used as the data-memory address with no incrementing or decrementing. When Xmod or Ymod = 1, ARx is decremented after the access is made. When Xmod or Ymod = 2, ARx is incremented after the access is made.

5.5.4.2 Dual-Operand Indexed Address Modifications (Xmod or Ymod = 3 and BK = 0)

When Xmod or Ymod = 3 and BK = 0, AR0 is added to ARx after each access. Otherwise, dual-operand indexed addressing is exactly as described in subsection 5.5.3.3 on page 5-15.

5.5.4.3 Dual-Operand Circular Address Modifications (Xmod or Ymod = 3 and BK $\neq$ 0)

When Xmod or Ymod = 3 and BK $\neq$ 0, AR0 is added to ARx using circular addressing after each access. Otherwise, dual-operand circular addressing is exactly as described in subsection 5.5.3.4 on page 5-15.

5.5.4.4 Single-Operand Instructions That Use the Dual-Operand Format

Some instructions with only one data-memory operand use dual data-memory operand addressing so that they fit in a single word for single-cycle execution. In these instructions, only Xmem is available and the Xmod and Xar fields define the addressing mode for the operand. Four single-operand instructions can be executed in a single cycle:

- BIT *Xmem*, *BITC*
- SACCD *src*, *Xmem*, *cond*
- SRCCD *Xmem*, *cond*
- STRCD *Xmem*, *cond*

Five instructions with optional shift also support this type of addressing for single-word, single-cycle execution:

- ADD *Xmem*, *SHFT*, *src*
- LD *Xmem*, *SHFT*, *dst*
- STH *src*, *SHFT*, *Xmem*
- STL *src*, *SHFT*, *Xmem*
- SUB *Xmem*, *SHFT*, *src*

5.5.5 TMS320C2x/C2xx/C5x Compatibility (ARP) Mode

ARP can be used in indirect addressing. This allows the AR to be defined by ARP to ease code translation from a ’C2x/C2xx/C5x device. With CMPT = 1 and ARF = 0, ARP is used to determine which AR is used to address memory. Figure 5–13 shows how the ARP indexes the auxiliary registers.

In using ARP, the ’54x differs from the ’C5x in that when the ’54x uses the AR pointed to by ARP, the ’54x does not update the ARP with the same instruction. Table 5–9 shows the assembler syntax for the ’C2x/C2xx/C5x compared to the ’54x.

Figure 5–13. How ARP Indexes the Auxiliary Registers

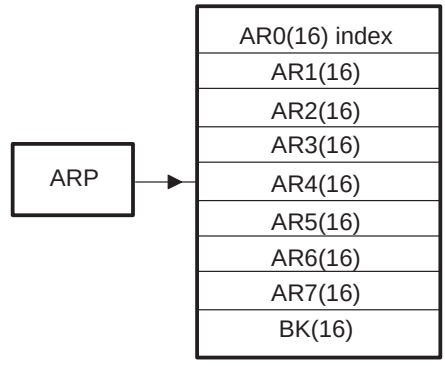


Table 5–9. Assembler Syntax Comparison for TMS320C2x/C2xx/C5x and ’54x

Syntax for ’C2x/C2xx/C5x	Syntax for ’54x	Syntax for ’C2x/C2xx/C5x	Syntax for ’54x
*	*AR0	*0–	*AR0–0
*–	*AR0–	*0+	*AR0+0
*+	*AR0+	*BR0–	*AR0–0B
		*BR0+	*AR0+0B

Figure 5–14 shows the indirect-addressing instruction format for the ARP mode. Table 5–10 describes the bits of the ARP-mode instruction.

Figure 5–14. Indirect-Addressing Instruction Format for Compatibility Mode

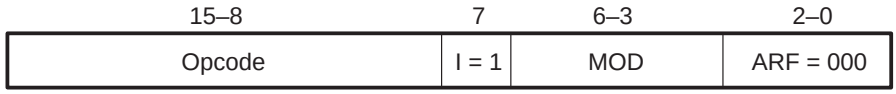


Table 5–10. Indirect-Addressing Instruction Bit Summary – Compatibility Mode

Bit	Name	Function
15 – 8	Opcode	This eight-bit field contains the operation code for the instruction.
7	I	I = 1, the addressing mode used by the instruction is the indirect addressing mode.
6–3	MOD	This 4-bit modification field defines the type of indirect addressing. Subsection 5.5.3, *Single-Operand Address Modifications*, on page 5-13, describes the 16 ways to specify addressing types with the MOD field.
2–0	ARF	This 3-bit auxiliary register field defines the auxiliary register used for addressing. ARF depends on the compatibility mode bit (CMPT) in status register ST1: CMPT = 0 Standard mode. In standard mode, ARF always specifies the auxiliary register, regardless of the value in ARP. ARP is not updated. ARP must always be set to 0 when the DSP is in this mode. CMPT = 1 Compatibility mode. In compatibility mode, ARP selects the auxiliary register if ARF = 0. Otherwise, ARF selects the auxiliary register and the ARF value is loaded into ARP when the access is completed. *AR0 in the assembly instruction indicates the auxiliary register selected by ARP in compatibility mode.

Note:

ARP must always be set to 0 when the DSP is in standard mode (CMPT = 0). At reset, both ARP and CMPT are set to 0 automatically.

5.6 Memory-Mapped Register Addressing

Memory-mapped register addressing is used to modify the memory-mapped registers without affecting either the current data-page pointer (DP) value or the current stack-pointer (SP) value. Because DP and SP do not need to be modified in this mode, the overhead for writing to a register is minimal. Memory-mapped register addressing works for both direct and indirect addressing.

Figure 5–15 shows how memory-mapped addresses are generated. Addresses are generated by:

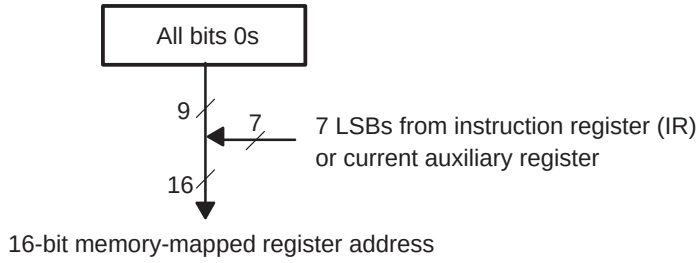
- ❑ Forcing the nine most significant bits (MSBs) of data-memory address to 0, regardless of the current value of DP or SP when direct addressing is used
- ❑ Using the seven LSBs of the current auxiliary register value when indirect addressing is used

Note:

In indirect addressing, the nine MSBs of the auxiliary register are forced to 0 after the operation.

For example, if AR1 is used to point to a memory-mapped register in memory-mapped register addressing mode and it contains a value of FF25h, then AR1 points to the timer period register (PRD), since the seven LSBs of AR1 are 25h and the address of the PRD is 0025h. After execution, the value remaining in AR1 is 0025h.

Figure 5–15. Memory-Mapped Register Addressing Block Diagram



Note:

In addition to registers, any scratch-pad RAM located on data page 0 can be modified by using memory-mapped register addressing.

Only eight instructions can use memory-mapped register addressing:

- LDM *MMR*, *dst*
- MVDM *dmad*, *MMR*
- MVMD *MMR*, *dmad*
- MVMM *MMRx*, *MMRy*
- POPM *MMR*
- PSHM *MMR*
- STLM *src*, *MMR*
- STM #*lk*, *MMR*

Note:

The following indirect addressing modes are not allowed for memory-mapped register addressing:

- *ARx(lk)
- *+ARx(lk)
- *+ARx(lk)%
- *(lk)

In these cases, the assembler issues a warning.

5.7 Stack Addressing

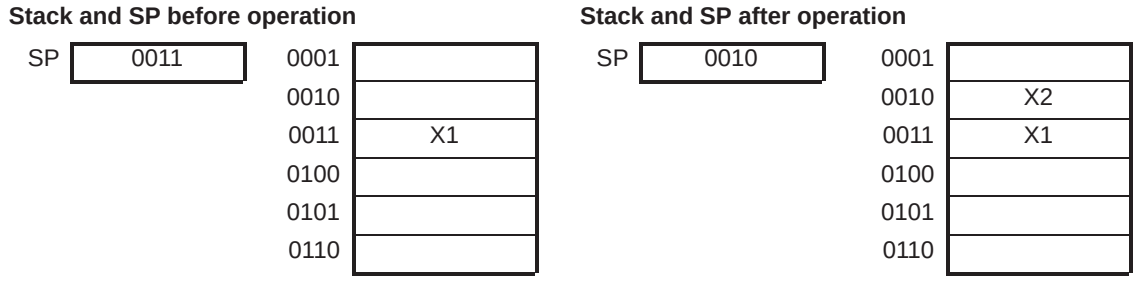
The system stack is used to automatically store the program counter during interrupts and subroutines. It can also be used at your discretion to store additional items of context or to pass data values. The stack is filled from the highest to the lowest memory address. The processor uses a 16-bit memory-mapped register, the stack pointer (SP), to address the stack. SP always points to the last element stored onto the stack.

Four instructions access the stack using the stack addressing mode:

- PSHD pushes a data-memory value onto the stack.
- PSHM pushes a memory-mapped register onto the stack.
- POPD pops a data-memory value from the stack.
- POPM pops a memory-mapped register from the stack.

A push predecrements and a pop postincrements the address in the SP. Figure 5–16 shows an example of the stack and SP before and after a push of X2 into the stack (PSHD X2).

Figure 5–16. Stack and Stack Pointer Before and After a Push Operation



Other operations also affect the stack and the stack pointer. The stack is used during interrupts and subroutines to save and restore the PC contents. When a subroutine is called or an interrupt occurs, the return address is automatically saved in the stack using a push operation. Instructions used for subroutine calls and interrupts are CALA[D], CALL[D], CC[D], INTR, and TRAP.

When a subroutine returns, the return address is retrieved from the stack using a pop operation and loaded into the PC. Instructions used for returns from subroutines are RET[D], RETE[D], RETEF[D], and RC[D].

The FRAME instruction also affects the stack. This instruction adds a short-immediate offset to the stack pointer. The stack is also used in SP-referenced direct addressing (see subsection 5.4.2, *SP-Referenced Direct Addressing*, on page 5-9).

5.8 Data Types

There are two basic data types for accessing memory in the ’54x: 16-bit and 32-bit. Most instructions can access 16-bit data. Accessing 32-bit data, however, requires the use of the special instructions listed in Table 5–11.

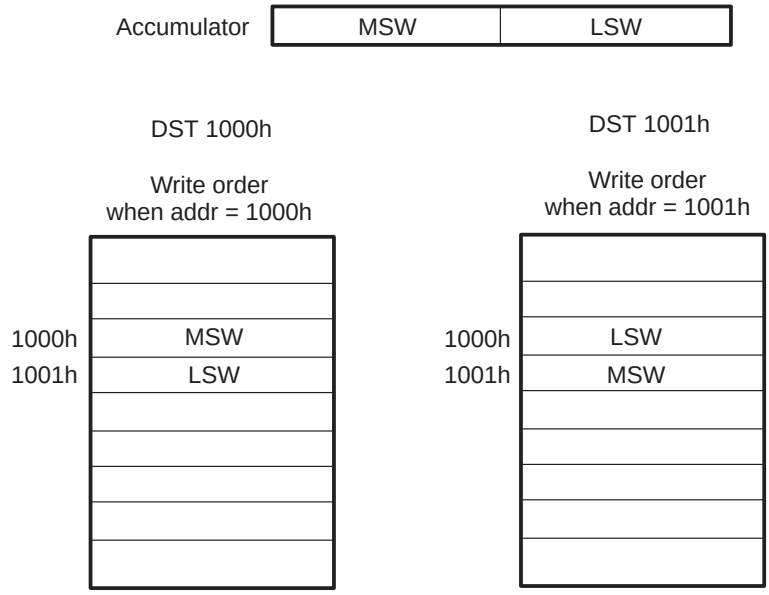
Table 5–11. Instructions With 32-Bit Word Operands

Instruction	Description
DADD	Double-precision add/dual 16-bit add to accumulator
DADST	Double-precision load with T add/dual 16-bit load with T add/subtract
DLD	Long-word load to accumulator
DRSUB	Double-precision subtract/dual 16-bit subtract from long word
DSADT	Long load with T subtract/dual 16-bit load with T subtract/add
DST	Store accumulator in long word
DSUB	Double-precision subtract/dual 16-bit subtract from accumulator
DSUBT	Long load with T subtract/dual 16-bit load with T subtract

For a 16-bit operand access, a 16-bit word is read from data memory through the D bus and written to data memory through the E bus. For a 32-bit operand access, both the C (for most-significant word) and the D (for least-significant word) buses are used for a read. However, because only the E bus is used for a write, the write operation (DST instruction) is executed in two cycles.

With 32-bit accesses, the first word accessed is treated as the most-significant word (MSW), while the second word accessed is the least-significant word (LSW). If the first word accessed is at an even address, then the second word is at the next (higher) address. If the first word accessed is at an odd address, then the second word is at the previous (lower) address. Figure 5–17 shows this effect.

Figure 5–17. Word Order in Memory



Chapter 6

Program Memory Addressing

This chapter discusses how program-memory addresses are generated and which addresses are loaded into the program counter (PC). This chapter also describes the program control operations that affect the value loaded in the PC:

- ❏ Branches
- ❏ Calls
- ❏ Returns
- ❏ Conditional operations
- ❏ Repeats of an instruction or a block of instructions
- ❏ Hardware reset
- ❏ Interrupts

These operations can cause a nonsequential address to be loaded into PC. Section 7.1, *Pipeline Operation*, and Section 7.2, *Interrupts and the Pipeline*, are helpful in understanding the operation of PC discontinuities. Power-down modes halt program execution.

Topic **Page**

6.1 Program-Memory Address Generation

Program memory contains code for applications, coefficient tables, and immediate operands. The ’54x can address a total of 64K words of program memory using the program address bus (PAB); the ’548 and ’549 have an additional seven address lines to provide external access to 128 64K-word pages. The program-address generation logic (PAGEN) generates the address used to access instructions, coefficient tables, 16-bit immediate operands, or other information stored in program memory and puts this address on the PAB.

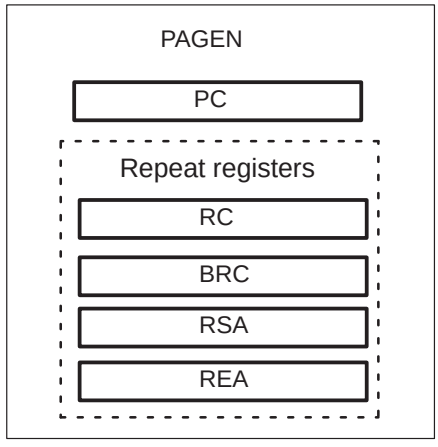
PAGEN consists of five registers (see Figure 6–1):

- Program counter (PC)
- Repeat counter (RC)
- Block-repeat counter (BRC)
- Block-repeat start address register (RSA)
- Block-repeat end address register (REA)

One additional register is used in the ’548 and ’549 to address extended memory:

- Program counter extension register (XPC)

Figure 6–1. Program-Address Generation Logic (PAGEN) Registers



’54x devices fetch instructions by putting the value of the PC on the PAB and reading the appropriate location in memory. While the memory location is read, PC is incremented for the next fetch. If a program address discontinuity occurs (for example, a branch, a call, a return, an interrupt, or a block repeat), the appropriate address is loaded into the PC. The instruction addressed through the PAB is then loaded into the instruction register (IR).

To improve the performance of certain instructions, the program address generation unit is also used to fetch operands from program memory. Operands are fetched from program memory when the device reads from or writes to a coefficient table or when it transfers data between program and data space. Some instructions, such as FIRS, MACD, and MACP, use the program bus to fetch a second multiplicand.

6.2 Program Counter (PC)

The PC is a 16-bit register that contains the internal or external program-memory address used when an instruction is fetched or when a 16-bit-immediate operand or coefficient table in program memory is accessed. To address program memory, the address in the PC is put onto the PAB.

The PC can be loaded several ways. Table 6–1 shows what is loaded into the PC according to the code operation performed.

Table 6–1. Loading Addresses Into PC

Code Operation	Address Loaded to the PC
Reset	PC is loaded with FF80h.
Sequential execution	PC is loaded with PC + 1.
Branch	PC is loaded with the 16-bit-immediate value directly following the branch instruction.
Branch from accumulator	PC is loaded with the lower 16-bit word of accumulator A or B.
Block repeat loop	PC is loaded with the repeat start address (RSA) when PC + 1 equals the repeat end address (REA) + 1, provided that BRAF = 1.
Subroutine call	PC + 2 is pushed onto the stack, and PC is loaded with the 16-bit-immediate value directly following the call instruction mnemonic. The return instruction pops the top of the stack back into PC to return to the calling sequence of code.
Subroutine call from accumulator	PC + 1 is pushed onto the stack, and PC is loaded with the lower 16-bit word of accumulator A or B. The return instruction pops the top of the stack back into PC to return to the calling sequence of code.
Hardware interrupt, software interrupt, or trap	PC is pushed onto the stack, and PC is loaded with the address of the appropriate trap vector. The return instruction pops the top of the stack back into PC to return to the interrupting sequence of code.

XPC is a 7-bit register that selects the current page of program memory for the ’548 and ’549. For more information about extended program memory in the ’548 and ’549, see section 3.1.1, *Extended Program Memory*, on page 3-8.

XPC can be loaded in several ways in conjunction with the loading of the PC. Table 6–2 lists operations that load XPC.

Table 6–2. Loading Addresses into XPC

Code Operation	Address Loaded to the PC
Reset	PC is loaded with FF80h. XPC is loaded with 0h.
Sequential execution	PC is loaded with PC + 1. XPC is not automatically incremented.
Far branch	PC is loaded with bits 15–0 of the immediate value directly following the branch instruction. XPC is loaded bits 23–16 of that value.
Far branch from accumulator	PC is loaded with bits 15–0 of accumulator A or B. XPC is loaded with bits 23–16 of accumulator A or B.
Far subroutine call	XPC is pushed onto the stack, PC + 2 is pushed onto the stack, PC and XPC are loaded with bits 15–0 and bits 23–16, respectively, of the immediate value specified by the call instruction.
Far subroutine call from accumulator	XPC is pushed onto the stack, PC + 1 is pushed onto the stack, and the PC and XPC are loaded with bits 15–0 and bits 23–16, respectively, of accumulator A or B.
Far return	The return instruction pops the top of the stack into PC and pops the next value into the XPC to return to the calling sequence of code.

Note:

XPC is not loaded by instructions other than those listed in Table 6–2.

6.3 Branches

Branches break the sequential flow of instructions by transferring control to another location in program memory. Therefore, branches affect the program address generated and stored in PC. The ’54x performs both unconditional and conditional branches, and both of these types can be either nondelayed or delayed.

6.3.1 Unconditional Branches

An unconditional branch is always executed when it is encountered. During the execution, PC is loaded with the specified branch-to-program-memory address and execution of the new section of code begins at that address. The address loaded into PC comes from either the second word of the branch instruction or the lower 16 bits of an accumulator (accumulator A or accumulator B).

By the time the branch instruction reaches the execute phase of the pipeline, the next two instruction words have already been fetched. How these two instruction words are handled depends in part on whether the branch is nondelayed or delayed:

- Nondelayed: The two instruction words are flushed from the pipeline so that they are not executed, and then execution continues at the branched-to address.
- Delayed: The one 2-word instruction or two 1-word instructions following the branch instruction are executed. This allows you to avoid flushing the pipeline, which requires extra cycles.

Note:

The two words following a delayed instruction cannot be an instruction that causes a PC discontinuity (a branch, call, return, or software interrupt).

Table 6–3 shows the unconditional branch instructions in the ’54x and the number of cycles needed to execute these instructions (both nondelayed and delayed). Delayed instructions use two cycles fewer than the corresponding nondelayed instructions because they do not flush the pipeline.

Table 6–3. Unconditional Branch Instructions

Instruction	Description	Number of Cycles (Nondelayed / Delayed)
B[D]	Load PC with the address specified by the instruction	4/2
BACC[D]	Load PC with the address specified by the low 16 bits of the designated accumulator	6/4

6.3.2 Conditional Branches

Conditional branches operate like unconditional branches, but they execute only when one or more user-specified conditions are met. The possible conditions are given in Table 6–12 on page 6-16. If all the conditions are met, PC is loaded with the second word of the branch instruction, which contains the address to branch to, and execution continues at this address.

By the time the conditions have been tested, the two instruction words following the conditional branch instruction have already been fetched and are in the pipeline. How these two instruction words are handled depends in part on whether the branch is nondelayed or delayed:

- Nondelayed: If all the conditions are met, these two instruction words are flushed from the pipeline so that they are not executed, and then execution continues at the branched-to address. If the conditions are *not* met, the two instruction words are executed instead of the branch.
- Delayed: The one 2-word instruction or two 1-word instructions following the branch instruction are executed. This allows you to avoid flushing the pipeline, which requires extra cycles. The conditions tested are not affected by the instructions following the delayed branch.

Note:

The two words following a delayed instruction cannot be an instruction that causes a PC discontinuity (a branch, call, return, or software interrupt).

Table 6–4 shows the conditional branch instructions and the number of cycles needed to execute these instructions. Because conditional branches use conditions determined by the execution of the previous instructions, the conditional branch instruction, BC[D], requires one more cycle than an unconditional one.

Table 6–4. Conditional Branch Instructions

Instruction	Description	Number of Cycles (Condition met / Not met)	
		Nondelayed	Delayed
BC[D]	Load PC with the address specified by the instruction if the condition specified by the instruction is met	5/3	3/3
BANZ[D]	Load PC with the address specified by the instruction if currently selected auxiliary register not equal to 0 (useful for loops)	4/2	2/2

6.3.3 Far Branches (Available on TMS320C548/549)

To allow branches to extended memory, the ’548 and ’549 include two far branch instructions:

- FB[D] branches to the 23-bit address specified by the the instruction.
- FBACC[D] branches to the 23-bit address specified in the designated accumulator.

Table 6–5 shows the far branch instructions in the ’548 and ’549 (both nondelayed and delayed) and the number of cycles needed to execute these instructions. Delayed instructions use two cycles fewer than the corresponding nondelayed instructions.

Table 6–5. Far Branch Instructions

Instruction	Description	Number of Cycles (Nondelayed / Delayed)
FB[D]	Load the PC and the XPC with the address specified in the instruction	4/2
FBACC[D]	Load the PC and the XPC with the address specified by the lower 23 bits of the designated accumulator	6/4

6.4 Calls

Like branches, calls break the sequential flow of instructions by transferring control to some other location in program memory. However, unlike branches, this transfer is intended to be temporary. When a subroutine or function is called, the address of the next instruction following the call is saved in the stack. This address is used to return to the calling program and resume execution. The ’54x performs both unconditional and conditional calls, and both of these types can be either nondelayed or delayed.

6.4.1 Unconditional Calls

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

By the time the unconditional call instruction reaches the execute phase of the pipeline, the next two instruction words have already been fetched. How these two instruction words are handled depends in part on whether the call is nondelayed or delayed:

- ❏ Nondelayed: The two instruction words are flushed from the pipeline so that they are not executed, the return address is stored to the stack, and then execution continues at the beginning of the called function.
- ❏ Delayed: The one 2-word instruction or two 1-word instructions following the call instruction are executed. This allows you to avoid flushing the pipeline, which requires extra cycles.

Note:

The two words following a delayed instruction cannot be an instruction that causes a PC discontinuity (a branch, call, return, or software interrupt).

Table 6–6 shows the unconditional call instructions in the ’54x (both nondelayed and delayed) and the number of cycles needed to execute these instructions. Delayed instructions need two cycles fewer than the corresponding nondelayed instructions because they do not flush the pipeline.

Table 6–6. Unconditional Call Instructions

Instruction	Description	Number of Cycles (Nondelayed / Delayed)
CALL[D]	Places the return address on the stack and then loads the PC with the address specified by the instruction	4/2
CALA[D]	Places the return address on the stack and then loads the PC with the address specified in the designated accumulator	6/4

6.4.2 Conditional Calls

Conditional calls operate like unconditional calls, but they execute only when one or multiple conditions are met. The possible conditions are given in Table 6–12 on page 6-16. If all the conditions are met, the PC is loaded with the second word of the call instruction, which contains the starting address of the function to be called. Before branching to the called function, the processor stores the address of the instruction following the call instruction to the stack. The function must end with a return instruction, which takes the address off the stack and loads PC, allowing the processor to resume execution of the calling program.

By the time the conditions of the conditional call instruction have been tested, the two instruction words following the call instruction have already been fetched in the pipeline. How these two instruction words are handled depends in part on whether the call is nondelayed or delayed:

- Nondelayed: If all the conditions are met, these two instruction words are flushed from the pipeline so that they are not executed, and then execution continues at the beginning of the called function. If the conditions are *not* met, the two instructions are executed instead of the call.
- Delayed: The one 2-word instruction or two 1-word instructions following the call instruction are always executed. This allows you to avoid flushing the pipeline, which requires extra cycles. The conditions tested are not affected by the instructions following the delayed call. If the conditions are *not* met, the processor executes the two instruction words instead of the call.

Note:

The two words following a delayed instruction cannot be an instruction that causes a PC discontinuity (a branch, call, return, or software interrupt).

Table 6–7 shows the conditional call instruction and the number of cycles needed to execute this instruction. Because there is a wait cycle for conditions to become stable, the conditional call instruction, CC[D], requires one more cycle than the unconditional one.

Table 6–7. Conditional Call Instruction

		Number of Cycles (Condition met / Not met)	
Instruction	**Description**	**Nondelayed**	**Delayed**
CC[D]	Places the return address on the stack and then loads the PC with the address specified by the instruction if the condition specified by the instruction is met	5/3	3/3

6.4.3 Far Calls (Available on TMS320C548 /549)

To allow calls to extended memory, the ’548 and ’549 include two far call instructions:

- The FCALL instruction pushes XPC onto the stack, pushes PC onto the stack, and branches to the 23-bit address specified by the the instruction.
- The FCALA pushes XPC onto the stack, pushes PC onto the stack, and branches to the 23-bit address specified in the designated accumulator.

Table 6–8 shows the far call instructions in the ’548 and ’549 (nondelayed and delayed) and the number of cycles needed to execute these instructions. Note that delayed instructions need two cycles fewer than the corresponding nondelayed instructions.

Table 6–8. Far Call Instructions

Instruction	**Description**	**Number of Cycles (Nondelayed / Delayed)**
FCALL[D]	Places XPC and PC on the stack and then loads XPC and PC with the address specified by the instruction	4/2
FCALA[D]	Places XPC and PC on the stack and then loads XPC and PC with the address specified in the designated accumulator	6/4

6.5 Returns

Return instructions provide a way to resume processing of a sequence of instructions that was broken by a call to another function or an interrupt service routine. When the called function or interrupt service routine has completed its execution, it is necessary to resume processing at the point immediately following the call or the point at which the interrupt occurred. Return instructions accomplish this by popping the top value of the stack, which contains the address of the next instruction to be executed, into the program counter (PC). The ’54x performs both unconditional and conditional returns, and both of these types can be either nondelayed or delayed.

The ’548 and ’549 offer an additional return instruction: an unconditional far return, both nondelayed and delayed.

6.5.1 Unconditional Returns

An unconditional return is always executed when it is encountered. When the return is executed, PC is loaded with the return address from the stack and execution resumes at the instruction following the instruction that called the function or at the point at which the interrupt occurred.

By the time the unconditional return instruction reaches the execute phase of the pipeline, the next two instruction words have already been fetched. How these two instruction words are handled depends in part on whether the return is nondelayed or delayed.

- ❏ Nondelayed: The two instruction words are flushed from the pipeline so that they are not executed, the return address is taken from the stack or from the RTN register, and then execution continues at that address in the calling function.
- ❏ Delayed: The one 2-word instruction or two 1-word instructions following the return instruction are executed. This lets you avoid flushing the pipeline, which requires extra cycles. The return address is taken from the stack or from the RTN register.

Note:

The two words following a delayed instruction cannot be an instruction that causes a PC discontinuity (a branch, call, return, or software interrupt).

Table 6–9 shows the unconditional return instructions in the ’54x (nondelayed and delayed) and the number of cycles needed to execute these instructions. Delayed instructions need two cycles fewer than the corresponding nondelayed instructions.

Table 6–9. Unconditional Return Instructions

Instruction	Description	Number of Cycles (Nondelayed / Delayed)
RET[D]	Load the PC with the return address at the top of the stack	5/3
RETE[D]	Load the PC with the return address at the top of the stack, and enable maskable interrupts	5/3
RETF[D]	Load the PC with the return address in the RTN register, and enable maskable interrupts	3/1

Enabling interrupts with the RETE and RETF instructions ensures that the return executes before another interrupt is processed. By using the RETF instruction, loading the PC from the RTN register rather than the stack allows a quicker return. This reduces the total number of cycles used by an interrupt routine, which is particularly important for short, frequently used interrupt routines.

Note:

The RTN register is a CPU-internal register that you cannot read from or write to.

6.5.2 Conditional Returns

By using the conditional return (RC) instruction, you can give a function or interrupt service routine (ISR) more than one possible return path. The path chosen depends on the data being processed. In addition, you can use a conditional return to avoid conditionally branching to/around the return instruction at the end of the function or ISR.

Conditional returns operate like unconditional returns, but they execute only when one or more conditions are met. The possible conditions are given in Table 6–12 on page 6-16. If all the conditions are met, the processor loads the return address from the stack to PC, and resumes execution of the calling program.

The conditional return is a single-word instruction; however, because of the potential PC discontinuity, it operates with the same effective execution time as the conditional branch or call.

By the time the conditions of the conditional return instruction have been tested, the two instruction words following the return instruction have already been fetched in the pipeline. How these two instruction words are handled depends in part on whether the return is nondelayed or delayed:

- Nondelayed: If all the conditions are met, these two instruction words are flushed from the pipeline so that they are not executed, and then execution of the calling program continues. If the conditions are *not* met, the two instructions are executed instead of the return.
- Delayed: The processor executes the two instructions that follow the return instruction. This allows you to avoid flushing the pipeline, which requires extra cycles. The conditions tested are not affected by the instructions following the delayed return.

Note:

The two words following a delayed instruction cannot be an instruction that causes a PC discontinuity (a branch, call, return, or software interrupt).

Table 6–10 shows the conditional return instruction and the number of cycles needed to execute this instruction.

Table 6–10. Conditional Return Instruction

		Number of Cycles (Condition met / Not met)	
Instruction	**Description**	**Nondelayed**	**Delayed**
RC[D]	Load PC with the return address at the top of the stack if the condition specified by the instruction is met	5/3	3/3

6.5.3 Far Returns (Available on TMS320C548/549)

To allow returns from extended memory, the ’548 and ’549 include two far-return instructions:

- FRET loads XPC from the stack and then loads PC from the stack, allowing program execution to resume at the previous point.
- FRETE loads XPC from the stack, loads PC from the stack, and enables maskable interrupts.

Table 6–11 shows the far return instructions in the ’548 and ’549 (nondelayed and delayed) and the number of cycles needed to execute these instructions. Note that delayed instructions need two cycles fewer than the corresponding nondelayed instructions.

Table 6–11. Far Return Instructions

Instruction	Description	Number of Cycles (Nondelayed / Delayed)
FRET[D]	Loads XPC with the value at the top of the stack and loads PC with the next value on the stack	6/4
FRETE[D]	Loads XPC with the value at the top of the stack, loads PC with the next value on the stack, and enables maskable interrupts	6/4

6.6 Conditional Operations

The ’54x includes instructions that execute only if one or more conditions are met. Table 6–12 lists the conditions that you can use with these instructions and their corresponding operand symbols.

Table 6–12. Conditions for Conditional Instructions

Condition	Description	Operand
A = 0	Accumulator A equal to 0	AEQ
B = 0	Accumulator B equal to 0	BEQ
A ≠ 0	Accumulator A not equal to 0	ANEQ
B ≠ 0	Accumulator B not equal to 0	BNEQ
A < 0	Accumulator A less than 0	ALT
B < 0	Accumulator B less than 0	BLT
A ≤ 0	Accumulator A less than or equal to 0	ALEQ
B ≤ 0	Accumulator B less than or equal to 0	BLEQ
A > 0	Accumulator A greater than 0	AGT
B > 0	Accumulator B greater than 0	BGT
A ≥ 0	Accumulator A greater than or equal to 0	AGEQ
B ≥ 0	Accumulator B greater than or equal to 0	BGEQ
AOV = 1	Accumulator A overflow detected	AOV
BOV = 1	Accumulator B overflow detected	BOV
AOV = 0	No accumulator A overflow detected	ANOV
BOV = 0	No accumulator B overflow detected	BNOV
C = 1	ALU carry set to 1	C
C = 0	ALU carry cleared to 0	NC
TC = 1	Test/control flag set to 1	TC
TC = 0	Test/control flag cleared to 0	NTC
$\overline{\text{BIO}}$ low	$\overline{\text{BIO}}$ signal is low	BIO
$\overline{\text{BIO}}$ high	$\overline{\text{BIO}}$ signal is high	NBIO
none	Unconditional operation	UNC

6.6.1 Using Multiple Conditions

Multiple conditions can be listed as operands of the conditional instructions. If multiple conditions are listed, all conditions must be met for the instruction to execute. Only certain combinations of conditions are acceptable (see Table 6–13). For each combination, the conditions must be selected from Group 1 or Group 2 as follows:

- Group 1: You can select one condition from category A and one condition from category B. The two conditions cannot be from the same category. For example, you can test EQ and OV at the same time but you cannot test GT and NEQ at the same time. The accumulator must be the same for both conditions; you cannot test conditions for both accumulators with the same instruction. For example, you can test AGT and AOV at the same time, but you can not test AGT and BOV at the same time.
- Group 2: You can select one condition from each of three categories (A, B, and C). No two conditions can be from the same category. For example, you can test TC, C, and BIO at the same time, but you cannot test NTC, C, and NC at the same time.

Table 6–13. Grouping of Conditions for Multiconditional Instructions

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

6.6.2 Conditional Execute (XC) Instruction

Where code branches conditionally over a 1- or 2-word code segment, you can replace the branch with a 1-cycle conditional execute instruction (XC). There are two forms for the XC instruction. One form is a conditional execute of a 1-word instruction (XC 1, cond). The second form is a conditional execute of one 2-word instruction or two 1-word instructions (XC 2, cond). Conditions for XC are the same as the conditions for conditional branches, calls, and returns (see Table 6–12).

The condition must be stable two full cycles before the XC instruction is executed. This ensures that the decision is made before the instruction following XC is decoded. Avoid changing the XC condition in the two 1-word instructions prior to XC. If no interrupts occur, these instructions have no effect on XC. However, if an interrupt occurs, it can trap between the instructions and XC, affecting the condition before XC is executed. See Chapter 7, *Pipeline*, for information about pipeline latencies.

6.6.3 Conditional Store Instructions

Some CPU registers can be conditionally stored in data memory using the conditional store instructions, listed in Table 6–14. The conditions used with conditional store instructions are listed in Table 6–15.

In a conditional store instruction, the address is modified and the memory operand is read regardless of the condition. If the condition is met, the corresponding register is stored in data memory. If the condition is not met, the operand is written into the same memory location from which it was read, so, the value of that memory location remains the same.

The conditional store instructions are single memory-operand instructions, but they use the dual memory-operand indirect addressing mode to put the instruction into one 16-bit word. Therefore, these instructions execute in one cycle.

Conditionally storing the block-repeat counter (BRC) allows you to store an index in a repeat-block loop.

Table 6–14. Conditional Store Instructions

Instruction	CPU Register
SACCD	Accumulator A or B
STRCD	Temporary register (T)
SRCCD	Block-repeat counter (BRC)

Table 6–15. Conditions for Conditional Store Instructions

Operand	Condition	Description
AEQ	A = 0	Accumulator A equal to 0
BEQ	B = 0	Accumulator B equal to 0
ANEQ	A ≠ 0	Accumulator A not equal to 0
BNEQ	B ≠ 0	Accumulator B not equal to 0
ALT	A < 0	Accumulator A less than 0
BLT	B < 0	Accumulator B less than 0
ALEQ	A ≤ 0	Accumulator A less than or equal to 0
BLEQ	B ≤ 0	Accumulator B less than or equal to 0
AGT	A > 0	Accumulator A greater than 0
BGT	B > 0	Accumulator B greater than 0
AGEQ	A ≥ 0	Accumulator A greater than or equal to 0
BGEQ	B ≥ 0	Accumulator B greater than or equal to 0

6.7 Repeating a Single Instruction

The ’54x includes two instructions, RPT and RPTZ, that cause the next instruction to be repeated. The number of times for the instruction to be repeated is obtained from an operand of the instruction, and is equal to this operand + 1. This value is stored in the 16-bit repeat counter (RC) register. You cannot program the value in the RC register; it is loaded by the repeat instructions (RPT or RPTZ) only. The maximum number of executions of a given instruction is 65 536. An absolute program or data address is automatically incremented when the single-repeat feature is used.

Once a repeat instruction is decoded, all interrupts, including $\overline{\text{NMI}}$ but not $\overline{\text{RS}}$, are disabled until the completion of the repeat loop. However, the ’54x does respond to the $\overline{\text{HOLD}}$ signal while executing an RPT/RPTZ loop—the response depends on the value of the HM bit of ST1.

The repeat function can be used with some instructions, such as multiply/accumulate and block moves, to increase the execution speed of these instructions. These multicycle instructions (see Table 6–16) effectively become single-cycle instructions after the first iteration of a repeat instruction.

Table 6–16. Multicycle Instructions That Become Single-Cycle Instructions When Repeated

Instruction	Description	# Cycles†
FIRS	Symmetrical FIR filter	3
MACD	Multiply and move result in accumulator with delay	3
MACP	Multiply and move result in accumulator	3
MVDK	Data-to-data move	2
MVDM	Data-to-MMR move	2
MVDP	Data-to-program move	4
MVKD	Data-to-data move	2
MVMD	MMR-to-data move	2
MVPD	Program-to-data move	3
READA	Program-to-data move	5
WRITA	Data-to-program move	5

† Number of cycles when instruction is not repeated

Single data-memory operand instructions cannot be repeated if a long offset modifier or an absolute address is used (for example, *ARn(lk), *+ARn(lk), *+ARn(lk)% and *(lk)). Instructions listed in Table 6–17 cannot be repeated using RPT.

Table 6–17. Nonrepeatable Instructions

Instruction	Description
ADDM	Add long constant to data memory
ANDM	AND data memory with long constant
B[D]	Unconditional branch
BACC[D]	Branch to accumulator address
BANZ[D]	Branch on auxiliary register not 0
BC[D]	Conditional branch
CALA[D]	Call to accumulator address
CALL[D]	Unconditional call
CC[D]	Conditional call
CMPR	Compare with auxiliary register
DST	Long word (32-bit) store
FB[D]	Far branch unconditionally
FBACC[D]	Far branch to location specified by accumulator
FCALA[D]	Far call subroutine at location specified by accumulator
FCALL[D]	Far call unconditionally
FRET[D]	Far return
FRETE[D]	Enable interrupts and far return from interrupt
IDLE	Idle instructions
INTR	Interrupt trap
LD ARP	Load auxiliary register pointer (ARP)
LD DP	Load data page pointer (DP)
MVMM	Move memory-mapped register (MMR) to another MMR
ORM	OR data memory with long constant
RC[D]	Conditional return
RESET	Software reset
RET[D]	Unconditional return

Table 6–17. Nonrepeatable Instructions (Continued)

Instruction	Description
RETE[D]	Return from interrupt
RETF[D]	Fast return from interrupt
RND	Round accumulator
RPT	Repeat next instruction
RPTB[D]	Block repeat
RPTZ	Repeat next instruction and clear accumulator
RSBX	Reset status register bit
SSBX	Set status register bit
TRAP	Software trap
XC	Conditional execute
XORM	XOR data memory with long constant

6.8 Repeating a Block of Instructions

The repeat-block instructions are used to repeat a block of code N + 1 times, where N is the value loaded into the block-repeat counter register (BRC). This block of code can contain one or more instructions. Unlike the repeat single operation, which disables all maskable interrupts, the repeat block operation can be interrupted.

The instructions used for this operation are RPTB and RPTBD (a delayed instruction). The RPTB instruction executes in four cycles. RPTBD allows the execution of one 2-word instruction or two 1-word instructions following the RPTBD instruction instead of flushing the pipeline; thus, RPTBD effectively executes in 2 cycles. When the RPTBD instruction is used, delayed instructions cannot be in the two words following the RPTBD instruction.

The repeat block feature provides zero-overhead looping. Zero-overhead looping is controlled by the block-repeat active flag (BRAF) in ST1 and the following memory-mapped registers:

- ❏ BRC contains the value N, which is one less than the number of times the block is to be repeated.
- ❏ The block-repeat start address register (RSA) holds the address of the first instruction of the block of code to be repeated.
- ❏ The block-repeat end address register (REA) holds the address of the last instruction word of the block of code to be repeated.

BRAF is set to 1 to activate the block repeat. The block repeat feature can be activated only if the number of iterations is greater than 0. The following steps start a loop:

Step 1: You load BRC with a loop count in the 0 through 65 535 range.

Step 2: The instruction loads the address of the first instruction to be repeated. This instruction is the one immediately following RPTB of the second instruction following RPTBD. The repeat-block (RPTB) or repeat block with delay (RPTBD) instruction automatically loads RSA with the address of the instruction following the RPTB instruction, or with the address of the second instruction following the RPTBD instruction.

Step 3: The instruction loads REA with the address following the last word of the last instruction to be repeated in the block, which is also the long-immediate operand given in the instruction. This action also sets BRAF. REA is loaded with the 16-bit-immediate operand of the RPTB or RPTBD instruction, and the BRAF bit is set. The value for the 16-bit-immediate operand of RPTB or RPTBD is L – 1, where L is the address of the instruction following the last word of the last instruction in the loop.

Every time the PC is updated during loop execution, REA is compared to the PC value. If the values are equal, BRC is decremented. If BRC is greater than or equal to 0, RSA is loaded into the PC to restart the loop. If not, BRAF is reset to 0 and the processor resumes execution past the end of the loop.

BRC is decremented during the instruction decode phase of the last repeat block instruction. For this reason, be careful when using the SRCCD instruction within a loop. To save the current loop counter value (the predecremented BRC), the SRCCD instruction must be placed a minimum of three instructions before the end of the loop.

There is only one set of block repeat registers, so multiple block repeats cannot be nested without saving the context of the outside loops. The simplest way of establishing nested loops is to use the RPTB[D] instruction for the innermost loop only, and use the BANZ[D] for all outer loops.

6.9 Reset Operation

Reset ($\overline{RS}$) is a nonmaskable external interrupt that can be used at any time to place the '54x into a known state. For correct system operation after power-up, $\overline{RS}$ must be asserted (low) for several clock cycles to ensure that the data, address, and control lines are configured properly. Approximately five clock cycles after $\overline{RS}$ is deasserted (goes high), the processor fetches the instruction at FF80h and begins executing code. See Section 10.5, *Start-Up Access Sequences,* on page10-23, for the reset sequence.

The following actions occur during a reset operation:

- ❑ IPTR is set to 1FFh.
- ❑ $\overline{RS}$ is deasserted.
- ❑ The MP/$\overline{MC}$ bit in PMST is set to the value of the MP/$\overline{MC}$ pin.
- ❑ PC is set to FF80h.
- ❑ XPC is cleared (’548 and ’549).
- ❑ FF80h is driven on the address bus, regardless of the state of MP/$\overline{MC}$.
- ❑ The data bus goes into the high-impedance state.
- ❑ The control lines are made inactive.
- ❑ The $\overline{IACK}$ signal is generated.
- ❑ INTM is set to 1 to disable all maskable interrupts.
- ❑ IFR is cleared to clear the interrupt flags.
- ❑ The single repeat counter (RC) is cleared.
- ❑ A synchronized reset ($\overline{SRESET}$) signal is sent to initialize the peripherals.
- ❑ The following status bits are set to their initial values:

■ ARP = 0	■ CLKOFF = 0	■ HM = 0	■ SXM = 1
■ ASM = 0	■ CMPT = 0	■ INTM = 1	■ TC = 1
■ AVIS = 0	■ CPL = 0	■ OVA = 0	■ XF = 1
■ BRAF = 0	■ DP = 0	■ OVB = 0	
■ C = 1	■ DROM = 0	■ OVLY = 0	
■ C16 = 0	■ FRCT = 0	■ OVM = 0	

Notes:

1) The remaining status bits are not initialized—your code must initialize them appropriately.

2) Reset does not initialize the stack pointer (SP). Your code must initialize it.

3) If MP/$\overline{MC}$ = 0, the device begins executing code from the on-chip ROM. Otherwise, it begins executing code from off-chip memory.

6.10 Interrupts

Interrupts are hardware- or software-driven signals that cause the ’54x to suspend its main program and execute another function called an interrupt service routine (ISR). Typically, interrupts are generated by hardware devices that need to give data to or take data from the ’54x (for example, ADCs, DACs, and other processors). Interrupts can also be used to signal that a particular event has taken place (for example, the timer is finished counting).

The ’54x supports both software and hardware interrupts:

- A *software interrupt* is requested by a program instruction (INTR, TRAP, or RESET).
- A *hardware interrupt* is requested by a signal from a physical device. Two types exist:
  - External hardware interrupts are triggered by signals at external interrupt ports.
  - Internal hardware interrupts are triggered by signals from the on-chip peripherals.

When multiple hardware interrupts are triggered at the same time, the ’54x services them according to a set priority ranking in which 1 has the highest priority. To determine the priorities for the hardware interrupts, refer to the table for your particular ’54x device in subsection 6.10.10, *Interrupt Tables*, on page 6-37.

Each of the ’54x interrupts, whether hardware or software, can be placed in one of the following two categories:

- Maskable interrupts. These are hardware or software interrupts that can be blocked (masked) or enabled (unmasked) using software. The ’54x supports up to 16 user-maskable interrupts (SINT15–SINT0). Each device uses a subset of these 16 interrupts. For example, the ’541 uses only nine of these interrupts (the others are tied high internally). Some of these have two names because they can be initiated by software or hardware; for the ’541, the hardware names for these interrupts are:
  - $\overline{\text{INT3}}$ through $\overline{\text{INT0}}$
  - RINT0, XINT0, RINT1, and XINT1 (serial port interrupts)
  - TINT (timer interrupt)
- Nonmaskable interrupts. These interrupts cannot be blocked. The ’54x always acknowledges this type of interrupt and branches from the main program to an ISR. The ’54x nonmaskable interrupts include all software interrupts and two external hardware interrupts: $\overline{\text{RS}}$ (reset) and $\overline{\text{NMI}}$. ($\overline{\text{RS}}$ and $\overline{\text{NMI}}$ can also be asserted using software.)

$\overline{RS}$ is a nonmaskable interrupt that affects all ’54x operating modes. See Section 6.9, *Reset Operation*, on page 6-25. $\overline{NMI}$ is a nonmaskable interrupt. Interrupts are globally disabled when $\overline{NMI}$ is asserted. $\overline{NMI}$ is different from $\overline{RS}$ because it does not affect any of the ’54x modes.

The ’54x handles interrupts in three phases:

1) Receive interrupt request. Suspension of the main program is requested via software (program code) or hardware (a pin or an on-chip peripheral). If the interrupt source is requesting a maskable interrupt, the corresponding bit in the interrupt flag register (IFR) is set when the interrupt is received.

2) Acknowledge interrupt. The ’54x must acknowledge the interrupt request. If the interrupt is maskable, predetermined conditions must be met in order for the ’54x to acknowledge it. For nonmaskable hardware interrupts and for software interrupts, acknowledgment is immediate.

3) Execute interrupt service routine (ISR). Once the interrupt is acknowledged, the ’54x executes the branch instruction you place at a predetermined address (the vector location) and performs the ISR.

6.10.1 Interrupt Flag Register (IFR)

IFR is a memory-mapped CPU register that identifies and clears active interrupts (see Figure 6–2). An interrupt sets its corresponding interrupt flag in IFR until it is recognized by the CPU. Any of the following four events clear an interrupt flag:

- The ’54x is reset ($\overline{RS}$ is low).
- An interrupt trap is taken.
- A 1 is written to the appropriate bit in IFR.
- The INTR instruction is executed using the appropriate interrupt number.

A 1 in any IFR bit indicates a pending interrupt. To clear an interrupt, write a 1 to the interrupt’s corresponding bit in the IFR. All pending interrupts can be cleared by writing the current contents of the IFR back into the IFR.

Figure 6–2. Interrupt Flag Register (IFR) Diagram

(a) '541 IFR

15–12	11	10	9	8	7	6	5	4	3	2	1	0
Resvd	Resvd	Resvd	Resvd	INT3	XINT1	RINT1	XINT0	RINT0	TINT	INT2	INT1	INT0

(b) '542 IFR

15–12	11	10	9	8	7	6	5	4	3	2	1	0
Resvd	Resvd	Resvd	HPINT	INT3	TXINT	TRINT	BXINT0	BRINT0	TINT	INT2	INT1	INT0

(c) '543 IFR

15–12	11	10	9	8	7	6	5	4	3	2	1	0
Resvd	Resvd	Resvd	Resvd	INT3	TXINT	TRINT	BXINT0	BRINT0	TINT	INT2	INT1	INT0

(d) '545 IFR

15–12	11	10	9	8	7	6	5	4	3	2	1	0
Resvd	Resvd	Resvd	HPINT	INT3	XINT1	RINT1	BXINT0	BRINT0	TINT	INT2	INT1	INT0

(e) '546 IFR

15–12	11	10	9	8	7	6	5	4	3	2	1	0
Resvd	Resvd	Resvd	Resvd	INT3	XINT1	RINT1	BXINT0	BRINT0	TINT	INT2	INT1	INT0

(f) '548 IFR

15–12	11	10	9	8	7	6	5	4	3	2	1	0
Resvd	BXINT1	BRINT1	HPINT	INT3	TXINT	TRINT	BXINT0	BRINT0	TINT	INT2	INT1	INT0

(g) '549 IFR

15–14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Resvd	BMINT1	BMINT0	BXINT1	BRINT1	HPINT	INT3	TXINT	TRINT	BXINT0	BRINT0	TINT	INT2	INT1	INT0

6.10.2 Interrupt Mask Register (IMR)

Figure 6–3 shows how the ’54x uses a memory-mapped IMR for masking external and internal interrupts. If INTM = 0 in ST1, a 1 in any IMR bit enables the corresponding interrupt. Neither $\overline{\text{NMI}}$ nor $\overline{\text{RS}}$ is included in the IMR, because IMR has no effect on these interrupts. You can read or write to the IMR.

Figure 6–3. Interrupt Mask Register (IMR) Diagram

(a) ’541 IMR

15–12	11	10	9	8	7	6	5	4	3	2	1	0
Resvd	Resvd	Resvd	Resvd	INT3	XINT1	RINT1	XINT0	RINT0	TINT	INT2	INT1	INT0

(b) ’542 IMR

15–12	11	10	9	8	7	6	5	4	3	2	1	0
Resvd	Resvd	Resvd	HPINT	INT3	TXINT	TRINT	BXINT0	BRINT0	TINT	INT2	INT1	INT0

(c) ’543 IMR

15–12	11	10	9	8	7	6	5	4	3	2	1	0
Resvd	Resvd	Resvd	Resvd	INT3	TXINT	TRINT	BXINT0	BRINT0	TINT	INT2	INT1	INT0

(d) ’545 IMR

15–12	11	10	9	8	7	6	5	4	3	2	1	0
Resvd	Resvd	Resvd	HPINT	INT3	XINT1	RINT1	BXINT0	BRINT0	TINT	INT2	INT1	INT0

(e) ’546 IMR

15–12	11	10	9	8	7	6	5	4	3	2	1	0
Resvd	Resvd	Resvd	Resvd	INT3	XINT1	RINT1	BXINT0	BRINT0	TINT	INT2	INT1	INT0

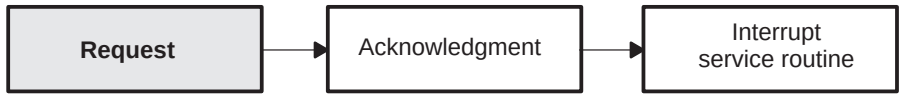
(f) ’548 IMR

15–12	11	10	9	8	7	6	5	4	3	2	1	0
Resvd	BXINT1	BRINT1	HPINT	INT3	TXINT	TRINT	BXINT0	BRINT0	TINT	INT2	INT1	INT0

(g) ’549 IMR

15–14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Resvd	BMINT1	BMINT0	BXINT1	BRINT1	HPINT	INT3	TXINT	TRINT	BXINT0	BRINT0	TINT	INT2	INT1	INT0

6.10.3 Phase 1: Receive Interrupt Request



An interrupt is requested by a hardware device or by a software instruction. When an interrupt request occurs, the corresponding flag (if any) is activated in the IFR (see subsection 6.10.1, *Interrupt Flag Register (IFR)*, on page 6-27). This flag is activated whether or not the interrupt is later acknowledged by the processor. The flag is automatically cleared when its corresponding interrupt is taken.

- Hardware interrupt requests. External hardware interrupts are requested by signals at external interrupt ports, and internal hardware interrupts are requested by signals from the on-chip peripherals. For example, on the ’541, hardware interrupts can be requested by or through:
  - Pins $\overline{\text{INT3}}$ through $\overline{\text{INT0}}$
  - Pins $\overline{\text{RS}}$ (reset) and $\overline{\text{NMI}}$
  - The serial ports interrupts (RINT0 and XINT0 or RINT1 and XINT1)
  - The timer interrupt (TINT)

Table 6–18 through Table 6–23 (pages 6-37 through 6-42) list the interrupt sources for each ’54x device.

- Software interrupt requests. A software interrupt is requested by one of the following program instructions:
  - INTR. This instruction allows you to execute any interrupt service routine. The instruction operand (K) indicates which interrupt vector location the CPU branches to. Table 6–18 through Table 6–23 (pages 6-37 through 6-42) show the operand K used to refer to each vector location. When an INTR interrupt is acknowledged, the interrupt mode bit (INTM) in ST1 is set to 1 to disable maskable interrupts.
  - TRAP. This instruction performs the same function as the INTR instruction without setting the INTM bit. Table 6–18 through Table 6–23 (pages 6-37 through 6-42) show the operand K used to refer to each vector location.
  - RESET. This instruction performs a nonmaskable software reset that can be used any time to put the ’54x into a known state. The RESET instruction affects ST0 and ST1, but does not affect PMST. For a summary of the registers and bits affected, see the description of the RESET instruction in *TMS320C54x DSP Reference Set, Volume 2: Mnemonic Instruction Set* or *Volume 3: Algebraic Instruction Set*.

When the RESET instruction is acknowledged, INTM is set to 1 to disable maskable interrupts. The initialization of IPTR and the peripheral registers is different from the initialization done by a hardware reset (see Section 6.9, *Reset Operation*, on page 6-25).

6.10.4 Phase 2: Acknowledge Interrupt

After an interrupt has been requested by hardware or software, the CPU must decide whether to acknowledge the request. Software interrupts and nonmaskable hardware interrupts are acknowledged immediately. Maskable hardware interrupts are acknowledged only after certain conditions are met:

- ❑ Priority is highest. When more than one hardware interrupt is requested at the same time, the '54x services them according to a set priority ranking in which 1 indicates the highest priority. Table 6–18 through Table 6–23 show the priorities for the hardware interrupts.

- ❑ INTM bit is 0. The interrupt mode bit (INTM), which is in ST1, enables or disables all maskable interrupts:

  - ■ When INTM = 0, all unmasked interrupts are enabled.
  - ■ When INTM = 1, all unmasked interrupts are disabled.

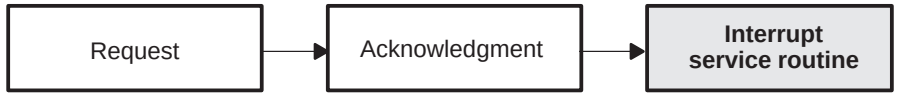
  INTM is set to 1 automatically when an interrupt is taken. If the program exits the interrupt service routine (ISR) using the RETE instruction (return from interrupt with automatic reenable), INTM is reenabled (cleared). INTM can also be set with a hardware reset ($\overline{RS}$) or by executing an SSBX INTM instruction (disable interrupt). INTM is reset by executing the RSBX INTM instruction (enable interrupt). INTM does not actually modify IMR or IFR.

- ❑ IMR mask bit is 1. Each of the maskable interrupts has its own mask bit in the IMR. To enable an interrupt, set its mask bit to 1. See subsection 6.10.2, *Interrupt Mask Register (IMR)*, on page 6-29.

The CPU acknowledges a maskable hardware interrupt, it jams the instruction bus with the INTR instruction. This instruction forces PC to the appropriate address and fetches the software vector. As the CPU fetches the first word of the software vector, it generates the $\overline{IACK}$ signal, which clears the appropriate interrupt flag bit.

For enabled interrupts, when $\overline{\text{IACK}}$ occurs, the interrupt number is indicated by address bits A6–A2 on the rising edge of CLKOUT. If the interrupt vectors reside in on-chip memory and you want to observe the addresses, the ’54x must operate in address visibility mode (AVIS = 1) so that the interrupt number can be decoded. If an interrupt occurs while the ’54x is on hold and HM = 0, the address cannot be present when $\overline{\text{IACK}}$ becomes active.

6.10.5 Phase 3: Execute Interrupt Service Routine (ISR)



After acknowledging the interrupt, the CPU:

1) Stores the program counter (PC) value (the return address) to the top of the stack in data memory

   (The program counter extension register, XPC, does not get pushed to the top of the stack; that is, it does not get saved on the stack.)

2) Loads the PC with the address of the interrupt vector

3) Fetches the instruction located at the vector address. (If the branch is delayed and you also stored one 2-word instruction or two 1-word instructions, the CPU also fetches these words.)

4) Executes the branch, which leads it to the address of your ISR. (If the branch is delayed, the additional instruction(s) are executed before the branch.)

5) Executes the ISR until a return instruction concludes the ISR

6) Follows the stack pointer (SP) to the top of the stack, and pops the return address off the stack and into PC

7) Continues executing the main program

To determine which vector address has been assigned to each of the interrupts, refer to the table for your specific ’54x device in subsection 6.10.10, *Interrupt Tables*, on page 6-37. Interrupt addresses are spaced four locations apart so that a delayed branch instruction and two 1-word instructions or one 2-word instruction can be accommodated in those locations.

6.10.6 Interrupt Context Save

When an interrupt service routine is executed, certain registers must be saved onto the stack. When the program returns from the ISR (by an RC[D], RETE[D], or RETF[D]), your software code must restore the contents of these registers. You can manage stack storage as long as the stack does not exceed the memory space. This stack is also used for subroutine calls; the ’54x supports subroutine calls within the ISR. Because the CPU registers and peripheral registers are memory-mapped, the PSHM and POPM instructions can transfer these registers to and from the stack. In addition, the PSHD and POPD instructions can transfer data-memory values to and from the stack.

There are a number of special considerations that you must follow when doing context saves and restores. The first consideration is that when you use the stack to save the context you must perform the restore in the exact reverse order. The second consideration is that BRC should be restored prior to restoring the BRAF bit in ST1. If you fail to follow this order, the BRAF bit will be cleared, if BRC = 0 before BRC is restored.

6.10.7 Interrupt Latency

The ’54x completes all instructions in the pipeline except the instructions in the prefetch and fetch stages before executing an interrupt, so the maximum interrupt latency depends on the contents of the pipeline. See Section 7.2, *Interrupts and the Pipeline*, on page 7-26 for more information about pipeline latencies associated with interrupts. Instructions that are extended by wait states for slower-memory access and repeated instructions require extra time to process an interrupt.

The single-repeat instructions (RPT and RPTZ) require that all executions of the next instruction be completed before allowing an interrupt to execute to protect the context of the repeated instructions. This protection is necessary, because these instructions run parallel operations in the pipeline, and the context of these operations cannot be saved in the ISR.

Since the hold function takes precedence over interrupts, it can also delay an interrupt trap. If an interrupt occurs when the CPU is on hold ($\overline{\text{HOLD}}$ is asserted) and the interrupt vector must be fetched from external memory, the interrupt is not taken until $\overline{\text{HOLDA}}$ is deasserted (after the hold state ends). However, if the processor is in the concurrent hold mode (HM = 0) and the interrupt vector table is located in internal memory, the CPU takes the interrupt, regardless of $\overline{\text{HOLD}}$.

Interrupts cannot be processed between the RSBX INTM instruction and the next instruction in a program sequence. If an interrupt occurs during the decode phase of RSBX INTM, the CPU always completes RSBX INTM as well as the following instruction before the pending interrupt is processed. Waiting for these instructions to complete ensures that a return (RET) can be executed in an ISR before the next interrupt is processed to protect against stack overflow. If an ISR ends with an RETE instruction (return from ISR with enable), the RSBX INTM instruction is unnecessary. Similar to an RSBX INTM instruction, an SSBX INTM instruction and the instruction that follows it cannot be interrupted.

Note:

Reset ($\overline{RS}$) is not delayed by multicycle instructions. $\overline{NMI}$ can be delayed by multicycle instructions and by $\overline{HOLD}$.

6.10.8 Interrupt Operation: A Quick Summary

Once an interrupt has been passed to the CPU, the CPU operates in the following manner (see Figure 6–5 on page 6-36):

- If a maskable interrupt is requested:
  1) The corresponding bit in the IFR is set.
  2) The acknowledgment conditions (INTM = 0 and IMR bit =1) are tested. If the conditions are true, the CPU acknowledges the interrupt, generating an $\overline{IACK}$ signal; otherwise, it ignores the interrupt and continues with the main program.
  3) When the interrupt has been acknowledged, its flag bit in the IFR is cleared to 0 and the INTM bit is set to 1 (to block other maskable interrupts).
  4) The PC is saved on the stack.
  5) The CPU branches to and executes the interrupt service routine (ISR).
  6) The ISR is concluded by a return instruction, which pops the return address off the stack.
  7) The CPU continues with the main program.
- If a nonmaskable interrupt is requested:
  1) The CPU immediately acknowledges the interrupt, generating an $\overline{IACK}$ signal.
  2) If the interrupt was requested by $\overline{RS}$, $\overline{NMI}$, or the INTR instruction, the the INTM bit is set to 1 to block maskable hardware interrupts.

3) If the INTR instruction has requested one of the maskable interrupts, the corresponding flag bit is cleared to 0.
4) The PC is saved on the stack.
5) The CPU branches to and executes the ISR.
6) The ISR is concluded by a return instruction, which pops the return address of the stack.
7) The CPU continues with the main program.

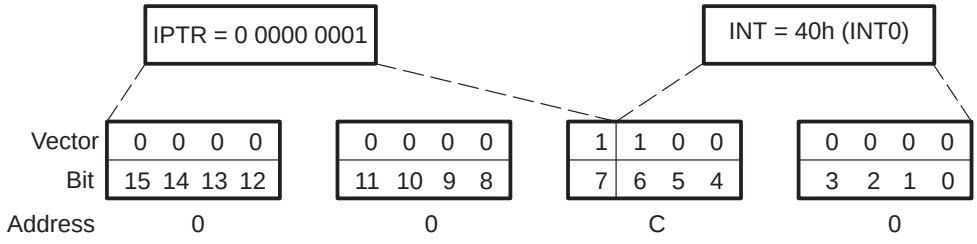
Note:

The INTR instruction disables maskable interrupts by setting the interrupt mode bit (INTM), but the TRAP instruction does not affect INTM.

6.10.9 Remapping Interrupt-Vector Addresses

The interrupt vectors can be remapped to the beginning of any 128-word page in program memory except in reserved areas. The interrupt-vector address is generated by concatenating the interrupt-pointer (IPTR) field of PMST with the interrupt-vector number (0–31) shifted by 2. Consider the example of Figure 6–4: if $\overline{\text{INT0}}$ is asserted low and IPTR = 0001h, the interrupt vector is fetched from 00C0h. The interrupt-vector number for $\overline{\text{INT0}}$ is 16 or 10h.

Figure 6–4. Interrupt-Vector Address Generation

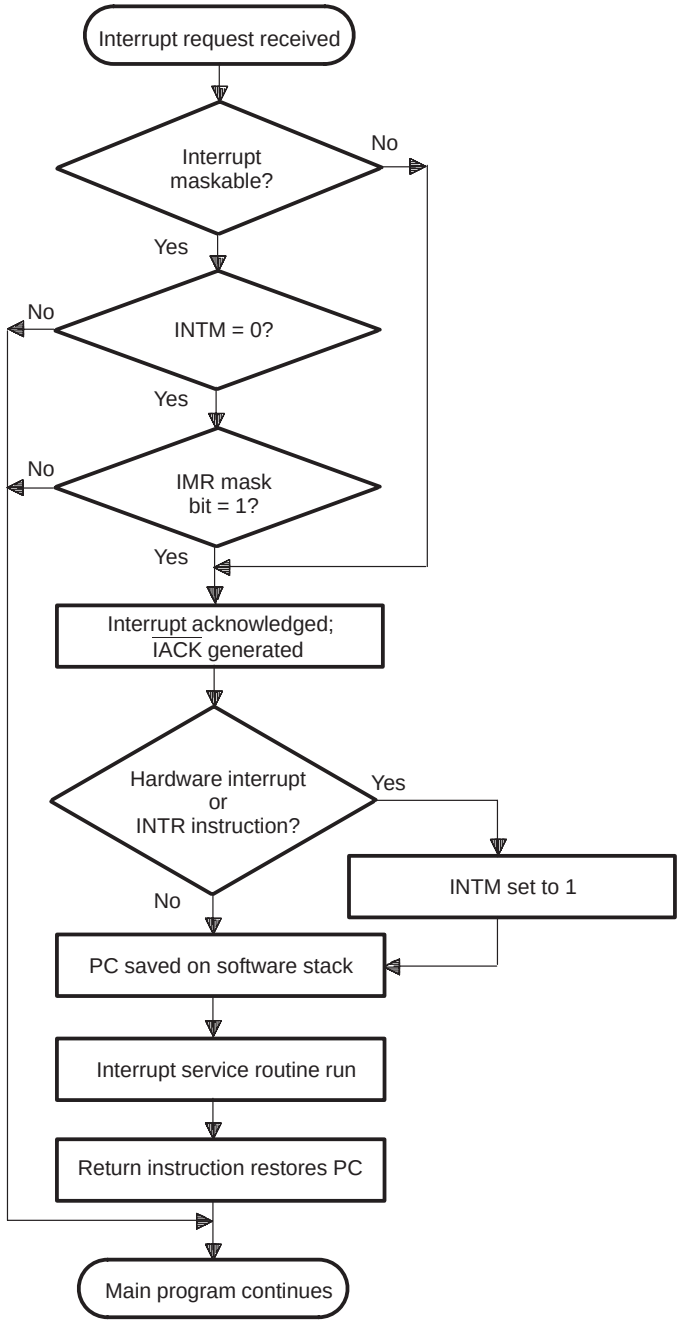


At reset, the IPTR bits are set to 1 (IPTR = 1FFh); this value maps the vectors to page 511 in program-memory space. Therefore, the reset vector for hardware resets always resides at location 0FF80h. The interrupt vectors can be mapped to another location by loading IPTR with a value other than 1FFh. For example, the interrupt vectors can be moved to start at location 0080h by loading IPTR with 0001h.

Note:

The hardware reset ($\overline{\text{RS}}$) vector cannot be remapped because the hardware reset loads the IPTR with 1s. Therefore, the reset vector for hardware resets is always fetched at location FF80h in program space.

Figure 6–5. Flow Diagram of Interrupt Operation



6.10.10 Interrupt Tables

Table 6–18 through Table 6–23 show the interrupt trap number, priority, and location for each ’54x device.

Table 6–18. TMS320C541 Interrupt Locations and Priorities

TRAP/INTR Number (K)	Priority	Name	Location	Function
0	1	$\overline{\text{RS}}$/SINTR	0	Reset (hardware and software reset)
1	2	$\overline{\text{NMI}}$/SINT16	4	Nonmaskable interrupt
2	–	SINT17	8	Software interrupt #17
3	–	SINT18	C	Software interrupt #18
4	–	SINT19	10	Software interrupt #19
5	–	SINT20	14	Software interrupt #20
6	–	SINT21	18	Software interrupt #21
7	–	SINT22	1C	Software interrupt #22
8	–	SINT23	20	Software interrupt #23
9	–	SINT24	24	Software interrupt #24
10	–	SINT25	28	Software interrupt #25
11	–	SINT26	2C	Software interrupt #26
12	–	SINT27	30	Software interrupt #27
13	–	SINT28	34	Software interrupt #28
14	–	SINT29	38	Software interrupt #29; reserved
15	–	SINT30	3C	Software interrupt #30; reserved
16	3	$\overline{\text{INT0}}$/SINT0	40	External user interrupt #0
17	4	$\overline{\text{INT1}}$/SINT1	44	External user interrupt #1
18	5	$\overline{\text{INT2}}$/SINT2	48	External user interrupt #2
19	6	TINT/SINT3	4C	Internal timer interrupt
20	7	RINT0/SINT4	50	Serial port 0 receive interrupt
21	8	XINT0/SINT5	54	Serial port 0 transmit interrupt
22	9	RINT1/SINT6	58	Serial port 1 receive interrupt
23	10	XINT1/SINT7	5C	Serial port 1 transmit interrupt
24	11	$\overline{\text{INT3}}$/SINT8	60	External user interrupt #3
25–31	–		64–7F	Reserved

Table 6–19. TMS320C542 Interrupt Locations and Priorities

TRAP/INTR Number (K)	Priority	Name	Location	Function
0	1	$\overline{RS}$/SINTR	0	Reset (hardware and software reset)
1	2	$\overline{NMI}$/SINT16	4	Nonmaskable interrupt
2	–	SINT17	8	Software interrupt #17
3	–	SINT18	C	Software interrupt #18
4	–	SINT19	10	Software interrupt #19
5	–	SINT20	14	Software interrupt #20
6	–	SINT21	18	Software interrupt #21
7	–	SINT22	1C	Software interrupt #22
8	–	SINT23	20	Software interrupt #23
9	–	SINT24	24	Software interrupt #24
10	–	SINT25	28	Software interrupt #25
11	–	SINT26	2C	Software interrupt #26
12	–	SINT27	30	Software interrupt #27
13	–	SINT28	34	Software interrupt #28
14	–	SINT29	38	Software interrupt #29, reserved
15	–	SINT30	3C	Software interrupt #30, reserved
16	3	$\overline{INT0}$/SINT0	40	External user interrupt #0
17	4	$\overline{INT1}$/SINT1	44	External user interrupt #1
18	5	$\overline{INT2}$/SINT2	48	External user interrupt #2
19	6	TINT/SINT3	4C	Internal timer interrupt
20	7	BRINT0/SINT4	50	Buffered serial port receive interrupt
21	8	BXINT0/SINT5	54	Buffered serial port transmit interrupt
22	9	TRINT/SINT6	58	TDM serial port receive interrupt
23	10	TXINT/SINT7	5C	TDM serial port transmit interrupt
24	11	$\overline{INT3}$/SINT8	60	External user interrupt #3
25	12	$\overline{HPINT}$/SINT9	64	HPI interrupt
26–31	–		68–7F	Reserved

Table 6–20. TMS320C543 Interrupt Locations and Priorities

TRAP/INTR Number (K)	Priority	Name	Location	Function
0	1	$\overline{\text{RS}}$/SINTR	0	Reset (hardware and software reset)
1	2	$\overline{\text{NMI}}$/SINT16	4	Nonmaskable interrupt
2	–	SINT17	8	Software interrupt #17
3	–	SINT18	C	Software interrupt #18
4	–	SINT19	10	Software interrupt #19
5	–	SINT20	14	Software interrupt #20
6	–	SINT21	18	Software interrupt #21
7	–	SINT22	1C	Software interrupt #22
8	–	SINT23	20	Software interrupt #23
9	–	SINT24	24	Software interrupt #24
10	–	SINT25	28	Software interrupt #25
11	–	SINT26	2C	Software interrupt #26
12	–	SINT27	30	Software interrupt #27
13	–	SINT28	34	Software interrupt #28
14	–	SINT29	38	Software interrupt #29, reserved
15	–	SINT30	3C	Software interrupt #30, reserved
16	3	$\overline{\text{INT0}}$/SINT0	40	External user interrupt #0
17	4	$\overline{\text{INT1}}$/SINT1	44	External user interrupt #1
18	5	$\overline{\text{INT2}}$/SINT2	48	External user interrupt #2
19	6	TINT/SINT3	4C	Internal timer interrupt
20	7	BRINT0/SINT4	50	Buffered serial port receive interrupt
21	8	BXINT0/SINT5	54	Buffered serial port transmit interrupt
22	9	TRINT/SINT6	58	TDM serial port receive interrupt
23	10	TXINT/SINT7	5C	TDM serial port transmit interrupt
24	11	$\overline{\text{INT3}}$/SINT8	60	External user interrupt #3
25–31	–		64–7F	Reserved

Table 6–21. TMS320C545 Interrupt Locations and Priorities

TRAP/INTR Number (K)	Priority	Name	Location	Function
0	1	$\overline{RS}$/SINTR	0	Reset (hardware and software reset)
1	2	$\overline{NMI}$/SINT16	4	Nonmaskable interrupt
2	–	SINT17	8	Software interrupt #17
3	–	SINT18	C	Software interrupt #18
4	–	SINT19	10	Software interrupt #19
5	–	SINT20	14	Software interrupt #20
6	–	SINT21	18	Software interrupt #21
7	–	SINT22	1C	Software interrupt #22
8	–	SINT23	20	Software interrupt #23
9	–	SINT24	24	Software interrupt #24
10	–	SINT25	28	Software interrupt #25
11	–	SINT26	2C	Software interrupt #26
12	–	SINT27	30	Software interrupt #27
13	–	SINT28	34	Software interrupt #28
14	–	SINT29	38	Software interrupt #29, reserved
15	–	SINT30	3C	Software interrupt #30, reserved
16	3	$\overline{INT0}$/SINT0	40	External user interrupt #0
17	4	$\overline{INT1}$/SINT1	44	External user interrupt #1
18	5	$\overline{INT2}$/SINT2	48	External user interrupt #2
19	6	TINT/SINT3	4C	Internal timer interrupt
20	7	BRINT0/SINT4	50	Buffered serial port receive interrupt
21	8	BXINT0/SINT5	54	Buffered serial port transmit interrupt
22	9	RINT1/SINT6	58	Serial port receive interrupt
23	10	XINT1/SINT7	5C	Serial port transmit interrupt
24	11	$\overline{INT3}$/SINT8	60	External user interrupt #3
25	12	$\overline{HPINT}$/SINT9	64	HPI interrupt
26–31	–		68–7F	Reserved

Table 6–22. TMS320C546 Interrupt Locations and Priorities

TRAP/INTR Number (K)	Priority	Name	Location	Function
0	1	$\overline{\text{RS}}$/SINTR	0	Reset (hardware and software reset)
1	2	$\overline{\text{NMI}}$/SINT16	4	Nonmaskable interrupt
2	–	SINT17	8	Software interrupt #17
3	–	SINT18	C	Software interrupt #18
4	–	SINT19	10	Software interrupt #19
5	–	SINT20	14	Software interrupt #20
6	–	SINT21	18	Software interrupt #21
7	–	SINT22	1C	Software interrupt #22
8	–	SINT23	20	Software interrupt #23
9	–	SINT24	24	Software interrupt #24
10	–	SINT25	28	Software interrupt #25
11	–	SINT26	2C	Software interrupt #26
12	–	SINT27	30	Software interrupt #27
13	–	SINT28	34	Software interrupt #28
14	–	SINT29	38	Software interrupt #29, reserved
15	–	SINT30	3C	Software interrupt #30, reserved
16	3	$\overline{\text{INT0}}$/SINT0	40	External user interrupt #0
17	4	$\overline{\text{INT1}}$/SINT1	44	External user interrupt #1
18	5	$\overline{\text{INT2}}$/SINT2	48	External user interrupt #2
19	6	TINT/SINT3	4C	Internal timer interrupt
20	7	BRINT0/SINT4	50	Buffered serial port receive interrupt
21	8	BXINT0/SINT5	54	Buffered serial port transmit interrupt
22	9	RINT1/SINT6	58	Serial port receive interrupt
23	10	XINT1/SINT7	5C	Serial port transmit interrupt
24	11	$\overline{\text{INT3}}$/SINT8	60	External user interrupt #3
25–31	–		64–7F	Reserved

Table 6–23. TMS320C548 Interrupt Locations and Priorities

TRAP/INTR Number (K)	Priority	Name	Location	Function
0	1	$\overline{RS}$/SINTR	0	Reset (hardware and software reset)
1	2	$\overline{NMI}$/SINT16	4	Nonmaskable interrupt
2	–	SINT17	8	Software interrupt #17
3	–	SINT18	C	Software interrupt #18
4	–	SINT19	10	Software interrupt #19
5	–	SINT20	14	Software interrupt #20
6	–	SINT21	18	Software interrupt #21
7	–	SINT22	1C	Software interrupt #22
8	–	SINT23	20	Software interrupt #23
9	–	SINT24	24	Software interrupt #24
10	–	SINT25	28	Software interrupt #25
11	–	SINT26	2C	Software interrupt #26
12	–	SINT27	30	Software interrupt #27
13	–	SINT28	34	Software interrupt #28
14	–	SINT29	38	Software interrupt #29, reserved
15	–	SINT30	3C	Software interrupt #30, reserved
16	3	$\overline{INT0}$/SINT0	40	External user interrupt #0
17	4	$\overline{INT1}$/SINT1	44	External user interrupt #1
18	5	$\overline{INT2}$/SINT2	48	External user interrupt #2
19	6	TINT/SINT3	4C	Internal timer interrupt
20	7	BRINT0/SINT4	50	Buffered serial port 0 receive interrupt
21	8	BXINT0/SINT5	54	Buffered serial port 0 transmit interrupt
22	9	TRINT/SINT6	58	TDM serial port receive interrupt
23	10	TXINT/SINT7	5C	TDM serial port transmit interrupt
24	11	$\overline{INT3}$/SINT8	60	External user interrupt #3
25	12	$\overline{HPINT}$/SINT9	64	HPI interrupt
26	13	BRINT1/SINT10	68	Buffered serial port 1 receive interrupt
27	14	BXINT1/SINT11	6C	Buffered serial port 1 transmit interrupt
28–31	–		70h–7F	Reserved

Table 6–24. TMS320C549 Interrupt Locations and Priorities

TRAP/INTR Number (K)	Priority	Name	Location	Function
0	1	$\overline{\text{RS}}$/SINTR	0	Reset (hardware and software reset)
1	2	$\overline{\text{NMI}}$/SINT16	4	Nonmaskable interrupt
2	–	SINT17	8	Software interrupt #17
3	–	SINT18	C	Software interrupt #18
4	–	SINT19	10	Software interrupt #19
5	–	SINT20	14	Software interrupt #20
6	–	SINT21	18	Software interrupt #21
7	–	SINT22	1C	Software interrupt #22
8	–	SINT23	20	Software interrupt #23
9	–	SINT24	24	Software interrupt #24
10	–	SINT25	28	Software interrupt #25
11	–	SINT26	2C	Software interrupt #26
12	–	SINT27	30	Software interrupt #27
13	–	SINT28	34	Software interrupt #28
14	–	SINT29	38	Software interrupt #29
15	–	SINT30	3C	Software interrupt #30
16	3	$\overline{\text{INT0}}$/SINT0	40	External user interrupt #0
17	4	$\overline{\text{INT1}}$/SINT1	44	External user interrupt #1
18	5	$\overline{\text{INT2}}$/SINT2	48	External user interrupt #2
19	6	TINT/SINT3	4C	Internal timer interrupt
20	7	BRINT0/SINT4	50	Buffered serial port 0 receive interrupt
21	8	BXINT0/SINT5	54	Buffered serial port 0 transmit interrupt
22	9	TRINT/SINT6	58	TDM serial port receive interrupt
23	10	TXINT/SINT7	5C	TDM serial port transmit interrupt
24	11	$\overline{\text{INT3}}$/SINT8	60	External user interrupt #3
25	12	$\overline{\text{HINT}}$/SINT9	64	HPI interrupt
26	13	BRINT1/SINT10	68	Buffered serial port 1 receive interrupt
27	14	BXINT1/SINT11	6C	Buffered serial port 1 transmit interrupt

Table 6–24. TMS320C549 Interrupt Locations and Priorities (Continued)

TRAP/INTR Number (K)	Priority	Name	Location	Function
28	15	BMINT0/SINT12	70	BSP #0 misalignment detection interrupt (’549 only)
29	16	BMINT1/SINT13	74	BSP #1 misalignment detection interrupt (’549 only)
30–31	–		78–7F	Reserved

6.11 Power-Down Modes

The ’54x has power-down modes in which it enters a dormant state and dissipates less power than normal operation while maintaining the CPU contents. This allows operations to continue unaltered when the power-down mode is terminated.

You can invoke one of the power-down modes either by executing the IDLE 1, IDLE 2 or IDLE 3 instructions, or by driving the $\overline{\text{HOLD}}$ signal low with the HM status bit set to 1. Power-down operation is summarized in Table 6–25 and described in detail in subsections 6.11.1 through 6.11.5.

Table 6–25. Operation During the Four Power-Down Modes

Operation/Feature	IDLE1	IDLE2	IDLE3	$\overline{\text{HOLD}}$
CPU halted	Yes	Yes	Yes	Yes†
CPU clock stopped	Yes	Yes	Yes	No
Peripheral clock stopped	No	Yes	Yes	No
Phase-locked loop (PLL) stopped	No	No	Yes	No
External address lines put in high-impedance state	No	No	No	Yes
External data lines put in high-impedance state	No	No	No	Yes
External control signals put in high-impedance state	No	No	No	Yes
Power-down terminated by:				
HOLD driven high	No	No	No	Yes
Unmasked internal hardware interrupts	Yes	No	No	No
Unmasked external hardware interrupts	Yes	Yes	Yes	No
$\overline{\text{NMI}}$	Yes	Yes	Yes	No
$\overline{\text{RS}}$	Yes	Yes	Yes	No

† Depending on the state of the HM bit, the CPU continues to execute unless the execution requires an external memory access.

6.11.1 IDLE1 Mode

The IDLE1 mode halts all CPU activities except the system clock. Because the system clock remains applied to the peripheral modules, the peripheral circuits continue operating and the CLKOUT pin remains active. Thus, peripherals such as serial ports and timers can take the CPU out of its power-down state.

Use the IDLE 1 instruction to enter the IDLE1 mode. To terminate IDLE1, use a wake-up interrupt. If INTM = 0 when the wake-up interrupt takes place, the

’54x enters the ISR when IDLE1 is terminated. If INTM = 1, the ’54x continues with the instruction following the IDLE 1 instruction. All wake-up interrupts must set to enable the corresponding bits in the IMR register regardless of the INTM value. The only exceptions are the nonmaskable interrupts, $\overline{RS}$ and $\overline{NMI}$.

6.11.2 IDLE2 Mode

The IDLE2 mode halts the on-chip peripherals as well as the CPU. Because the on-chip peripherals are stopped in this mode, they cannot be used to generate the interrupt to wake up the ’54x as with IDLE1. However, power is significantly reduced because the device is completely stopped.

Use the IDLE 2 instruction to enter the IDLE2 mode. To terminate IDLE2, activate any of the external interrupt pins ($\overline{RS}$, $\overline{NMI}$, and $\overline{INTx}$) with a 10-ns minimum pulse. If INTM = 0 when the wake-up interrupt takes place, the ’54x enters the ISR when IDLE2 is terminated. If INTM = 1, the ’54x continues with the instruction following IDLE 2 instruction. All wake-up interrupts must be set to enable the corresponding bits in the IMR register regardless of the INTM value. Reset all peripherals when IDLE2 terminates, especially if they are externally clocked.

When $\overline{RS}$ is the wake-up interrupt in IDLE2, a 10-ns minimum pulse of $\overline{RS}$ can activate the reset sequence.

6.11.3 IDLE3 Mode

The IDLE3 mode functions like IDLE2 but it also halts the PLL. IDLE3 is used for a complete shutdown of the ’54x. This mode reduces power dissipation more than IDLE2. Furthermore, the IDLE3 state allows you to reconfigure the PLL externally if the system requires the ’54x to operate at a lower speed to save power.

Use the IDLE 3 instruction to enter the IDLE3 mode. To terminate IDLE3, activate any of the external interrupt pins ($\overline{RS}$, $\overline{NMI}$, and $\overline{INTx}$) with a 10-ns minimum pulse. If INTM = 0 when the wake-up interrupt takes place, the ’54x enters the ISR when IDLE3 is terminated. If INTM = 1, the ’54x continues with the instruction following the IDLE 3 instruction. All wake-up interrupts should be set to enable the corresponding bits on the IMR register, regardless of the INTM value. Reset all peripherals when IDLE3 terminates, especially if they are externally clocked.

To terminate IDLE3, the external interrupt must be a minimum of 10 ns to activate the wake-up sequence. The ’54x can accept multiple interrupts during the wake-up sequence; the interrupt with highest priority is serviced first after IDLE3. See subsection 10.5.2, *IDLE3*, on page 10-25, and subsection 8.4.2, *Software-Programmable PLL (Available on TMS320C545LP/546LP/548),* on page 8-19 for more details on PLL lockup time requirements.

When $\overline{RS}$ is the wake-up interrupt in IDLE3, a 10-ns minimum pulse of $\overline{RS}$ can activate the reset sequence. However, $\overline{RS}$ should be kept active for 50 μs so that the PLL can secure and provide stable system clock to internal logic.

6.11.4 Hold Mode

The Hold mode is another power-down mode. It enables you to put the address, data, and control lines into the high-impedance state. Depending on the value of the HM bit, you can also use this mode to halt the CPU.

This power-down mode is initiated by the $\overline{HOLD}$ signal. The effect of $\overline{HOLD}$ depends on the value of HM. If HM = 1, the CPU stops executing and address, data, and control lines go into the high-impedance state for further power reduction. If HM = 0, the address, data, and control signals are put into the high-impedance state, but the CPU continues to execute internally. You can use HM = 0 with the $\overline{HOLD}$ signal when your system does not require external-memory accesses. The ’54x continues to operate normally unless an off-chip access is required by an instruction; then the processor halts until $\overline{HOLD}$ is released.

This mode does not stop the operation of on-chip peripherals (such as timers and serial ports); they continue to operate regardless of the $\overline{HOLD}$ level or the condition of the HM bit.

This mode is terminated when $\overline{HOLD}$ becomes inactive.

6.11.5 Other Power-Down Capabilities

The ’54x has two other functions that affect the power-down operation: external bus off and CLKOUT off.

- **External bus off** allows the ’54x to disable the internal clock of external interfaces, thus placing the interface into a lower power-consumption mode.

  The external interface clock is disabled by setting bit 0 of the bank-switching control register (BSCR) to 1. At reset, this bit is cleared to 0 and the external interface clock is enabled. See subsection 10.3.2, *Bank-Switching Logic*, on page 10-8, for more information.

- **CLKOUT off** allows the ’54x to disable CLKOUT using software instructions. The CLKOFF bit of PMST determines whether CLKOUT is enabled or disabled. See subsection 4.1.2, *Processor Mode Status Register (PMST)*, on page 4-6. At reset, CLKOUT is enabled.

Chapter 7

Pipeline

This chapter describes the ’54x pipeline operation and lists the pipeline latency cycles for operations with various registers.

Topic **Page**

7.1 Pipeline Operation

The ’54x CPU has a six-level deep instruction pipeline. The six stages of the pipeline are independent of each other, which allows overlapping execution of instructions. During any given cycle, from one to six different instructions can be active, each at a different stage of completion.

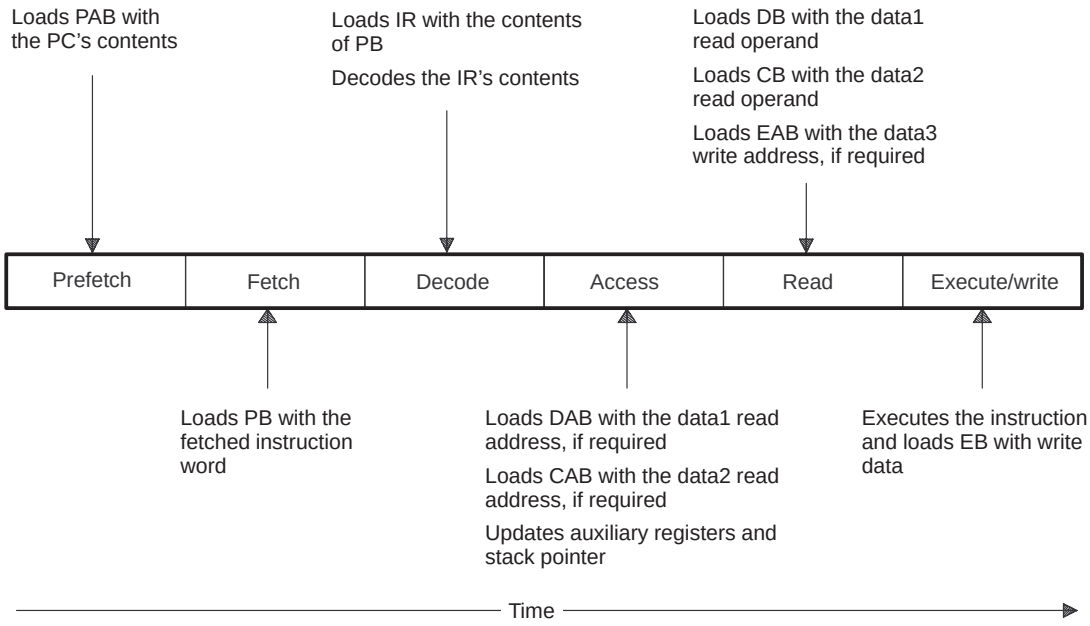
The six levels and functions of the pipeline structure are:

- Program prefetch. Program address bus (PAB) is loaded with the address of the next instruction to be fetched.
- Program fetch. An instruction word is fetched from the program bus (PB) and loaded into the instruction register (IR). This completes an instruction fetch sequence that consists of this and the previous cycle.
- Decode. The contents of the instruction register (IR) are decoded to determine the type of memory access operation and the control sequence at the data-address generation unit (DAGEN) and the CPU.
- Access. DAGEN outputs the read operand’s address on the data address bus, DAB. If a second operand is required, the other data address bus, CAB, is also loaded with an appropriate address. Auxiliary registers in indirect addressing mode and the stack pointer (SP) are also updated. This is considered the first of the 2-stage operand read sequence.
- Read. The read data operand(s), if any, are read from the data buses, DB and CB. This completes the two-stage operand read sequence. At the same time, the two-stage operand write sequence begins. The data address of the write operand, if any, is loaded into the data write address bus (EAB). For memory-mapped registers, the read data operand is read from memory and written into the selected memory-mapped registers using the DB.
- Execute. The operand write sequence is completed by writing the data using the data write bus (EB). The instruction is executed in this phase.

Figure 7–1 shows the six stages of the pipeline and the events that occur in each stage.

The first two stages of the pipeline, prefetch and fetch, are the instruction fetch sequence. In one cycle, the address of a new instruction is loaded. In the following cycle, an instruction word is read. In case of multiword instructions, several such instruction fetch sequences are needed.

Figure 7–1. Pipeline Stages



During the third stage of the pipeline, decode, the fetched instruction is decoded so that appropriate control sequences are activated for proper execution of the instruction.

The next two pipeline stages, access and read, are an operand read sequence. If required by the instruction, the data address of one or two operands are loaded in the access phase and the operand or operands are read in the following read phase.

Any write operation is spread over two stages of the pipeline, the read and execute stages. During the read phase, the data address of the write operand is loaded onto EAB. In the following cycle, the operand is written to memory using EB.

Each memory access is performed in two phases by the ’54x pipeline. In the first phase, an address bus is loaded with the memory address. In the second phase, a corresponding data bus reads from or writes to that memory address. Figure 7–2 shows how various memory accesses are performed by the ’54x pipeline. It is assumed that all memory accesses in the figure are performed by single-cycle, single-word instructions to on-chip dual-access memory. The on-chip dual-access memory can actually support two accesses in a single pipeline cycle. This is discussed in Section 7.3, *Dual-Access Memory and the Pipeline*, on page 7-28.

Figure 7–2. Pipelined Memory Accesses

(a) Instruction word fetch (one cycle)

Prefetch	Fetch	Decode	Access	Read	Execute/Write
Load PAB	Read from PB				

*(b) Instruction performing single operand read (for example, LD *AR1, A; one cycle)*

Prefetch	Fetch	Decode	Access	Read	Execute/Write
			Load DAB	Read from DB	

*(c) Instruction performing dual-operand read (for example, MAC *AR2+, *AR3+, A or DLD *AR2, A; one cycle)*

Prefetch	Fetch	Decode	Access	Read	Execute/Write
			Load DAB and CAB	Read from DB and CB	

*(d) Instruction performing single-operand write (for example, STH A, *AR1; one cycle)*

Prefetch	Fetch	Decode	Access	Read	Execute/Write
				Load EAB	Write to EB

*(e) Instruction performing dual-operand write, (for example, DST A, *AR1; two cycles)*

Prefetch	Fetch	Decode	Access	Read	Execute
				Load EAB	Write to EAB

Prefetch	Fetch	Decode	Access	Read	Execute/Write
				Load EAB	Write to EB

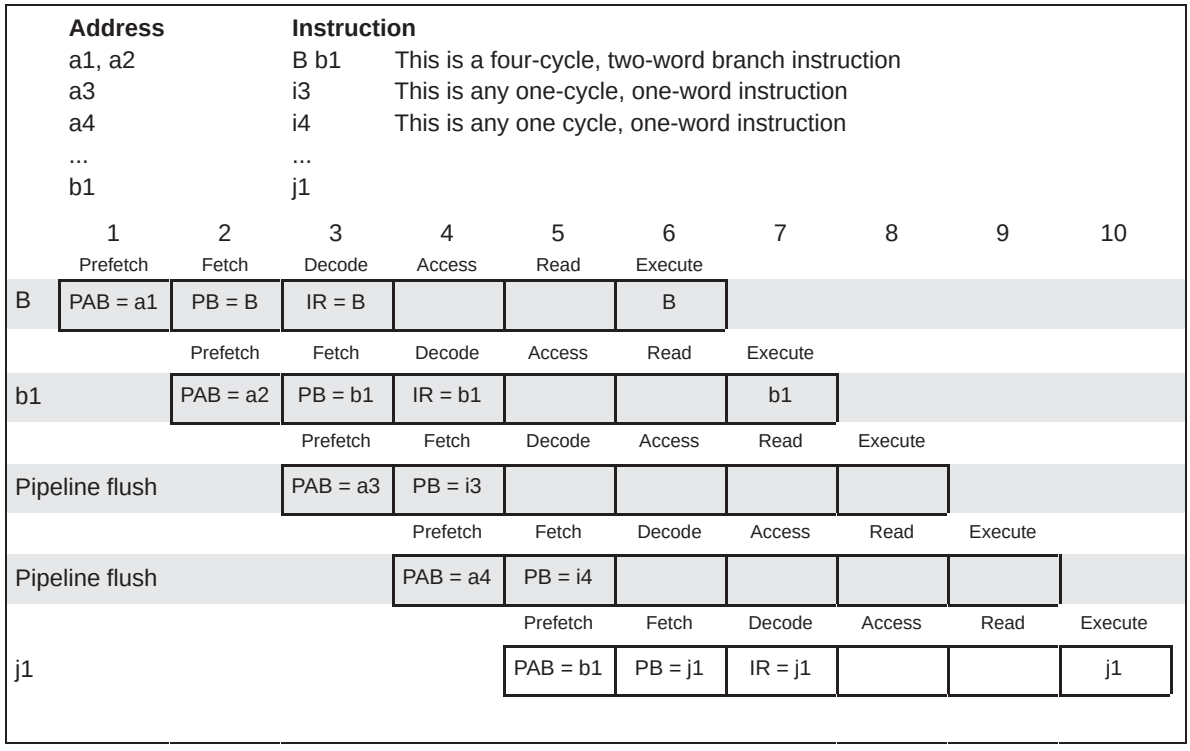
*(f) Instruction performing operand read and operand write (for example, ST A, *AR2 || LD *AR3, B; one cycle)*

Prefetch	Fetch	Decode	Access	Read	Execute/Write
			Load DAB	Read from DB Loads EAB	Write to EB

The following subsections provide examples that demonstrate how the pipeline works while executing different types of instructions. Unless otherwise noted, all instructions shown in the examples are considered single-cycle, single-word instructions residing in on-chip memory.

The pipeline is depicted in these examples as a set of staggered rows in which each row corresponds to one instruction word moving through the stages of the pipeline. Example 7–1 is a sample pipeline diagram.

Example 7–1. Sample Pipeline Diagram



Each row in the example is labeled on the left as an instruction, an operand, a multicycle instruction, or a pipeline flush. The numbers across the top represent single instruction cycles. Some cycles do not show all pipeline stages—this is done intentionally to avoid displaying unnecessary information.

Each box in the example contains relevant actions that occur at that pipeline stage. The name of each pipeline stage is shown above the box in which the action occurs.

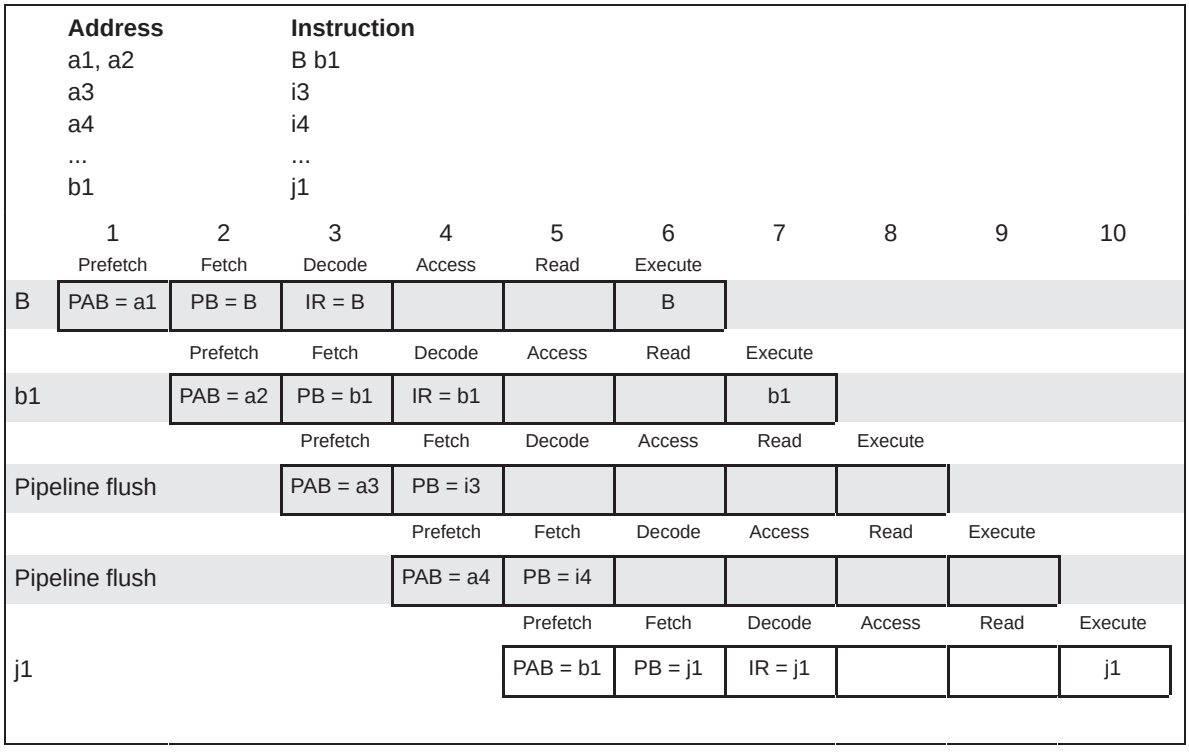
Shading represents all instruction fetches and pipeline flushes that are necessary to complete the instruction whose operation is shown.

7.1.1 Branch Instructions in the Pipeline

Example 7–2 and Example 7–3 show the pipeline's behavior during the execution of a branch (B) instruction and a delayed-branch (BD) instruction, respectively.

Because a branch instruction consists of two instruction words, it should take at least two instruction cycles to execute completely. However, a standard branch instruction actually takes four cycles to execute. This is illustrated in Example 7–2.

Example 7–2. Branch Instruction in the Pipeline



For the branch instruction in Example 7–2 to execute completely, the following events occur:

Cycle 1:	The PAB is loaded with the address of a branch instruction.
Cycles 2 and 3:	Two words of the branch instruction are fetched.

Cycles 4 and 5: Two more instructions, i3 and i4, are fetched. Although the two instructions after the branch instruction, i3 and i4, are fetched by the ’54x, they are not allowed to move past the decode stage and are eventually discarded. After the second word of the branch instruction (represented by b1 in the left column) is decoded, PAB is loaded with this new value (in cycle 5).

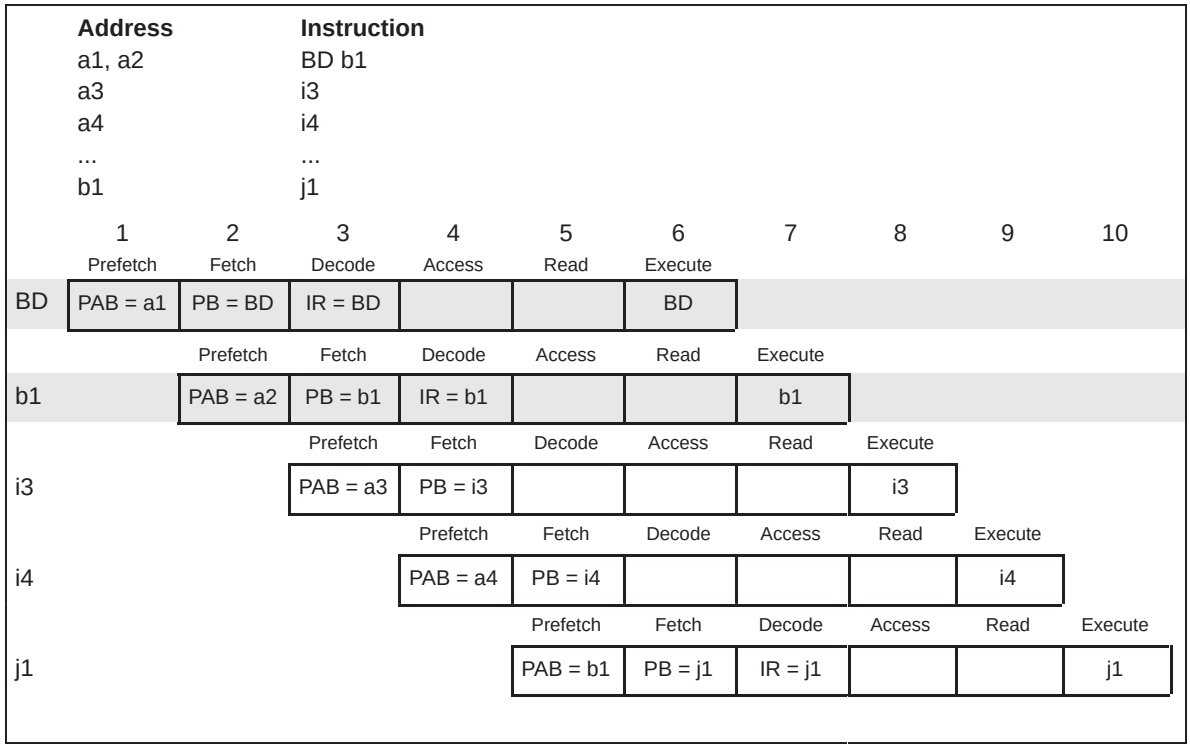
Cycles 6 and 7: The two-word branch instruction enters the execution stage of the pipeline in cycles 6 and 7. Also, j1 is fetched from address b1 in cycle 6.

Cycles 8 and 9: These cycles are also consumed by the same branch instruction since the next two instructions, i3 and i4, were not allowed to complete their execution; this is why a branch instruction takes four cycles to execute.

Cycle 10: j1 completes execution.

Example 7–3 shows the pipeline’s behavior for a delayed-branch instruction.

Example 7–3. Delayed-Branch Instruction in the Pipeline



In this case, the pipeline behaves in the same manner as it did for the regular branch instruction. However, the two instructions following the branch, i3 and i4, are allowed to complete their execution. Therefore, only cycles 6 and 7 are consumed by the delayed-branch instruction, making the delayed branch into a 2-cycle instruction.

7.1.2 Call Instructions in the Pipeline

A standard call instruction takes four cycles to execute. Although a standard call is a two-word instruction and seems to need only two cycles, it actually flushes the pipeline for two cycles, taking four cycles to execute.

Example 7–4 shows the pipeline's behavior during the execution of a call instruction.

Example 7–4. Call Instruction in the Pipeline

Address	Instruction
a1, a2	CALL b1
a3	i3
a4	i4
...	...
b1	j1

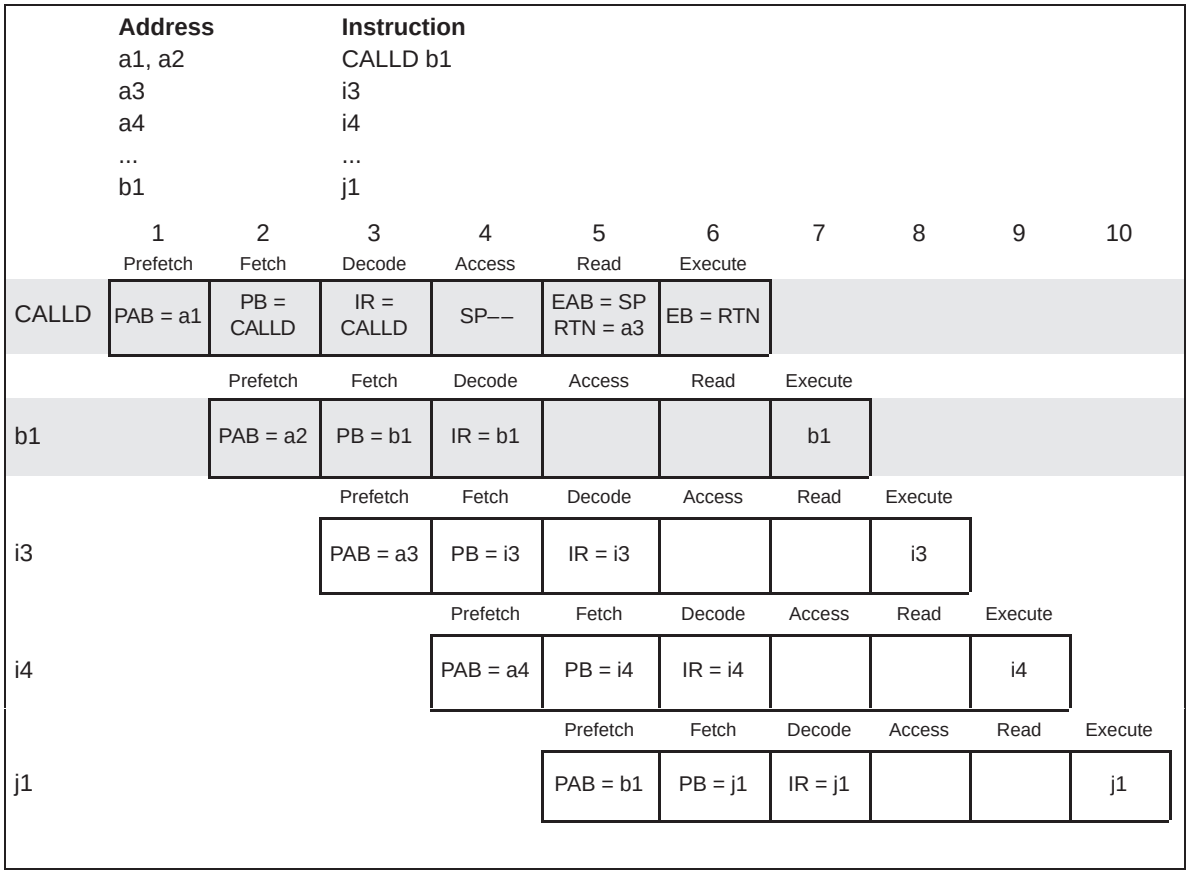
	1	2	3	4	5	6	7	8	9	10
	Prefetch	Fetch	Decode	Access	Read	Execute				
CALL	PAB = a1	PB = CALL	IR = CALL	SP– –	EAB = SP RTN = a3	EB = RTN				
		Prefetch	Fetch	Decode	Access	Read	Execute			
b1		PAB = a2	PB = b1	IR = b1			b1			
			Prefetch	Fetch	Decode	Access	Read	Execute		
Pipeline flush			PAB = a3	PB = i3						
				Prefetch	Fetch	Decode	Access	Read	Execute	
Pipeline flush				PAB = a4	PB = i4					
					Prefetch	Fetch	Decode	Access	Read	Execute
j1					PAB = b1	PB = j1	IR = j1			j1

In Example 7–4, the following events occur:

Cycle 1:	PAB is loaded with the address of the call instruction.
Cycles 2 and 3:	Two words of the call instruction are fetched.
Cycle 4:	SP is decremented (represented by SP – –), because the return address is placed on the stack. The instruction i3 is fetched; however, it is not allowed to move past the decode stage.
Cycle 5:	The write address bus (EAB) is loaded with SP's contents and the on-chip return register (RTN) is loaded with the return address, a3. After the second word of the call instruction (b1) is decoded, PAB is loaded with the new value in cycle 5 (shown in row j1).
Cycles 6 and 7:	The RTN contents are written to the stack using EB in cycle 6. The instruction, j1, at address b1 is fetched in cycle 6. The two-word call instruction enters the execution stage of the pipeline in cycles 6 and 7.
Cycles 8 and 9:	These cycles are consumed by the call instruction, because the next two instructions are not allowed to complete their execution.
Cycle 10:	j1 completes execution.

Example 7–5 shows the pipeline behavior for a delayed-call instruction.

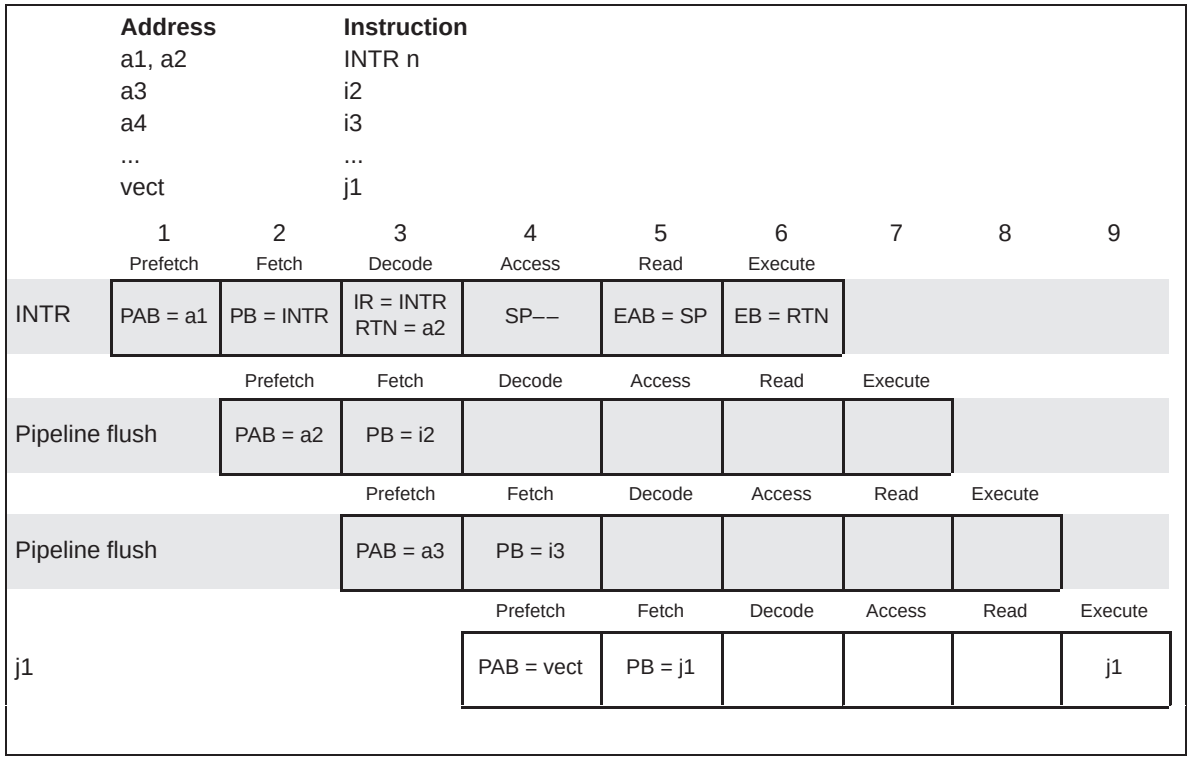
Example 7–5. Delayed-Call Instruction in the Pipeline



In this case, the pipeline behaves in the same manner as with the normal call instruction. However, in this case the following two instructions, i3 and i4, are allowed to complete their execution. Therefore, only cycles 6 and 7 are consumed by the delayed-call instruction, making it a 2-cycle instruction.

The INTR instruction behaves like a CALL instruction. However, because INTR is a 1-word instruction, it can compute the vector table address and prefetch it one cycle earlier. As shown in Example 7–6, INTR takes only three cycles to execute.

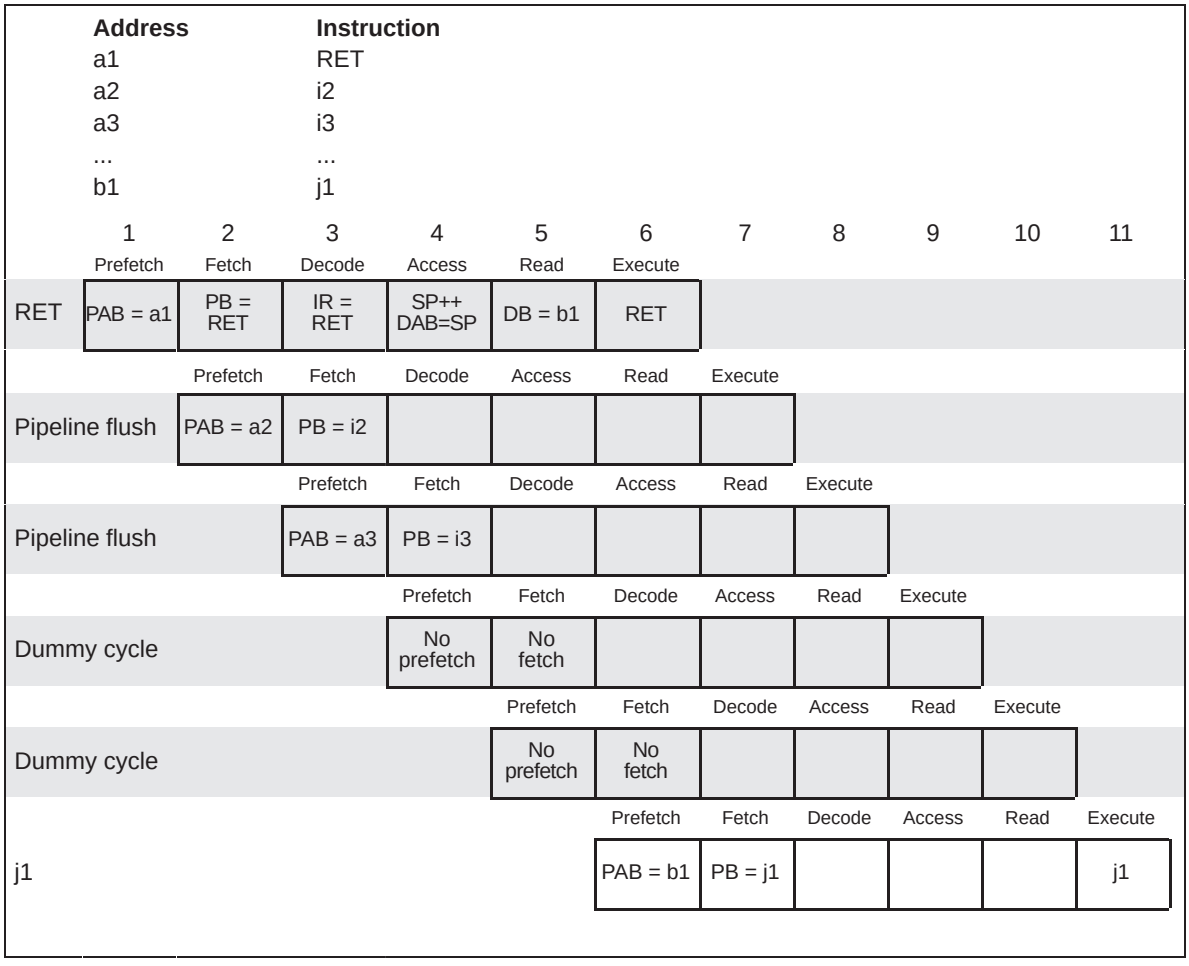
Example 7–6. INTR Instruction in the Pipeline



7.1.3 Return Instructions in the Pipeline

Because a return is a single-word instruction, you would expect it to take at least one cycle to completely execute. In reality, a standard return instruction takes five cycles to execute. Example 7–7 shows the pipeline's behavior during the execution of a return instruction.

Example 7–7. Return Instruction in the Pipeline



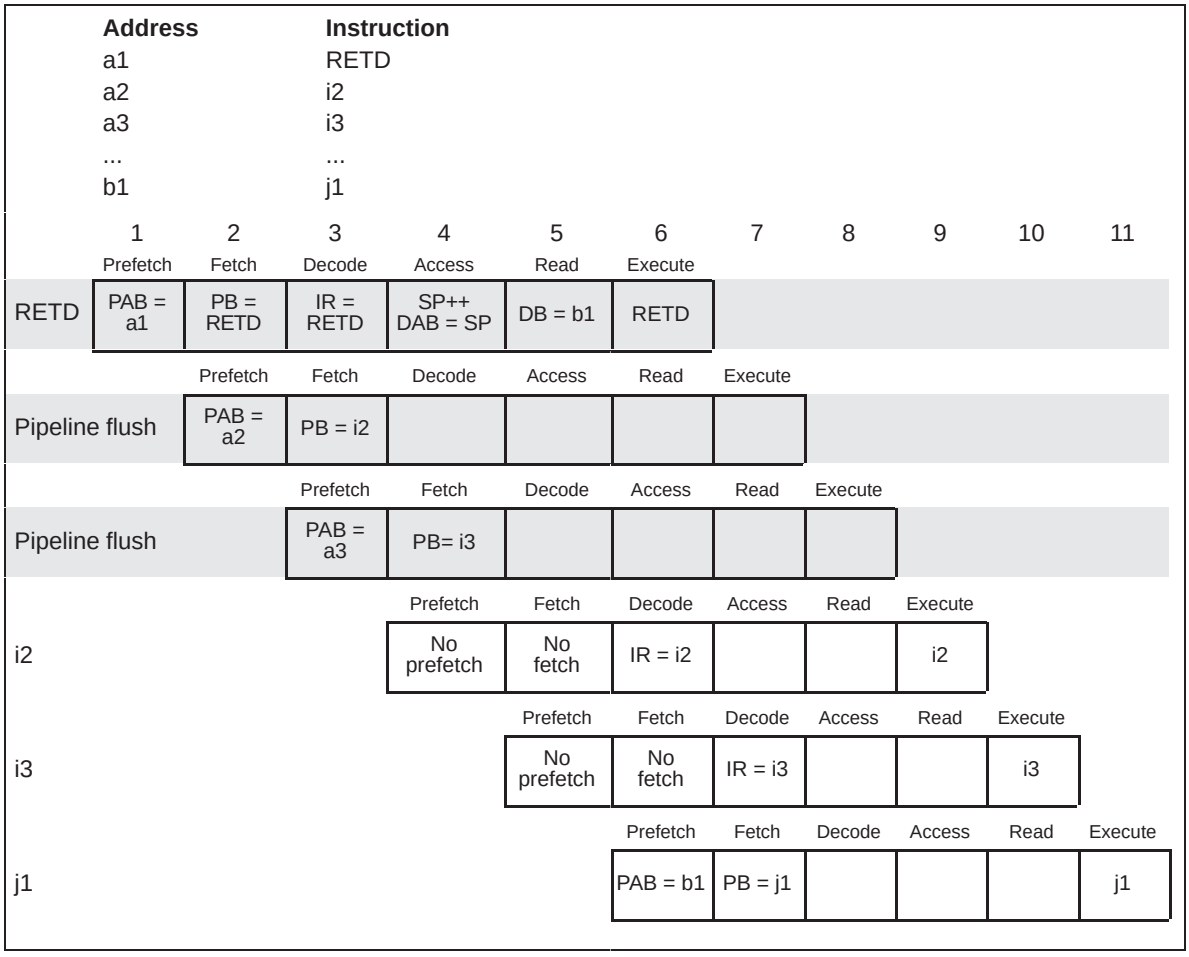
In Example 7–7, the following events occur:

Cycle 1:	The PAB is loaded with the address of the return instruction.
Cycle 2:	The return instruction opcode is fetched.
Cycles 3 and 4:	Two more instructions, i2 and i3, are fetched. Although these two instructions are fetched by the device, they are not allowed to move past the decode stage and are discarded. In cycle 4, SP is incremented (represented by SP ++) and DAB is loaded with the contents of SP in order to read the return address from the stack.
Cycle 5:	The top of the stack is read using DB.
Cycle 6:	The return instruction enters the execution stage of the pipeline. The address fetched from the stack is loaded onto PAB. This allows for fetching the next instruction, j1, from the return address.
Cycles 7 and 8:	These cycles are consumed by the return instruction, because the next two instructions, i3 and i4, do not complete their execution.
Cycles 9 and 10:	Because no instructions were fetched in cycles 4 and 5, cycles 9 and 10 are dummy cycles.
Cycle 11	j1 completes execution.

Example 7–8 shows the pipeline's behavior during the execution of a delayed-return instruction.

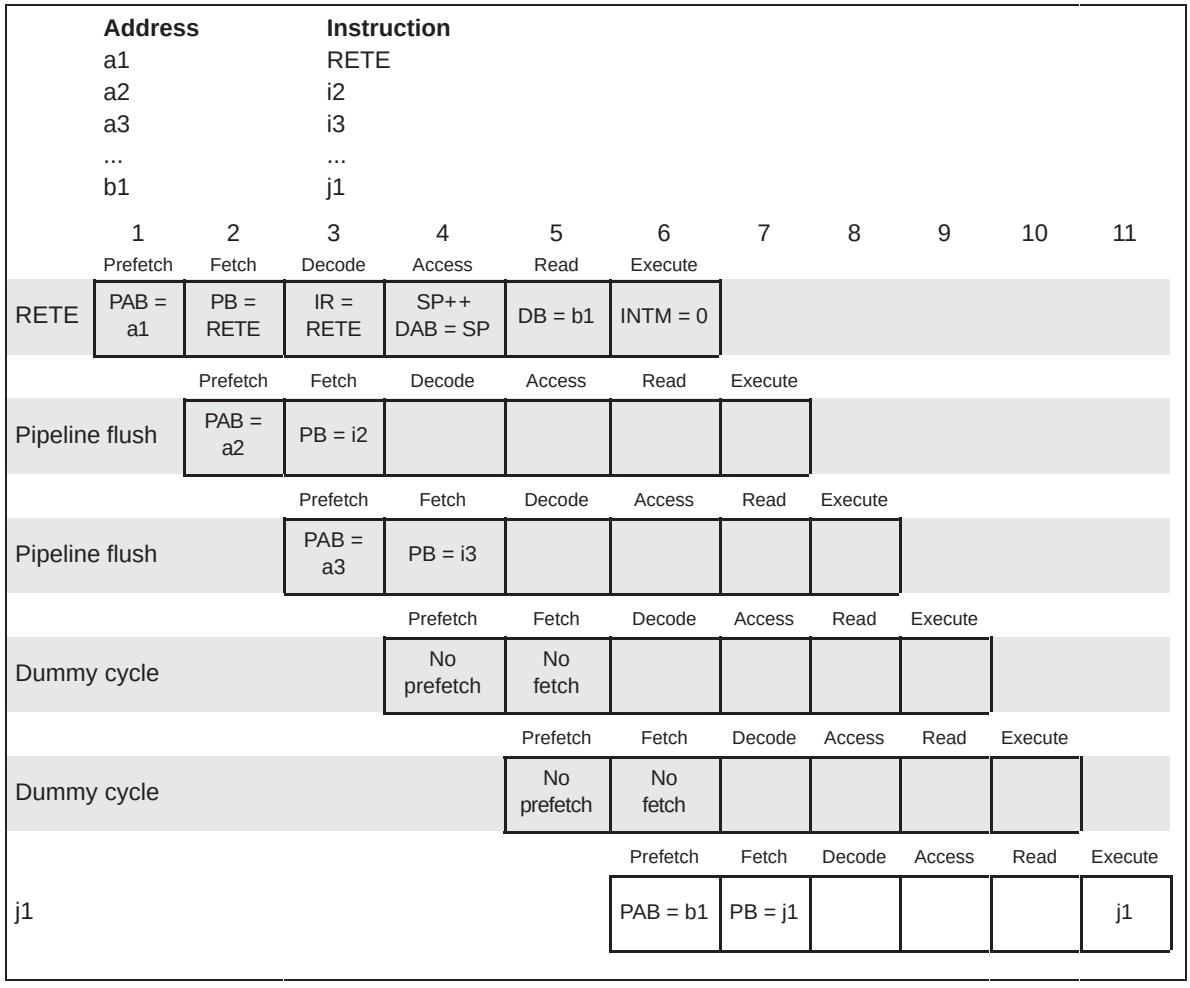
In a delayed-return instruction, the '54x pipeline behaves in the same way as with the normal return instruction. However, the following two instructions, i3 and i4, are allowed to complete their execution, so only cycles 6, 7, and 8 are consumed by the delayed-return instruction, making it a 3-cycle instruction as shown in Example 7–8.

Example 7–8. Delayed-Return Instruction in the Pipeline

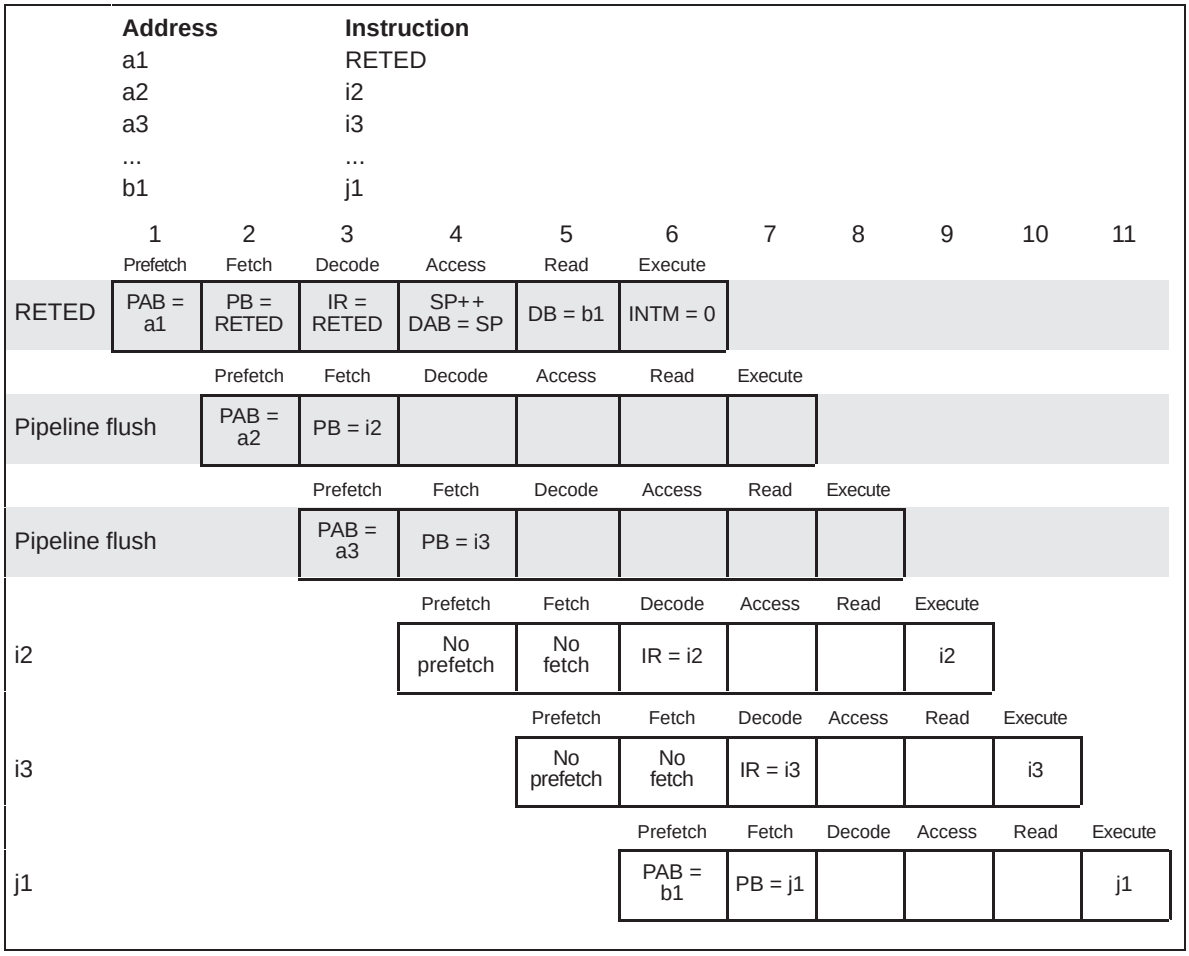


Example 7–9 and Example 7–10 show the pipeline behavior for a return-with-interrupt-enable (RETE) instruction and a delayed return-with-interrupt-enable (RETED) instruction, respectively. The pipeline behavior for these instructions is similar to that of the standard return and delayed-return instructions, respectively, and these instructions also take same number of cycles to execute. The difference is that these two instructions enable interrupts globally by resetting the INTM bit during the execute stage of the pipeline.

Example 7–9. Return-With-Interrupt-Enable Instruction in the Pipeline



Example 7–10. Delayed Return-With-Interrupt-Enable Instruction in the Pipeline



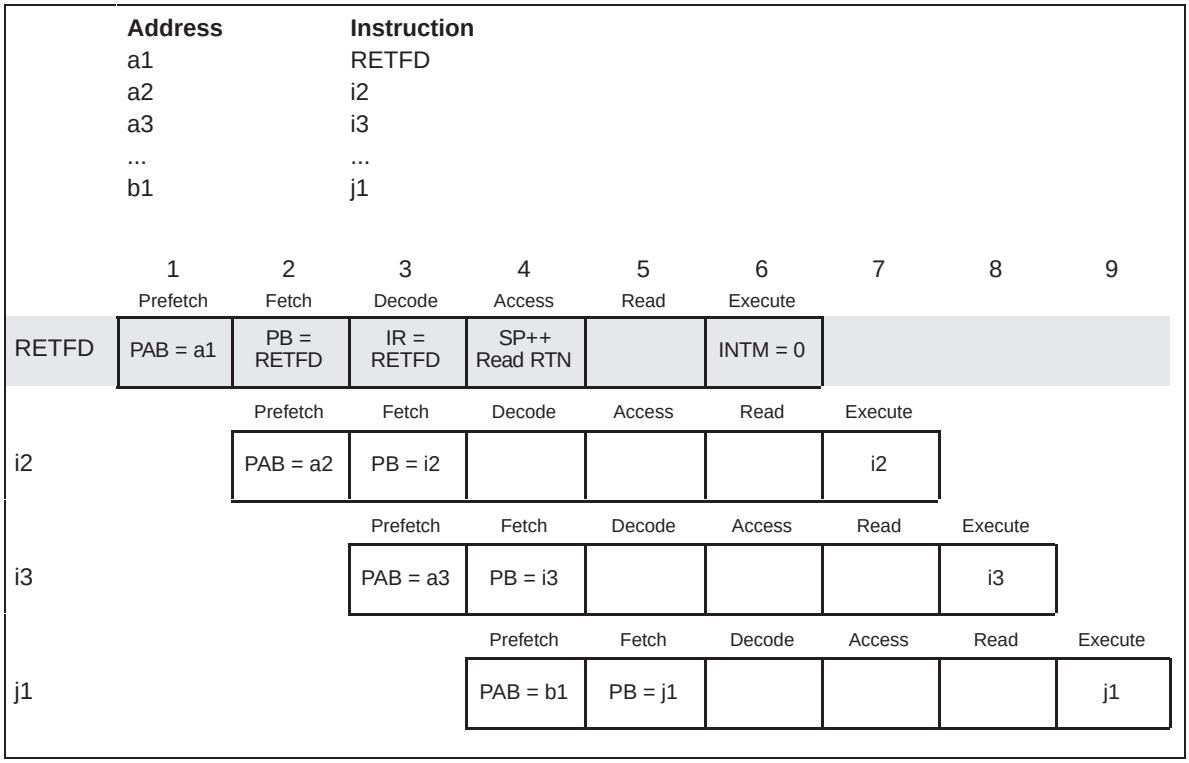
Example 7–11 and Example 7–12 show pipeline behavior for a return-fast (RETF) instruction and for a delayed return-fast (RETFD) instruction, respectively. The RETF instruction, unlike the RETE instruction, does not read the return address from the stack. Instead, it reads it from the RTN register. This allows the instruction to load PAB with the return address two cycles earlier than a RETE instruction can. As shown in the examples, the RETF instruction takes only three cycles to execute; the delayed version of the instruction, RETFD, executes in one cycle.

Example 7–11. Return-Fast Instruction in the Pipeline

	Address	**Instruction**
	a1	RETF
	a2	i2
	a3	i3
	...	...
	b1	j1

	1 Prefetch	2 Fetch	3 Decode	4 Access	5 Read	6 Execute	7	8	9
RETF	PAB = a1	PB = RETF	IR = RETF	SP++ Read RTN		INTM = 0			
		Prefetch	Fetch	Decode	Access	Read	Execute		
Pipeline flush		PAB = a2	PB = i2						
			Prefetch	Fetch	Decode	Access	Read	Execute	
Pipeline flush			PAB = a3	PB = i3					
				Prefetch	Fetch	Decode	Access	Read	Execute
j1				PAB = b1	PB = j1				j1

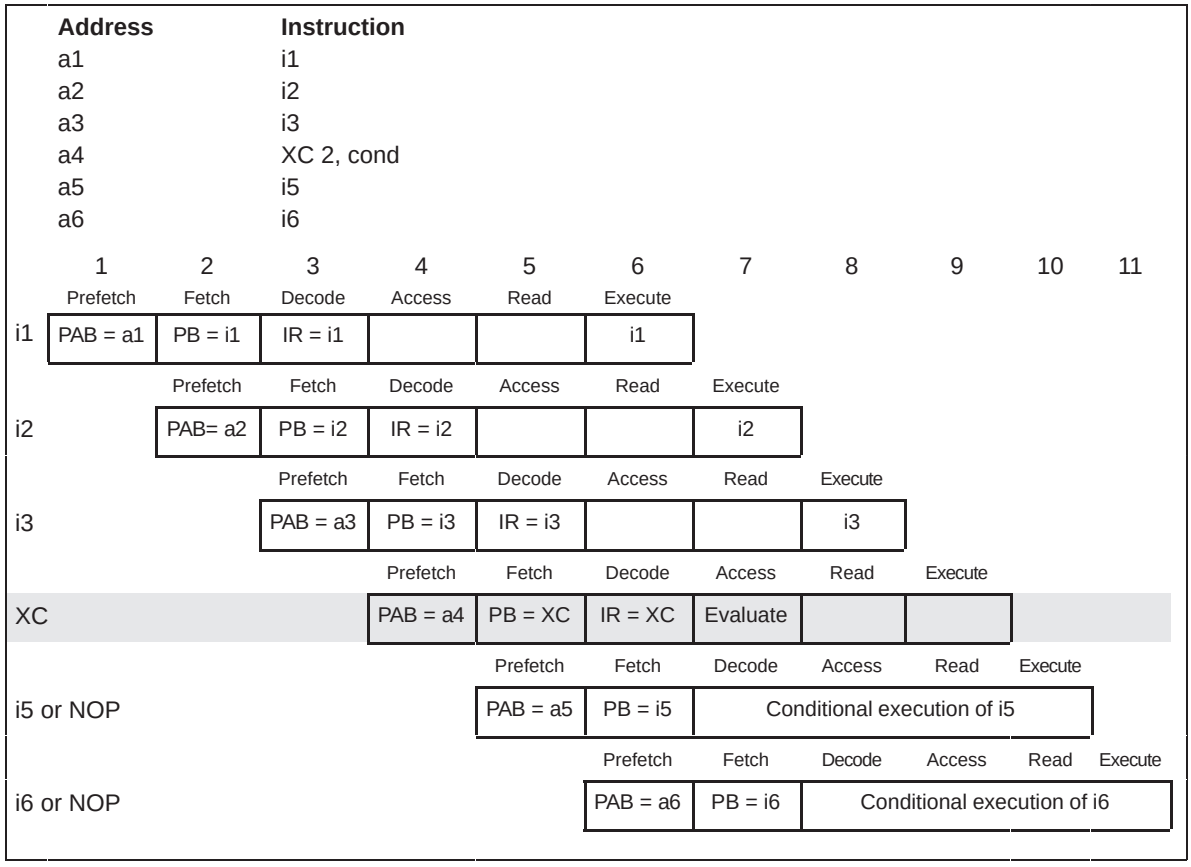
Example 7–12. Delayed Return-Fast Instruction in the Pipeline



7.1.4 Conditional Execute Instructions in the Pipeline

Because XC is single-word instruction, it takes at least one instruction cycle to completely execute. Example 7–13 shows pipeline behavior during the execution of XC.

Example 7–13. XC Instruction in the Pipeline



In Example 7–13, the following events occur:

Cycle 4: The PAB is loaded with the address of the XC instruction.

Cycle 5: The XC instruction opcode is fetched.

Cycle 7: When the XC instruction moves into the access stage of the pipeline in cycle 7, any conditions specified by the XC instruction are evaluated. If the tested conditions are true, the next two instructions, i5 and i6, are decoded and allowed to execute. However, if the tested conditions are false, i5 and i6 are not decoded.

To execute XC in one cycle, the CPU evaluates test conditions in the access stage of the pipeline. This means that the two 1-word instructions (or one 2-word instruction) immediately prior to the XC instruction will not have completely executed before the conditions are tested. Because the condition codes are affected only by instructions in the execute stage, those two instructions have no effect on the operation of XC.

7.1.5 Conditional-Call and Conditional-Branch Instructions in the Pipeline

Because a call instruction consists of two instruction words, you would expect it to take at least two cycles to execute completely. A standard conditional-call instruction actually takes either five cycles to execute if the call is taken or three cycles to execute if the call is not taken.

Example 7–14 shows pipeline behavior during the execution of a conditional-call instruction (CC).

A conditional-call instruction is similar in its pipeline behavior to an unconditional-call instruction. The only exception is that the test conditions for a conditional-call instruction are evaluated in the execute stage of the pipeline. As shown in Example 7–14, when the test conditions are evaluated in cycle 7, the previous instruction, i1, has completely executed. Furthermore, the next two instructions after CC, i4 and i5, are also fetched. If the test conditions are evaluated to be false, these two instructions proceed through the pipeline. Otherwise, they are discarded. The instruction prefetch in cycle 7 is also dependent on the evaluated conditions. If the conditions are true, PAB is loaded with the call address (b1); otherwise, it is loaded with the next incremental address (a6).

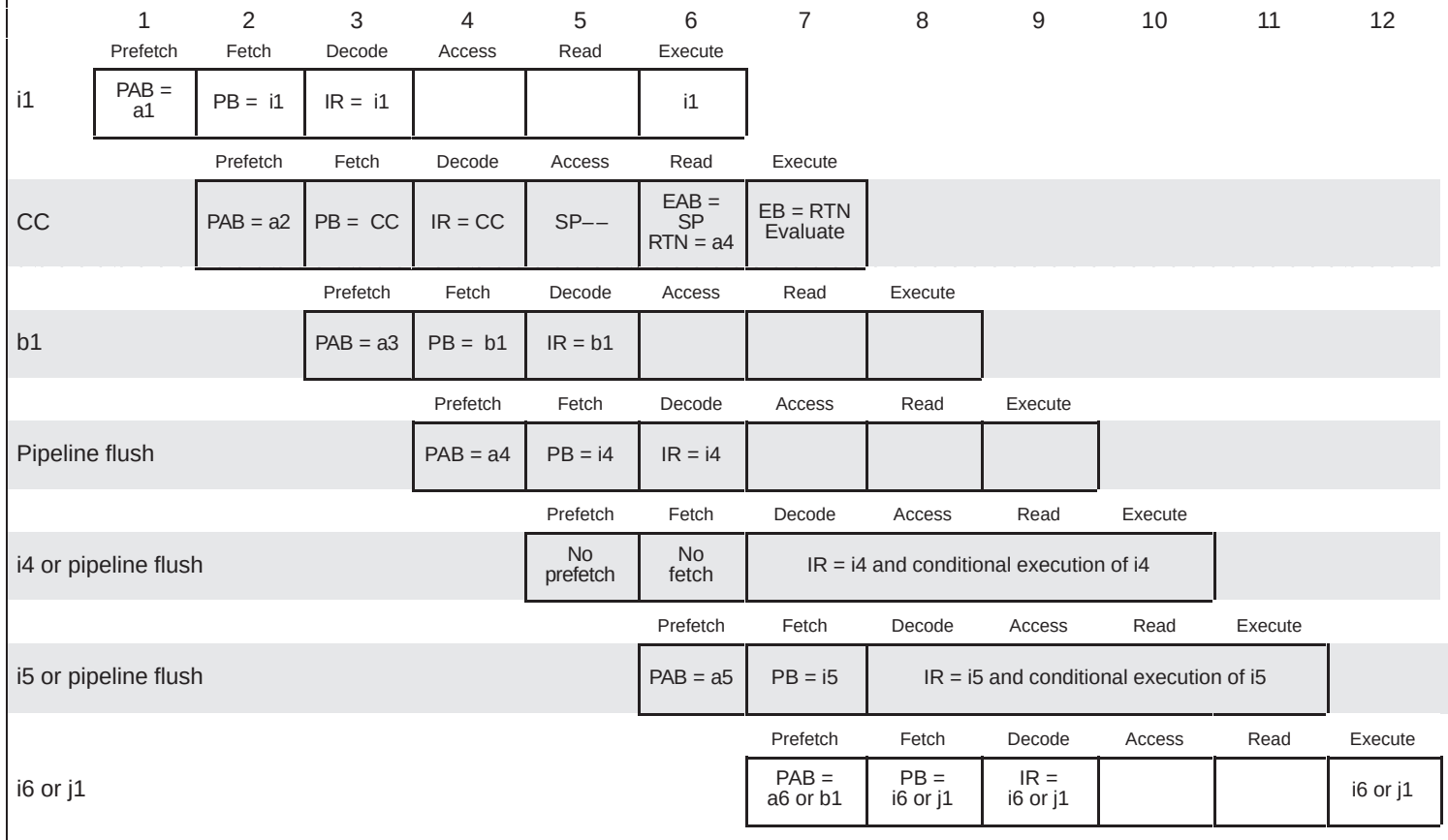
If the evaluated conditions are true, i4 and i5 do not execute in cycles 10 and 11. In this case, the CC instruction becomes a 5-cycle instruction. However, if the evaluated conditions are false, i4 and i5 execute, making CC a 3-cycle instruction.

Example 7–15 shows pipeline behavior during the execution of a delayed conditional-call (CCD) instruction.

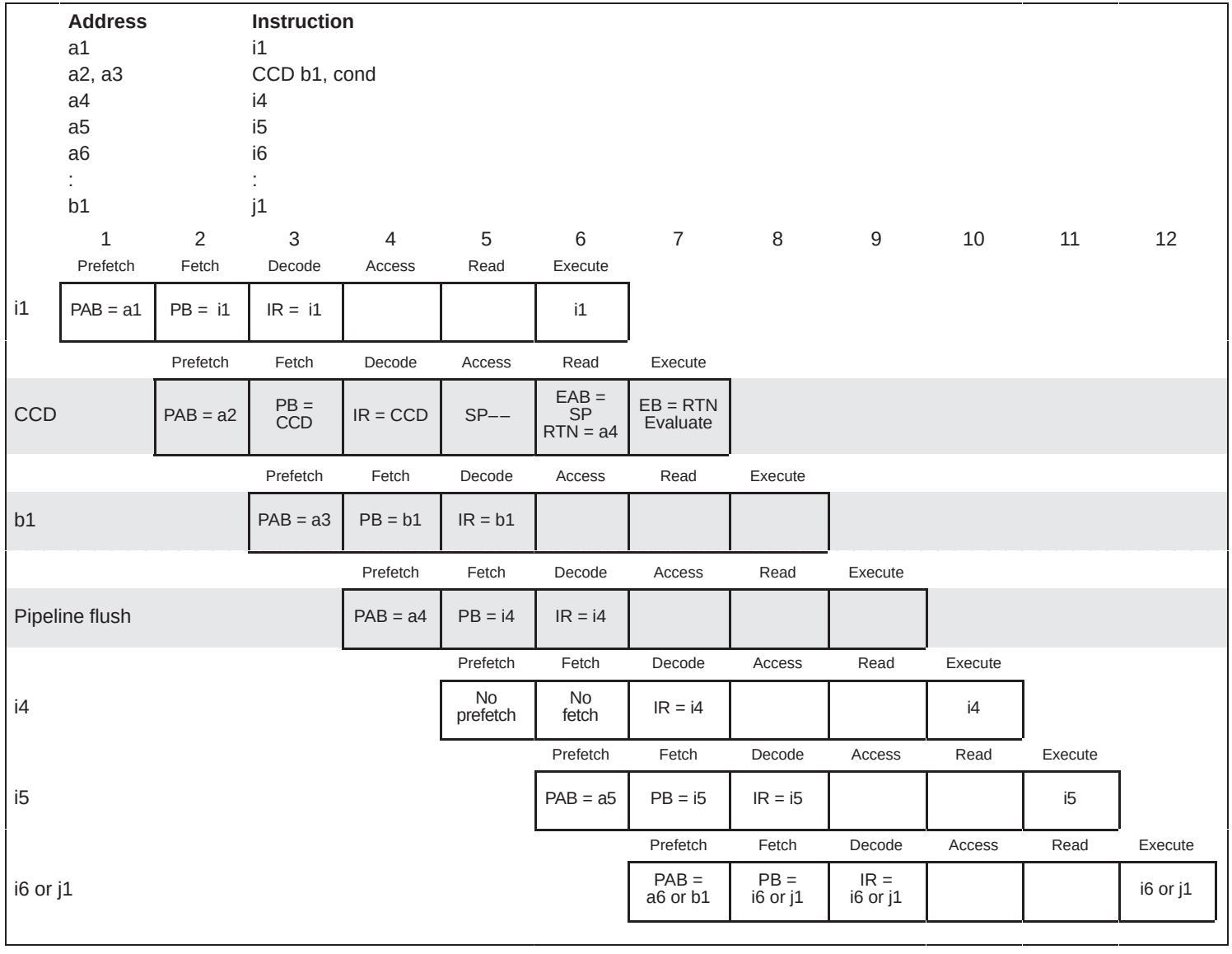
The pipeline behaves in the same manner as it does for the CC instruction. However, the following two instructions, i3 and i4, are allowed to complete their execution regardless of whether the tested conditions are true or not. Only cycles 7, 8, and 9 are consumed by the CCD instruction, making it a 3-cycle instruction.

Example 7–14. CC Instruction in the Pipeline

Address	Instruction
a1	i1
a2,a3	CC b1, cond
a4	i4
a5	i5
a6	i6
...	...
b1	j1



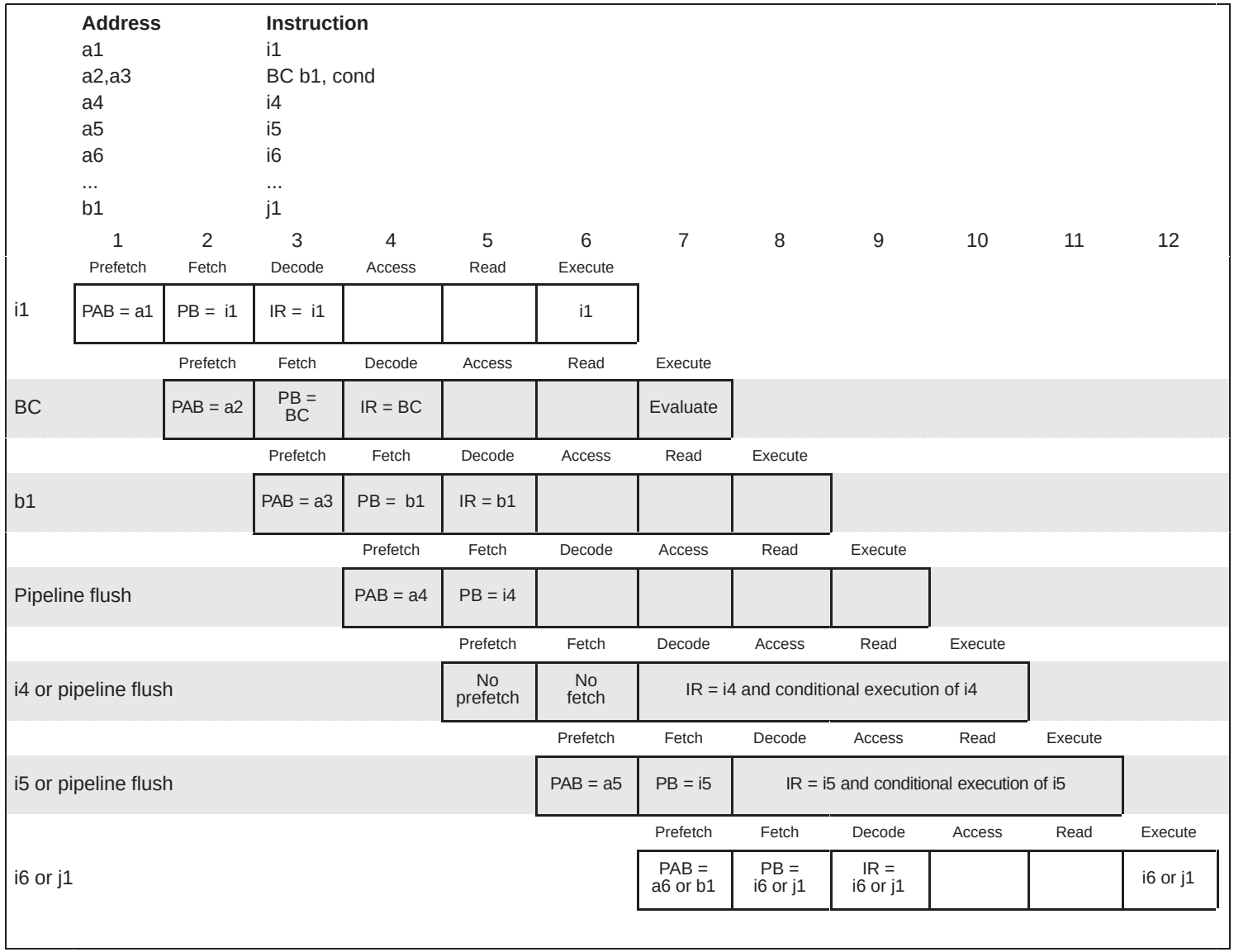
Example 7–15. CCD Instruction in the Pipeline



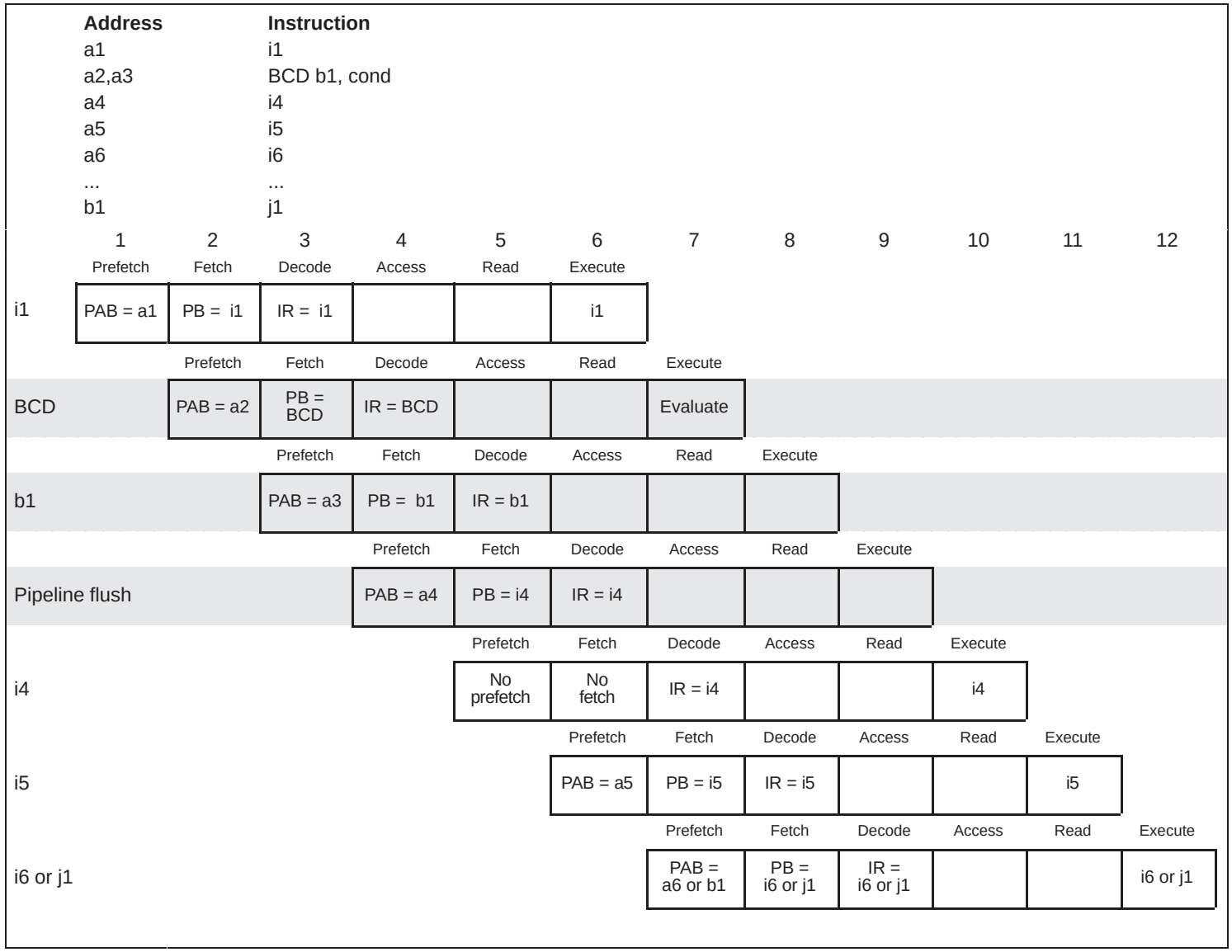
Example 7–16 and Example 7–17 show the pipeline's behavior during the execution of a conditional branch (BC) instruction and a delayed conditional branch (BCD) instruction.

The behavior of the conditional branch (BC) and the delayed conditional branch (BCD) instructions in the pipeline is similar to that of the CC and CCD instructions, respectively. The difference is that no return address is written to the stack in this case. As shown in Example 7–16, a BC instruction takes either three or five cycles to execute, depending on whether or not the branch is taken. A BCD instruction executes in three cycles.

Example 7–16. BC Instruction in the Pipeline



Example 7–17. BCD Instruction in the Pipeline



7.2 Interrupts and the Pipeline

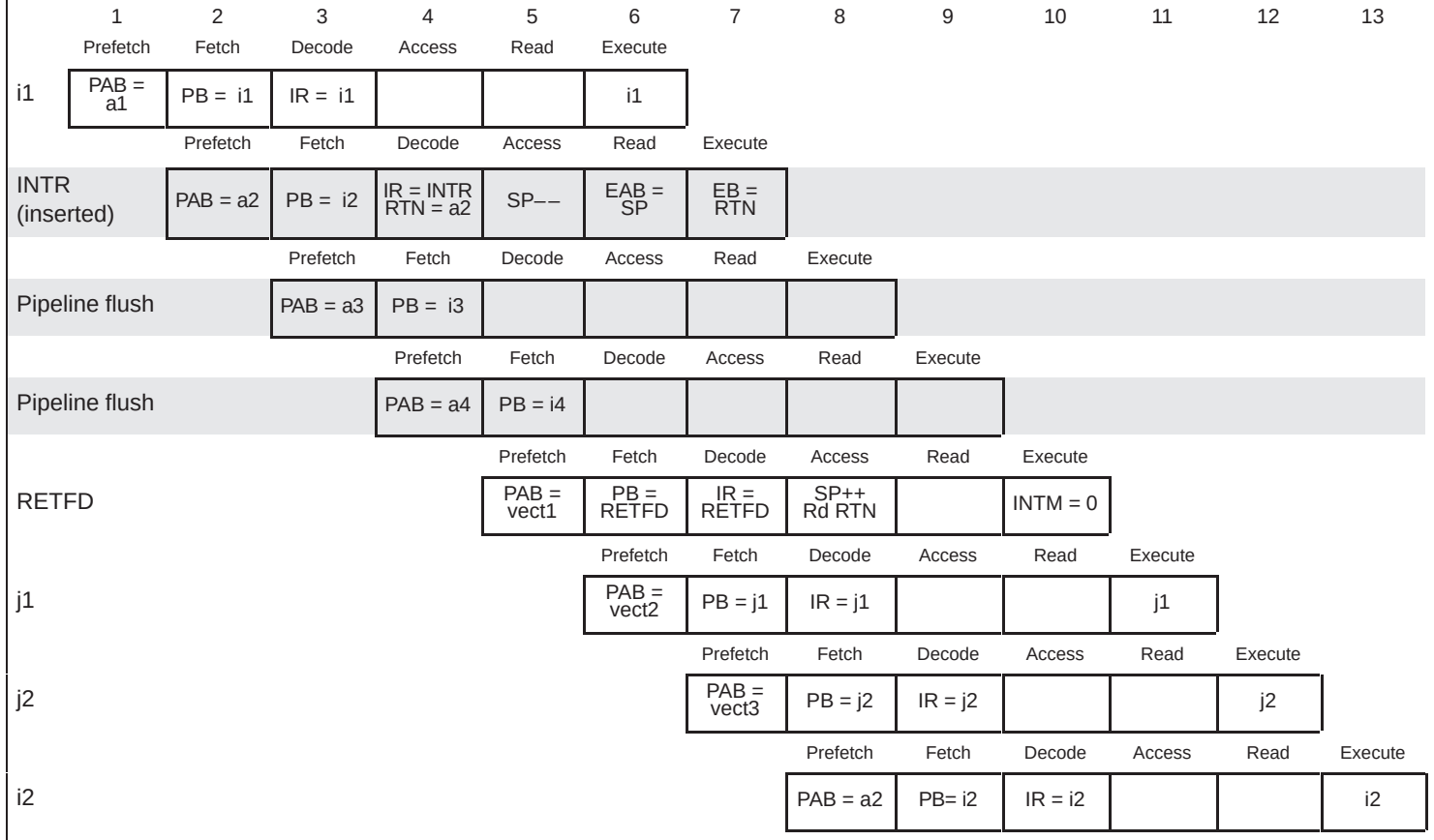
Example 7–18 shows the pipeline behavior when an interrupt is taken.

As shown in Example 7–18, if an interrupt is serviced at the end of cycle 3, an INTR instruction is automatically placed into the decode stage of the pipeline during the next cycle (4). The instruction, i2, is not decoded because the INTR instruction is placed into the pipeline at that stage. During the next three cycles, the instructions that have already been decoded are executed. Cycles 7, 8, and 9 are taken by the INTR instruction. The first instruction in the ISR, RETFD, is executed in cycle 10. Cycles 11 and 12 are consumed by the two 1-word instructions that constitute the two delay slots of the RETFD instruction. In the following cycle, instruction i2 is executed, completing the return from the ISR.

As shown in the figure, interrupt overhead (the number of cycles required to branch to an ISR) is only three cycles. The return from the interrupt takes only one cycle because RETFD is a single-cycle instruction. Because only four words are reserved for each interrupt in the interrupt vector table, if an ISR requires more than four instruction words, it must be located elsewhere. In this case, a branch type instruction in the interrupt vector table. This would result in a slightly higher interrupt overhead.

Example 7–18. Interrupt Response by the Pipeline

```
Address         Instruction
a1              i1          ; Interrupt taken after executing this instruction
a2              i2
a3              i3
a4              i4
:               :
vect1           RETFD       ; First instruction in the interrupt vector table
vect2           j1
vect3           j2
```



7.3 Dual-Access Memory and the Pipeline

The ’54x features on-chip memory that supports two accesses in a single cycle. This dual-access memory is organized as several independent memory blocks. Simultaneous accesses to different blocks are supported with no conflicts: while one instruction in the pipeline accesses one block, another instruction at the same stage in the pipeline can access a different block without conflict. Furthermore, each memory block supports two accesses in a single cycle: two instructions, each in different stages of the pipeline, can access the same block simultaneously. However, a conflict can occur when two simultaneous accesses are performed on the same block. The ’54x CPU resolves these conflicts automatically; this is discussed later in this section. Table 7–1 shows the block size and number of blocks for each ’54x device. For more information about DARAM organization, see subsection 3.3.2, *On-Chip RAM Organization*, on page 3-15.

Table 7–1. DARAM Blocks

Device	Block Size†	Number of Blocks
’541	1K words	5
’542	2K words	5
’543	2K words	5
’545	2K words	3
’546	2K words	3
’548	2K words	4
’549	2K words	4

† Note that the first block is slightly smaller due to the memory-mapped registers and the scratch-pad RAM.

Each dual-access memory block supports two accesses in one cycle by performing one access in the first half-cycle and the other in the next half-cycle. Table 7–2 lists the accesses performed in each half-cycle, and Figure 7–3 shows how the different types are performed. Address bus loads are omitted from the diagram for simplicity.

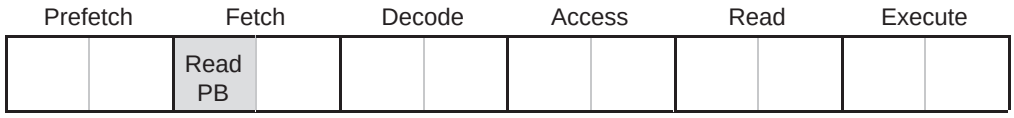
Table 7–2. Accessing DARAM Blocks

This type of access ...	Is performed in the ...
Instruction fetch using PAB/PB	First half-cycle
First data operand read using DAB/DB	First half-cycle

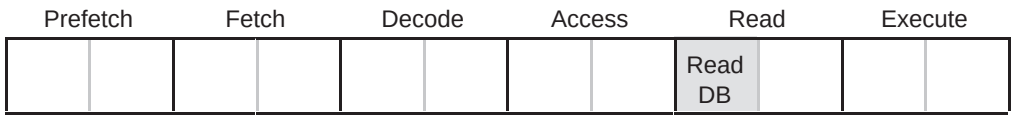
Second data operand read using CAB/CB	Second half-cycle
Data operand write using EAB/EB	Second half-cycle

Figure 7–3. Half-Cycle Accesses to Dual-Access Memory

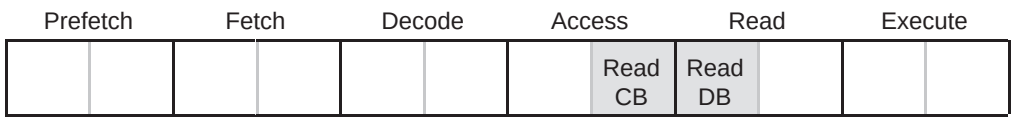
(a) Instruction word fetch



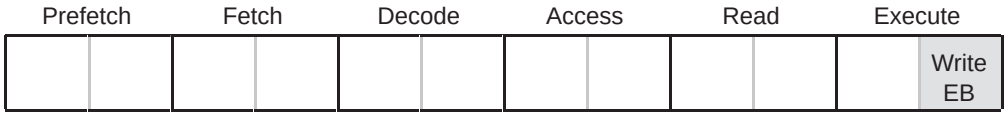
(b) Instruction performing single-operand read



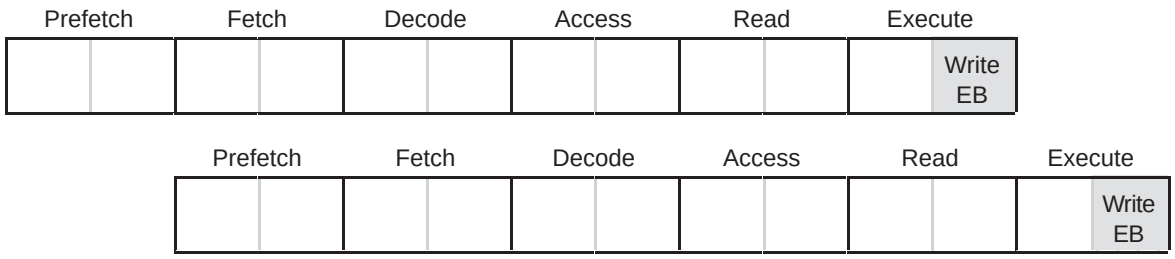
(c) Instruction performing dual-operand read



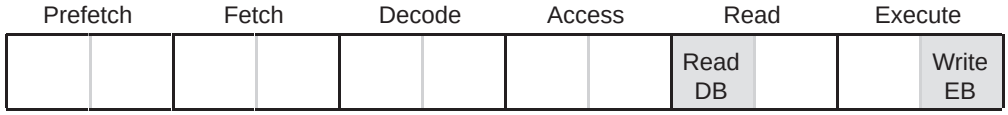
(d) Instruction performing single-operand write



(e) Instruction performing dual-operand write (two cycles)



(f) Instruction performing operand read and operand write

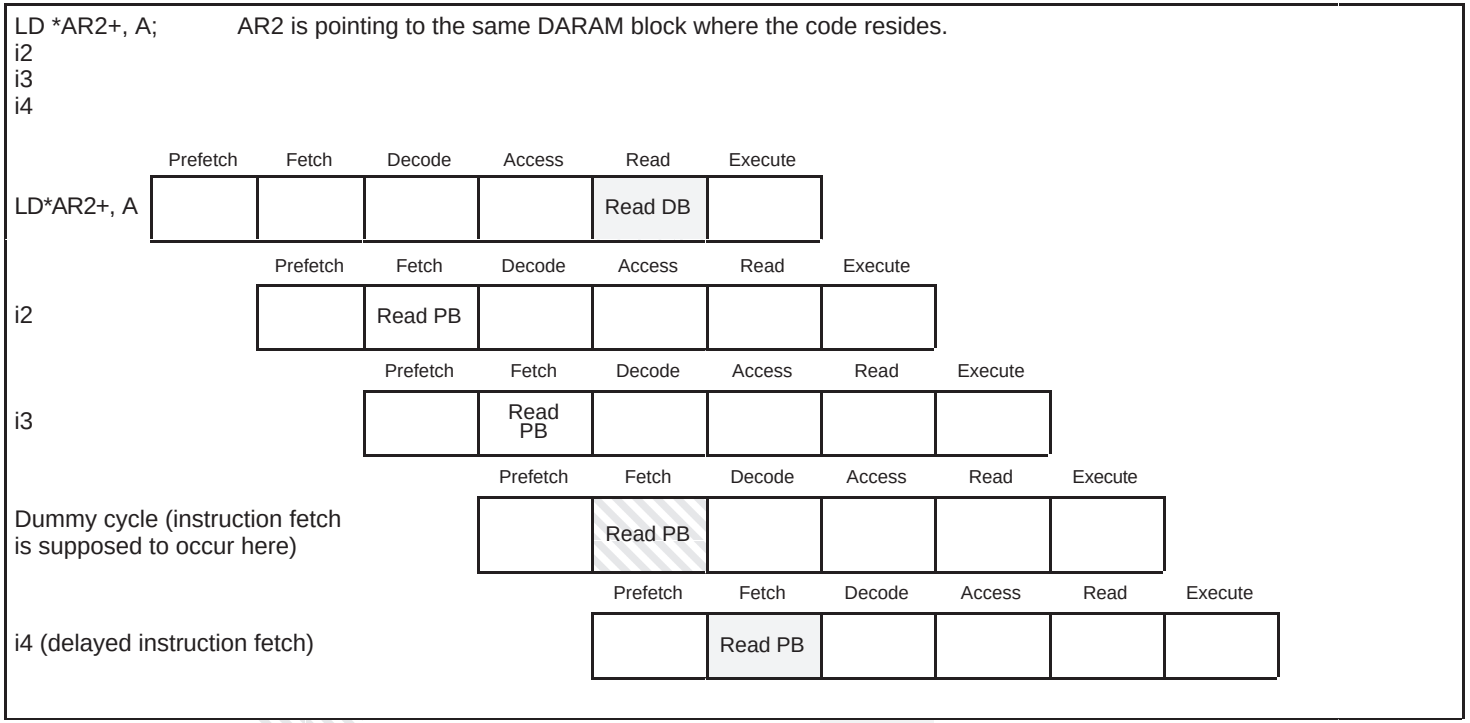


Because two types of access are scheduled and only one access is performed in each half-cycle, conflicts can occur. These conflicts are automatically resolved by the CPU either by rearranging the order of accesses or by delaying an access by one cycle. The following subsections describe these resolved memory access conflicts. Keep in mind that these conflicts appear only if all accesses are being performed on the *same* dual-access memory block.

7.3.1 Resolved Conflict Between Instruction Fetch and Operand Read

If a dual-access memory block is mapped in both program and data spaces, an instruction fetch will conflict with a data operand read access if they are performed on the same memory block. The ’54x resolves this conflict by delaying the instruction fetch by one cycle, as shown in Example 7–19. In the figure, it is assumed that instructions i2 and i3 do not access the dual-access memory block where the code resides.

Example 7–19. Instruction Fetch and Operand Read



Legend:

Where instruction fetch is supposed to occur

Where a memory access actually occurs

7.3.2 Resolved Conflict Between Operand Write and Dual-Operand Read

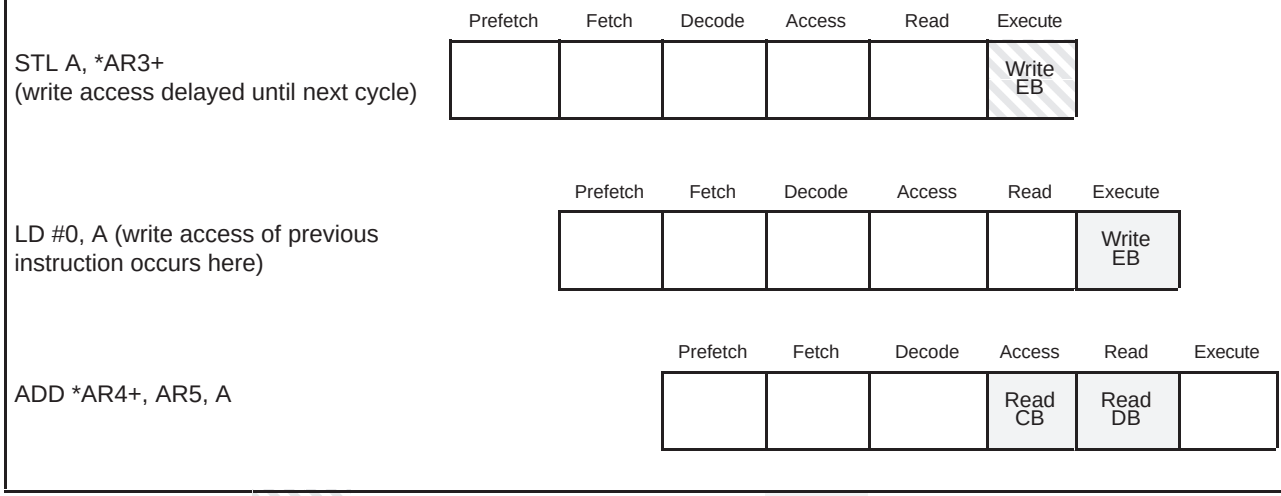
Another conflict arises if a single-operand write instruction is followed by an instruction that does not perform a write access and this instruction is followed by a dual-operand read instruction. This is shown in Example 7–20, in which AR3 and AR5 point to the same dual-access memory block.

There is a conflict between the operand write access (EB bus) and the second data read access (CB bus). This conflict is resolved automatically by delaying the write access by one cycle. The actual execution time of these instructions does not increase, because the delayed write access is performed while the second instruction is in the execute stage.

If any read access (via DB or CB) is from the *same* memory location in on-chip memory where the write access should occur, the CPU bypasses reading the actual memory location; instead, it reads the data directly from an internal bus. This allows the pipeline to perform a write access in a later pipeline stage than that in which the next instruction reads from the same memory location.

Example 7–20. Operand Write and Dual-Operand Read Conflict

```
STL A, *AR3+
LD #0, A
ADD *AR4+, *AR5+, A          ; AR3 and AR5 both point to the same dual-access memory block.
```



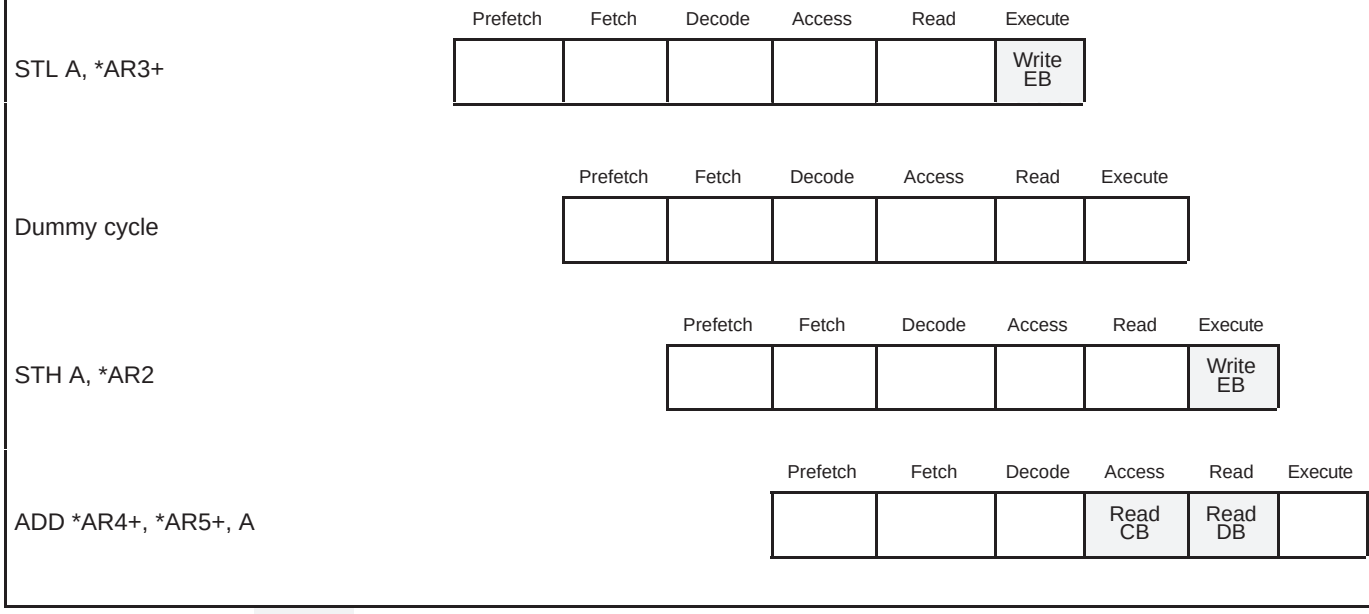
Legend: Where an access is supposed to occur — Where an access actually occurs

7.3.3 Resolved Conflict Among Operand Write, Operand Write, and Dual-Operand Read

If the second instruction in the case described above is an operand-write type instruction, then the write access requested by the first instruction cannot be moved to the next cycle. The CPU resolves the conflict by inserting a dummy cycle after the first instruction. This is illustrated in Example 7–21, in which AR3 and AR5 point to the same dual-access memory block.

Example 7–21. Operand Write and Operand Read Conflict

```
STL A, *AR3+
STH A, *AR2
ADD *AR4+, *AR5+, A          ; AR3 and AR5 both point to the same dual-access memory block
```



Legend: Where read or write occurs

7.4 Single-Access Memory and the Pipeline

The ’54x also features on-chip single-access memory that supports one access per cycle to each memory block. There are two different types of single-access memory that are available on ’54x devices:

- ❏ Single access read-write memory (SARAM)
- ❏ Single-access read-only memory (ROM or DROM)

Both types of single-access memory behave similarly in terms of pipelined accesses, with the exception that ROM and DROM cannot be written to. These memory blocks are contiguous in memory with the first block beginning at the start address of SARAM or ROM. For more information about memory blocking, see subsection 3.2.2, *On-Chip ROM Organization*, on page 3-11, and subsection 3.3.2, *On-Chip RAM Organization*, on page 3-15.

Simultaneous accesses with no conflicts are supported by single-access memory as long as the access are to different memory blocks; while one instruction in a pipeline stage accesses one memory block, another instruction can access a different memory block in the same cycle without any conflict.

A conflict can occur when two simultaneous accesses are performed on the same memory block. In case of such a conflict, only one access is performed in that cycle and the second access is delayed until the following cycle. This results in a one-cycle pipeline latency.

A pipeline conflict due to single access memory may occur in several different situations.

- ❏ Dual-Operand instructions. Many instructions have two memory operands to read or write data. If both operands are pointing to the same single-access memory block, a pipeline conflict occurs. The CPU automatically delays the execution of that instruction by one cycle to resolve the conflict. For example:

```
MAC *AR2+, *AR3+%,A,B ; This instruction will take two
                      ; cycles if both operands are in
                      ; same SARAM or DROM block.
```

- ❏ 32-bit operand instructions. Instructions that read 32-bit memory operands still take only one cycle to execute, even if their operand is in single-access memory. Single-access memory blocks are designed to allow a 32-bit read to occur in one cycle. Instructions that write 32-bit operands take two cycles to execute.

```
DLD *AR2, A   ; This instruction only takes 1 cycle even
              ; if the operand is in single–access memory.
```

- Read-write conflict. If an instruction that writes to a single-access memory block is followed by an instruction that reads from the same single-access memory block, a conflict occurs because both instructions try to access the same memory block simultaneously. In this case, the read access is delayed automatically by one cycle. For example:

```
STL A, *AR1+  ; AR1 and AR3 points at the same SARAM
              ; block.
LD  *AR3, B   ; This instruction takes 1 additional
              ; cycle due to a memory access conflict.
```

  On the other hand, a dual-operand instruction that has a read operand and a write operand does not cause this conflict because the two accesses are done in two different pipeline stages. For example:

```
ST A, *AR2+     ; This instruction does not take any
||ADD *AR3+, B  ; extra cycles, even if AR2/AR3 point
                ; at the same single access memory
                ; block.
```

- Code-data conflict. Another type of memory access conflict can occur when SARAM or ROM is mapped in both program and data spaces. In this case, if instructions are fetched from a memory block and data accesses (read or write) are also performed on the same memory block, the instruction fetch is delayed by one cycle. For example:

```
LD  *AR1+, A  ; This read data access delays a
              ; subsequent instruction fetch.
STH A, *AR2   ; This write data access delays a
              ; subsequent instruction fetch
```

  This situation causes significantly higher pipeline latency than the cases described previously. This is because each time there is a read or write access to the memory block, the pipeline is stalled for one cycle. It is generally recommended that each single-access memory block be reserved for either data or program storage to avoid hits each time a data access is made to that block.

7.5 Pipeline Latencies

The ’54x pipeline allows multiple instructions to access CPU resources simultaneously. Because CPU resources are limited, conflicts can occur when one CPU resource is accessed by more than one pipeline stage. Some of these pipeline conflicts are resolved automatically by the CPU by delaying accesses. Other conflicts are unprotected and must be resolved by the programmer.

In general, unprotected conflicts are resolved by rearranging instructions or by inserting NOP instruction (no operation performed). They can also be avoided by using only instructions that do not create any pipeline conflicts or by observing necessary delays before certain registers are accessed.

7.5.1 Recommended Instructions for Accessing Memory-Mapped Registers

Unprotected pipeline conflicts can occur when any one of the following memory-mapped registers is accessed:

- Auxiliary registers (AR0 – AR7)
- Block size register (BK)
- Stack pointer (SP)
- Temporary register (T)
- Processor mode status register (PMST)
- Status registers (ST0 and ST1)
- Block-repeat counter register (BRC)
- Memory-mapped accumulator registers (AG, AH, AL, BG, BH, BL)

However, certain instructions can access these registers without causing pipeline conflicts if you observe appropriate latency cycles. Table 7–3 lists these instructions.

Table 7–3 is valid only if programmers limit themselves to those instructions that are listed in column 3 in order to perform functions listed in column 2. Otherwise, refer to the following subsections to find the latency of each individual instruction. Furthermore, this table is provided as a quick reference for pipeline latencies. It does not describe all possible pipeline latencies, nor does it provide detailed information about latencies.

Table 7–3. Recommended Instructions for Accessing Memory-Mapped Registers

Cat†	Function	Instruction(s)	Latency	Additional Restrictions
1	Writing to ARx/BK without using an accumulator	STM MVDK MVMM MVMD	ARx update: None BK update: The next word must not use circular addressing	None
2	Writing to ARx/BK using an accumulator	STLM STH STL Store type‡	The next 2 words (ARx) or 3 words (BK) must not use the same register.	The next instruction must not write to *any* ARx, BK, or SP using STM, MVDK, or MVMD.
3	Popping ARx/BK from stack	POPM	The next 1 word (ARx) or 2 words (BK) must not use the same register.	Do not precede a category 3 instruction with any category 2 or 5 instruction that writes to *any* ARx, BK, or SP.
4	Writing to SP without using an accumulator	STM MVDK MVMM MVMD	None if CPL = 0 The next 1 word must not use SP if CPL = 1.	None
5	Writing to SP using an accumulator	STLM STH STL Store type‡	The next 2 (if CPL = 0) or 3 (if CPL = 1) words must not use SP.	The next instruction must not write to ARx, BK, or SP using STM, MVDK, or MVMD.
6	Writing to T without using an accumulator	STM MVDK LD Smem,T LD Smem,T || ST	None	None
7	Writing to T using an accumulator	STLM STH STL	The next word must not use T.	None
8	Writing to BRC without using an accumulator	STM MVDK	None	None
9	Writing to BRC using an accumulator	STLM STH STL Store type‡	The next instruction must not be a RPTB[D].	None
10	Writing to ARP	LD #k, ARP	None	None

† Category
‡ Any other store-type instruction. See Table 7–5 on page 7-42 for a list of store-type instructions.

Table 7–3. Recommended Instructions for Accessing Memory-Mapped Registers (Continued)

Cat†	Function	Instruction(s)	Latency	Additional Restrictions
11	Writing to DP	LD #k, DP LD Smem, DP	None	None
12	Writing to CPL	RSBX SSBX	The next 3 words must not use direct addressing mode.	None
13	Writing to SXM	RSBX SSBX	The next word must not be affected by SXM status.	None
14	Writing to ASM	LD #k, ASM LD Smem, ASM	None	None
15	Writing to BRAF	RSBX SSBX	The next 5 words must not contain the last instruction word in the RPTB loop.	None
16	Writing BRC to memory	SRCCD	The next 2 words must not contain the last instruction word in the RPTB loop.	None
17	Writing to OVLY, MP/$\overline{MC}$, or IPTR	ANDM ORM XORM	The next 6 cycles must not include an instruction fetch from the on-chip memory's address range.	An external-bus cycle may cause additional latency.
18	Writing to DROM bit	ANDM ORM XORM	The next 3 words must not access the DROM's address range.	An external-bus cycle may cause additional latency.
19	Calculating an exponent	EXP	The next instruction must not use T.	None

† Category
‡ Any other store-type instruction. See Table 7–5 on page 7-42 for a list of store-type instructions.

Table 7–3. Recommended Instructions for Accessing Memory-Mapped Registers (Continued)

Cat†	Function	Instruction(s)	Latency	Additional Restrictions
20	Stack manipulation in compiler mode (CPL = 1)	FRAME POPM/POPD PSHM/PSHD	The next instruction must not use direct addressing mode (CPL = 1).	None
21	Reading AG, AH, AL, BG, BH, or BL as memory-mapped registers	Any instruction that can read from memory	The previous instruction must not modify accumulator A or accumulator B.	None

† Category
‡ Any other store-type instruction. See Table 7–5 on page 7-42 for a list of store-type instructions.

7.5.2 Updating ARx, BK, or SP—A Resolved Conflict

Table 7–4 lists ’54x instructions that update data-address generation logic (DAGEN) registers in the read stage of the pipeline. The DAGEN registers are the auxiliary registers (ARx), the block size register (BK), and the stack pointer (SP).

All other instructions that write to these registers perform their writes in the execute stage and are store-type instructions. They are listed in Table 7–5.

Table 7–4. Instructions That Access DAGEN Registers in the Read Stage

Instruction Type	Instructions	
Constant initialization	STM ST	#lk, MMR #lk, Smem†,‡
Move type 1	MVDD POPM POPD DELAY	Xmem, Ymem† MMR Smem†,‡ Smem†,‡
Move type 2	MVDK MVMD	Smem, dmad† MMR, dmad†

† This operand must be pointing to one of the DAGEN registers.
‡ DP must be 0 to access DAGEN registers.

Table 7–5. Store-Type Instructions

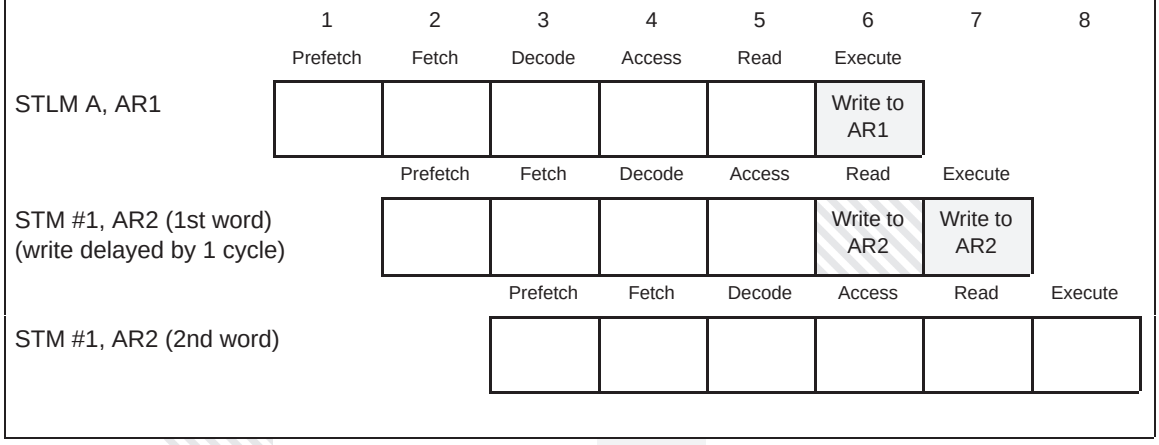
Instruction		Instruction		Instruction	
MVKD	dmad, Smem	STL	src, SHFT, Xmem	SRCCD	Xmem, cond
MVDM	dmad, MMR	STL	src, SHIFT, Smem	STRCD	Xmem, cond
MVPD	dmad, Smem	ST || ADD		CMPS	src, Smem
STH	src, Smem	ST || LD		ST	T, Smem
STH	src, ASM, Smem	ST || LT		ST	TRN, Smem
STH	src, SHFT, Xmem	ST || MAC[R]		ADDM	Smem, #lk
STH	src, SHIFT, Smem	ST || MAS[R]		ANDM	Smem, #lk
STLM	src, MMR	ST || MPY		ORM	#lk, Smem
STL	src, Smem	ST || SUB		XORM	Smem, #lk
STL	src, ASM, Smem	SACCD	src, Xmem, cond		

When a store-type instruction is immediately followed by an instruction that updates ARx, BK, or SP in the read stage, a conflict can occur, because both instructions try to access DAGEN registers. The DAGEN register set can be written to only once in a given cycle, so the CPU delays the read stage access by one cycle. This access is performed when the second instruction is in the execute stage of the pipeline. This generally does not affect the execution time of that instruction. Example 7–22, Example 7–23, and Example 7–24 show this conflict.

Example 7–22. Resolving Conflict When Updating Multiple ARxs

(a) Updating AR1 in execute stage and AR2 in read stage

```
STLM   A, AR1     ; This instruction updates AR1 in the Execute stage.
STM    #1, AR2    ; This instruction tries to update AR2 in its read stage,
                  ; creating a conflict with the previous instruction.
                  ; The AR2 update is delayed by one cycle.
```

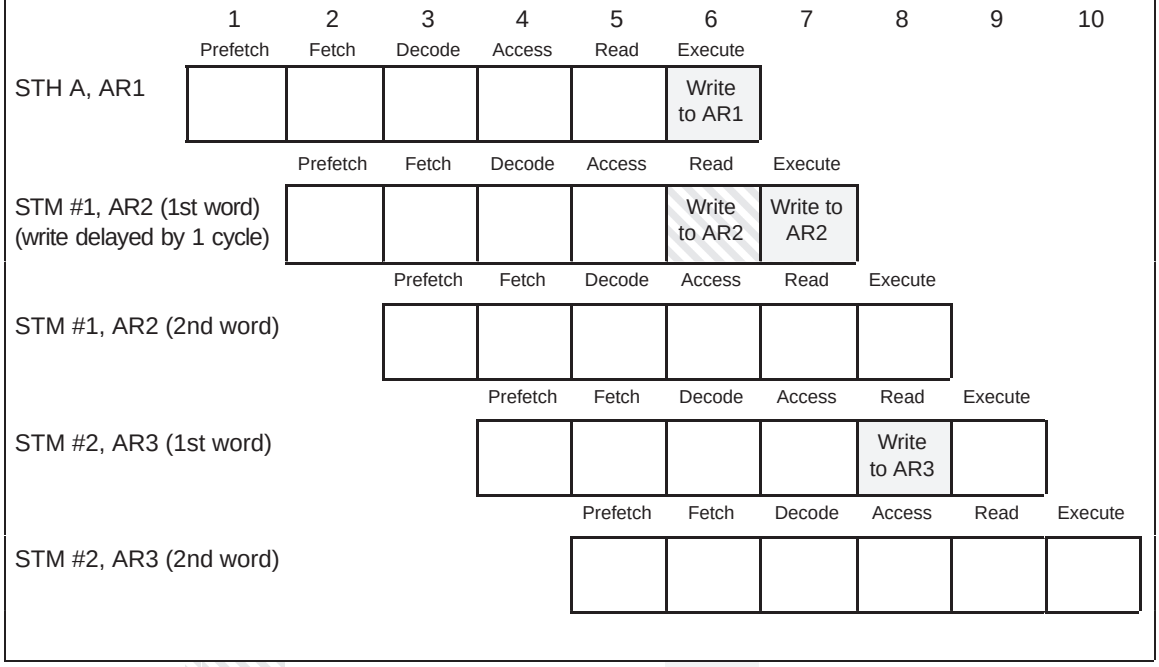


Legend: Where a write conflict occurs — Where the write actually occurs

Example 7–22. Resolving Conflict When Updating Multiple ARxs (Continued)

(b) Updating AR1 in execute stage, AR2 in read stage, and AR3 in read stage

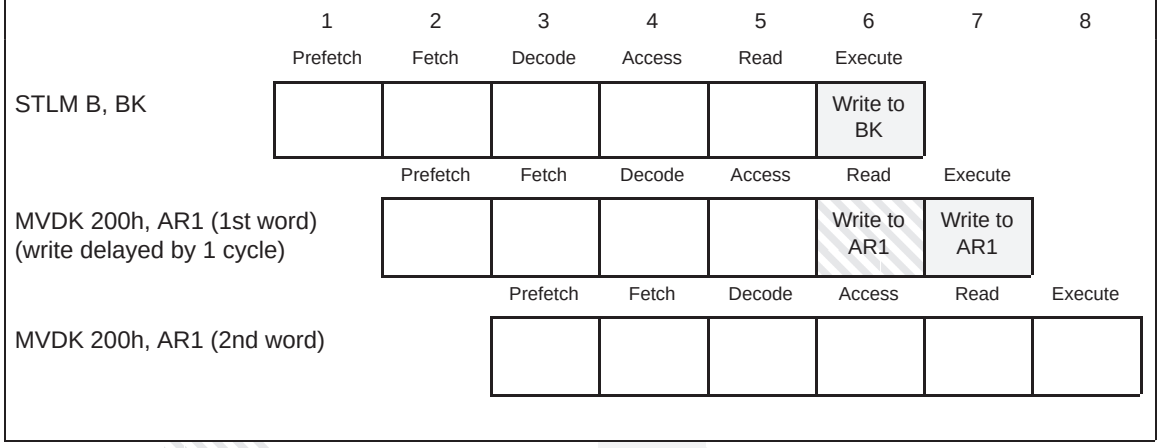
```
STH    A, AR1    ; This instruction updates AR1 in the execute stage.
;
STM    #1, AR2   ; This instruction tries to update AR2 in its read stage,
                 ; causing a conflict. The update is delayed by one cycle.
;
STM    #2, AR3   ; This instruction updates AR3 in its read stage. It
                 : creates no conflict since the previous instruction was
                 ; a two-word instruction.
```



Legend: Where a write conflict occurs Where the write actually occurs

Example 7–23. Resolving Conflict When Updating ARx and BK

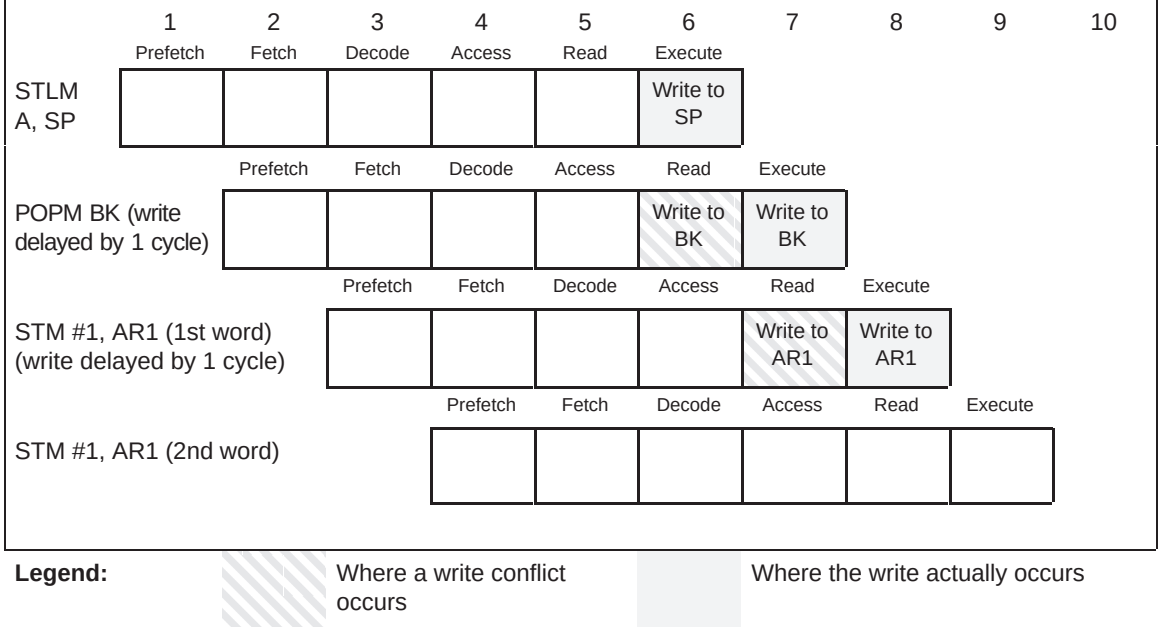
```
STLM   B, BK     ; This instruction updates BK in the execute stage.
MVDK   200h, AR1 ; This instruction tries to update AR1 in it read
                 ; stage. The CPU delays this update by one cycle to
                 ; resolve the conflict.
```



Legend: Where a write conflict occurs — Where the write actually occurs

Example 7–24. Resolving Conflict When Updating SP, BK, and ARx

```
STLM   A, SP     ; This instruction updates SP in the execute stage.
;
POPM   BK        ; This instruction tries to update BK in its read stage.
                 ; The CPU delays this update by one cycle.
;
STM    #1, AR1   ; This instruction tries to update AR1 in its read stage,
                 ; conflicting with the previous instruction.
                 ; The update is delayed by one cycle.
```



Legend: Where a write conflict occurs — Where the write actually occurs

These conflicts are automatically resolved by the ’54x CPU. This generally does not affect instruction-execution behavior. However, there is one case in which resolution by the CPU can cause an unprotected pipeline conflict. This is explained in subsection 7.5.3, *Rules to Determine DAGEN Register Access Conflicts*.

7.5.3 Rules to Determine DAGEN Register Access Conflicts

Some instructions update DAGEN registers in the read stage. This can result in conflict if the previous instruction tries to update a DAGEN register in its execute stage. This conflict is resolved automatically by the CPU's delaying the read stage update by one cycle. This delay can cause an additional cycle of latency between the instruction that writes to the DAGEN register in its read stage and the instruction that follows it.

The following set of conditions determines when such a conflict can occur:

- ❑ The first instruction is one of two types:
  - ■ Store-type instruction that accesses any DAGEN register to load a new value (see Table 7–5 on page 7-42)
  - ■ Move type 1 instruction (see Table 7–4) that has a DAGEN register conflict with the previous instruction.
- ❑ The second instruction is a constant-initialization-type instruction, a move type 1 instruction, or a move type 2 instruction (see Table 7–4) that writes to BK, SP, or *any* ARx. The instruction must not use a long offset modifier (see Example 7–27).
- ❑ The third instruction uses the same register as the second instruction in indirect addressing mode.

7.5.4 Latencies for ARx and BK

An unprotected pipeline conflict can occur when accessing an auxiliary register or BK when both of the following two conditions are met:

- ❑ An instruction writes to an auxiliary register or BK.
- ❑ The next instruction uses the *same* auxiliary register as an address pointer or index in indirect addressing mode, or uses BK in circular addressing mode. This instruction could also be an MVMM or a CMPR that reads BK or the same ARx.

This conflict occurs because the first instruction updates ARx or BK in either the read or execute stage of the pipeline and the following instruction uses BK or the same ARx when it is in the access stage of the pipeline. This results in an incorrect ARx or BK read by the second instruction, because the previous instruction has not yet updated the register's contents.

Certain instructions (see Table 7–6) do not have any latency in updating ARx. Use these instructions wherever possible to avoid pipeline conflicts.

Table 7–6. Pipeline-Protected Instructions for Updating ARx

To do this:	Use this instruction:	
Write an immediate value to ARx	STM	#lk, MMR†
Copy a memory location to ARx	MVDK	Smem, MMR†
Copy the contents of an ARx to another ARx	MVMM	MMR, MMR

† See Table 7–7 for one possible conflict with these instructions.

STM and MVDK do not conflict with the next instruction for two reasons:

- They are two-word instructions.
- They update ARx when the first instruction word is in the read stage of the pipeline.

Table 7–7 shows the latencies between instructions that update and subsequently use ARx. The second and third instructions must access the *same* auxiliary register or BK to cause a latency. Any instruction not mentioned in the table has no latency.

Table 7–8 shows the latencies between instructions that update and subsequently use BK.

Note:

You are responsible for rearranging instructions or inserting NOPs, if necessary, to accommodate latencies.

Table 7–7. Latencies for Accessing ARx

(a) Latencies based on third-instruction category

Second Instruction§	Third Instruction: Category I	Third Instruction: Category II
STM #lk, auxreg ST #lk, auxreg	0†	0
MVDK Smem, auxreg MVMD MMR, auxreg	0†	0
MVKD dmad, auxreg MVDM dmad, auxreg MVPD pmad, auxreg	1	0
POPM auxreg POPD auxreg DELAY auxreg LTD auxreg MVDD Xmem, auxreg	1†	0†
Store-type instructions (see Table 7–5)	2	1

(b) Categories for the third instruction

Category I		Category II	
MVMM auxreg, MMR			
CMPR CC, auxreg			
MVKD dmad, auxind MVDM dmad, auxind MVPD pmad, auxind MACP auxind, pmad, src MACD auxind, pmad, src	With a long-offset modifier	MVKD dmad, auxind MVDM dmad, auxind MVPD pmad, auxind MACP auxind, pmad, src MACD auxind, pmad, src	Without a long-offset modifier
ADD auxind, shift, src, dst LD auxind, shift, dst STH src, shift, auxind STL src, shift, auxind SUB auxind, shift, src, dst	With an extended shift¶ and a long-offset modifier	ADD auxind, shift, src, dst LD auxind, shift, dst STH src, shift, auxind STL src, shift, auxind SUB auxind, shift, src, dst	With an extended shift¶ and without a long-offset modifier
BANZ[D] auxind	With or without a long-offset modifier	FIRS	With one operand using indirect addressing mode with or without a long-offset modifier
Instructions not listed here that use ARx in indirect addressing mode.			

† Add one more cycle of latency if the first instruction meets the DAGEN register conflict criteria. See subsection 7.5.3, *Rules to Determine DAGEN Register Access Conflicts*, for more information.

§ The destination operand *auxreg* must point at AR0-AR7 in either direct or indirect addressing mode. The operand *auxind* must use indirect addressing mode.

¶ Shift value between –16 and 15

Notes: 1) Any instruction that does not fit in either of the two categories has zero latency.

2) The first instruction can be any ’54x instruction.

Table 7–8. Latencies for Accessing BK

(a) Latencies based on third-instruction category

Second Instruction§	Third Instruction: Category I	Third Instruction: Category II
STM #lk, bkreg ST #lk, bkreg	1†	0†
MVDK Smem, bkreg MVMD MMR, bkreg	1†	0†
MVKD dmad, bkreg MVDM dmad, bkreg MVPD pmad, bkreg	2	1
POPM bkreg POPD bkreg DELAY bkreg LTD bkreg MVDD Xmem, bkreg	2†	1†
Store-type instructions (see Table 7–5)	3	2

(b) Categories for the third instruction

Category I		Category II	
MVKD dmad, circind MVDM dmad, circind MVPD pmad, circind MACP circind, pmad, src MACD circind, pmad, src	With a long-offset modifier	MVKD dmad, circind MVDM dmad, circind MVPD pmad, circind MACP circind, pmad, src MACD circind, pmad, src	Without a long-offset modifier
ADD circind, shift, src, dst LD circind, shift, dst STH src, shift, circind STL src, shift, circind SUB circind, shift, src, dst	With an extended shift¶ and a long-offset modifier	ADD circind, shift, src, dst LD circind, shift, dst STH src, shift, circind STL src, shift, circind SUB circind, shift, src, dst	With an extended shift¶ and without a long-offset modifier
BANZ[D] circind	With or without a long-offset modifier	FIRS	With one operand using indirect addressing mode with or without a long-offset modifier
Instructions not listed here that use BK in circular addressing mode.			

† Add one more cycle of latency if the first instruction meets the DAGEN register conflict criteria. See subsection 7.5.3, *Rules to Determine DAGEN Register Access Conflicts*, for more information.

§ The destination operand *bkreg* must point at BK in either direct or indirect addressing mode. The operand *circind* must use circular addressing mode.

¶ Shift value between –16 and 15

Notes: 1) Any instruction that does not fit in either of the two categories has zero latency.

2) The first instruction can be any ’54x instruction.

Example 7–25. ARx Updated With No Latency

(a)

```
ADD    A, B
STM    #100h, AR3    ; This instruction does not conflict
                     ; with the previous instruction.
LD     *AR3+, A      ; No latency is required to use AR3.
```

(b)

```
ADD    A, B          ; This instruction does not create
                     ; a DAGEN conflict.
MVDK   200h, AR7     ; This instruction has zero latency.
STH    B, *AR7+
```

(c)

```
STLM   A, AR1        ; This instruction updates AR1 in
                     ; the execute stage, possibly
                     ; creating a DAGEN conflict.
MVDK   *(200h),AR2   ; However, this instruction uses a
                     ; long offset modifier. Therefore,
                     ; it creates no DAGEN conflict.
MAR    *AR2+         ; No latency is required to use AR2.
```

Example 7–26. ARx Updated With a 1-Cycle Latency

(a)

```
ADD    A, B          ; This instruction does not create
                     ; a DAGEN conflict.
POPM   AR3           ; This instruction has a 1-cycle
                     ; latency if AR3 is used in the next
NOP                  ; instruction. The NOP is inserted
LD     *AR3+, A      ; to avoid the conflict.
```

(b)

```
STLM   A, AR1        ; This instruction updates AR1 in
                     ; the execute stage.
POPM   BK            ; This instruction tries to update
                     ; BK in the read stage. The CPU
                     ; delays the update by one cycle.
STM    #1, AR2       ; This instruction tries to update
NOP                  ; AR2 in the read stage. The CPU
                     ; delays this update by one cycle.
LD     *AR2+, B      ; This is why one NOP is required.
```

Example 7–27. ARx Updated With and Without a 1-Cycle Latency

(a) ARx updated with a one-cycle latency

```
STLM   A, AR1        ; This instruction creates DAGEN
                     ; conflict.
MVDK   100h, AR2     ; The AR2 update is delayed by one
NOP                  ; cycle.
MAR    *AR2+
```

(b) ARx updated with no latency after reordering instructions

```
MVDK   100h, AR2     ; This instruction does not require
                     ; any latency.
STLM   A, AR1        ; This instruction is placed after
                     ; MVDK to avoid a DAGEN conflict.
MAR    *AR2+         ; No latency is required now.
```

Example 7–28. ARx Updated With and Without a 2-Cycle Latency

(a) ARx updated with a two-cycle latency

```
STLM   A, SP         ; This instruction creates a DAGEN
                     ; conflict.
POPM   AR1           ; This instruction has a 2-cycle
NOP                  ; latency due to the DAGEN conflict.
NOP
LD     *AR1+, A      ; Two NOPs avoid this conflict.
```

(b) ARx updated with no latency after reordering instructions

```
POPM   AR1           ; This instruction has a 1-cycle
                     ; latency.
STLM   A, SP         ; This instruction is placed after
                     ; the POPM to avoid DAGEN conflicts
                     ; and to eliminate the need for
                     ; NOPs.
LD     *AR1+, A
```

Example 7–29. ARx Updated With a 2-Cycle Latency

```
ADD A, B             ; This instruction does not create a
                     ; DAGEN conflict.
STLM   A, AR1        ; This instruction has a 2-cycle
NOP                  ; latency.
NOP
MVMM   AR1, AR2
```

Example 7–30. BK Updated With a 1-Cycle Latency

```
ADD    A, B          ; This instruction does not create a
STM    #100h, BK     ; DAGEN conflict.
NOP                  ; This instruction needs a 1-cycle
                     ; latency.
ADD    *AR1+%, B     ; This instruction uses BK for
                     ; circular addressing
```

7.5.5 Latencies for the Stack Pointer

Stack pointer (SP) latencies discussed in this subsection occur when SP is used in one of two ways:

- As an offset in direct addressing (when CPL = 1)
- In a push, pop, call, return, FRAME, or MVMM operation

7.5.5.1 SP Used in Compiler Mode (CPL = 1)

A pipeline conflict occurs if two conditions are simultaneously met:

- One instruction writes to SP.
- The next instruction uses SP as the base address for direct addressing in compiler mode (CPL = 1).

The conflict occurs because the second instruction tries to use SP in a pipeline stage that occurs before the previous instruction updates it.

Table 7–9 lists the latencies between instructions that update and subsequently use SP in compiler mode.

Note:

You are responsible for rearranging instructions or inserting NOPs, if necessary, to accommodate for SP latencies.

Table 7–9. Latencies for SP in Compiler Mode (CPL = 1)

(a) Latencies based on third-instruction category

Second Instruction	Third Instruction: Category I	Third Instruction: Category II
STM #lk, SP ST #lk, SP	1†	0†
MVDK Smem, SP MVMD MMR, SP	1†	0†
MVKD dmad, SP MVDM dmad, SP MVPD pmad, SP	2	1
MVDD Xmem, spind POPM SP POPD SP	2†	1†
FRAME k MVMM MMR, SP POPM MMR POPD Smem PSHM MMR PSHD Smem RETFD	1	0
Store-type instructions (see Table 7–5)	3	2

(b) Categories for the third instruction

Category I	Category II
All instructions using SP in direct addressing mode, except those listed in Category II.	MVKD dmad, dirmem MVDM dirmem, MMR MVPD pmad, dirmem MACP dirmem, pmad, src MACD dirmem, pmad, src ADD dirmem, shift, src, dst LD dirmem, shift, dst STH src, shift, dirmem STL src, shift, dirmem SUB dirmem, shift, src, dst } With an extended shift‡

Legend:

SP	Destination operand pointing to the stack pointer in either direct or indirect addressing modes
MMR	Any memory-mapped register except SP
spind	Destination operand pointing to the stack pointer using indirect addressing mode
dirmem	Operand using direct addressing mode in compiler mode (CPL = 1)

† Add one more cycle of latency if the first instruction meets the DAGEN register conflict criteria. See subsection 7.5.3, *Rules to Determine DAGEN Register Access Conflicts*, for more information.

‡ Shift value between –16 and 15.

Notes: 1) Any instruction that does not fit in either of the two categories has zero latency.

2) The first instruction can be any '54x instruction.

Example 7–31. SP Load With No Latency in Compiler Mode (CPL =1)

(a)

```
ADD    A, B       ; This instruction does not create a
                  ; DAGEN conflict.
MVDK   100h, SP   ; This SP update requires a zero
                  ; latency according to the above table.
ADD    50h,-3,A,B
```

(b)

```
STLM   A, AR2     ; This instruction does not affect
                  ; pipeline latency.
POPM   AR1        ; This SP update requires a zero
                  ; latency according to the table.
MVKD   100h,1h
```

Example 7–32. SP Load With a 1-Cycle Latency in Compiler Mode (CPL = 1)

(a)

```
STLM   A, AR2     ; This instruction does not affect
                  ; pipeline latency.
MVMM   AR1, SP    ; This SP update requires a one-cycle
                  ; latency since the next instruction
NOP               ; uses SP when CPL = 1.
LD     50h,A
```

(b)

```
ADD    A, B       ; This instruction does not create a
                  ; DAGEN conflict
STM    #100h, SP  ; This SP update requires a one-cycle
                  ; latency since the next instruction
NOP               ; uses SP when CPL = 1.
LD     50h,A
```

(c)

```
ADD    A,B        ; This instruction does not affect
                  ; pipeline latency.
RETFD             ; SP is incremented after popping the
                  ; return address.
NOP
LD     50h, A     ; This instruction cannot be placed in
                  ; the first delay slot since it uses
                  ; direct addressing mode with the new
                  ; SP value.
```

Example 7–33. SP Load With and Without a 2-Cycle Latency

(a) SP Load With a Two-Cycle Latency

```
    STLM   A, BK      ; This instruction creates a DAGEN
                      ; conflict.
    MVDK   100h, SP   ; This SP update requires 2 cycles
    NOP               ; of latency according to the above
    NOP               ; table.
    LD     50h, A
```

(b) SP Load With No Latency

```
    MVDK   100h, SP   ; This SP update requires 1 cycle
                      ; of latency.
    STLM   A, BK      ; This instruction is placed after the
                      ; MVDK instruction to prevent a DAGEN
                      ; conflict.
    LD     50h, A     ; No NOPs are required in this case.
```

Example 7–34. SP Load With a 2-Cycle Latency in Compiler Mode (CPL = 1)

```
    ADD    A, B       ; This instruction does not affect
                      ; pipeline latency.
    POPM   SP         ; The new value of SP is popped from
    NOP               ; the stack. Two NOPs are required to
    NOP               ; use the new value of SP.
    LD     50h, A
```

Example 7–35. SP Load With a 3-Cycle Latency in Compiler Mode (CPL = 1, DP = 0)

```
    STLM   A, AR1     ; This instruction does not affect
                      ; pipeline latency.
    STH    A, SP      ; This SP update requires a three-cycle
    NOP               ; latency since the next instruction
    NOP               ; uses SP when CPL is 1.
    NOP
    LD     50h,A
```

7.5.5.2 SP Used in Push, Pop, Call, Return, FRAME, and MVMM Operations

A pipeline conflict occurs if two conditions are simultaneously met:

- ❑ An instruction updates SP.
- ❑ The next instruction uses the stack for a push, pop, call, return, FRAME, or MVMM operation.

The conflict occurs because the second instruction tries to use SP in a pipeline stage that occurs before the stage in which the previous instruction updates SP.

Table 7–10 lists instructions that do not have any latency in updating SP when the CPU is not in compiler mode (CPL = 0). These instructions should be used wherever possible to avoid conflicts.

Table 7–10. Pipeline-Protected Instructions to Update SP in Noncompiler Mode (CPL = 0)

To do this:	Use this instruction:	
Write an immediate value to SP	STM	#lk, SP†
Copy a memory location to SP	MVDK	Smem, SP†
Copy the contents of an ARx or BK to SP	MVMM	MMR, SP
Move SP by a frame	FRAME	k

† See Table 7–11 for one possible conflict with these instructions.

Table 7–11 lists the latencies between instructions that update and use SP in noncompiler mode (CPL = 0).

Note:

You are responsible for rearranging instructions or inserting NOPs, if necessary, to accommodate SP latencies.

Table 7–11. Latencies for SP in Noncompiler Mode (CPL = 0)

(a) Latencies based on third-instruction category

Second Instruction	Third Instruction: Category I	Third Instruction: Category II
STM #lk, SP ST #lk, SP	0†	0
MVDK Smem, SP MVMD MMR, SP	0†	0
MVKD dmad, SP MVDM dmad, SP MVPD pmad, SP	1	0
MVDD Xmem, spind POPM SP POPD SP	1†	0†
Store-type instructions (see Table 7–5)	2	1

(b) Categories for the third instruction

Category I		Category II	
PSHM MMR PSHD Smem POPM MMR PSHM Smem	Without a long-offset modifier	PSHM MMR PSHD Smem POPM MMR PSHM Smem	With a long-offset modifier
CALL[D] address CC[D] address FCALL[D] FRET[D] FRETE[D] INTR k RC[D] RET[D] RETE[D] RETF[D]		CALA[D] address FCALA[D]	
MVMM SP, MMR			
FRAME k TRAP n			

Legend:
SP Destination operand pointing to the stack pointer in either direct or indirect addressing modes
MMR Any memory-mapped register except SP
spind Destination operand pointing to the stack pointer using indirect addressing mode
† Add one more cycle of latency if the first instruction meets the DAGEN register conflict criteria. See subsection 7.5.3, *Rules to Determine DAGEN Register Access Conflicts*, for more information.

Notes: 1) Any instruction that does not fit in either of the two categories has zero latency.

2) The first instruction can be any '54x instruction.

Example 7–36. SP Load With No Latency in Noncompiler Mode (CPL = 0)

(a)

```
ADD    A, B       ; This instruction does not create
                  ; a DAGEN conflict
STM    #100h, SP  ; This SP update does not require any
                  ; latency according to the above table.
PSHM   AR1
```

(b)

```
STH    A, 100h    ; This instruction does not create
                  ; a DAGEN conflict.
MVDK   200h, SP   ; This SP update does not require any
                  ; latency according to the above table.
FRAME  10
```

Example 7–37. SP Load With and Without a 1-Cycle Latency in Noncompiler Mode (CPL = 0)

(a) SP Load With a One-Cycle Latency

```
STLM   A, AR1     ; This instruction causes a DAGEN
                  ; conflict with the next instruction.
MVDK   200h, SP   ; This SP update requires a one-cycle
NOP               ; latency.
PSHM   AR2
```

(b) SP Load With No Latency

```
MVDK   200h, SP   ; This instruction requires no latency.
STLM   A, AR1     ; This instruction was placed after
                  ; MVDK to avoid a DAGEN conflict.
PSHM   AR2
```

Example 7–38. SP Load With a 1-Cycle Latency in Noncompiler Mode (CPL = 0)

```
STLM   A, AR1     ; This instruction does not affect the
                  ; pipeline latency for the next
                  ; instruction.
STLM   B, SP      ; This SP update requires a one-cycle
NOP               ; latency.
CALA   A
```

7.5.6 Latencies for Temporary Register (T)

A pipeline conflict can occur when accessing T if two conditions are simultaneously met:

- ❑ An instruction writes to T
- ❑ The next instruction uses T for a shift or bit-test operation.

The conflict occurs because the second instruction tries to use T in a pipeline stage that occurs before the previous instruction updates it.

Table 7–12 lists instructions that do not have any latency in updating T. Use these instructions wherever possible to avoid any conflicts.

Table 7–12. Pipeline-Protected Instructions for Updating T

To do this:	Use this instruction:
Write an immediate value to T	STM #lk, T
Copy a memory location to T	MVDK Smem, T
Copy a memory location to T	LD Smem, T
Copy a memory location to T	ST src, Ymem || LD Xmem, T

Table 7–13 lists the latencies between instructions that update and use T.

Note:

You are responsible for rearranging instructions or inserting NOPs, if necessary, to accommodate T latencies.

Table 7–13. Latencies for the T Register Based on Second-Instruction Category

(a) Latencies Based on Second-Instruction Category

First Instruction		Second Instruction Category I
MVKD MVDM	dmad, T dmad, T	1
POPM POPD DELAY MVDD	T T T Xmem, T_{ind}	1
Store-type instructions (see Table 7–5)		1
EXP	src	1

(b) Categories for the Second Instruction

Category I		
LD ADD SUB	Smem, TS, dst Smem, TS, src Smem, TS, src	Without a long-offset modifier
NORM	src, dst	
BITT	Xmem	
DADST	Lmem, dst	
DSADT	Lmem, dst	
DSUBT	Lmem, dst	

Legend: T Destination operand pointing at T in either direct or indirect addressing modes.
T_{ind} Destination operand pointing at T using indirect addressing mode.

Note: Any instruction that does not fit in Category I has zero latency.

Example 7–39. T Load With No Latency

(a)

```
    LD     *AR3+, T      ; This T update does not require
    LD     *AR5+,TS,A    ; any latency.
```

(b)

```
    STM    #100h, T      ; This T update does not require
    LD     *AR5+,TS,A    ; any latency.
```

Example 7–40. T Load With a 1-Cycle Latency

(a)

```
    POPM   T             ; This instruction requires a one-
    NOP                  ; cycle latency.
    BITT   *AR5+
```

(b)

```
    EXP    A             ; This instruction requires a one-
    NOP                  ; cycle latency.
    NORM   A
```

7.5.7 Latencies for Accessing Status Registers

The following status register fields and bits are affected by latency:

- ARP (auxiliary register pointer)
- CMPT (compatibility mode bit)
- CPL (compiler mode bit)
- DP (data page pointer)
- SXM (sign-extension mode bit)
- ASM (accumulator shift mode field)
- BRAF (block-repeat activity flag)

7.5.7.1 ST1 and a Repeat Block Loop

Some instructions write to ST1 in the execute stage of the pipeline. If any of these instructions is immediately followed by a RPTB[D] instruction that sets the BRAF flag in ST1 an incomplete repeat-block loop results. This occurs because RPTB[D] sets BRAF in the access stage of the pipeline and the previous instruction overwrites ST1 one cycle later.

To avoid this conflict, it is recommended that only instructions listed in Table 7–14 be used to write to ST1 immediately prior to a RPTB[D] instruction. Any other instruction that writes to ST1 must not be immediately followed by a RPTB[D] instruction.

Table 7–14. Recommended Instructions for Writing to ST1

To do this	Use this instruction
Store a value to ST1	STM #k, ST1 ST #k, ST1
Copy a value from data memory to ST1	MVDK #k, ST1 MVMD #k, ST1
Clear a bit in ST1	RSBX
Set a bit in ST1	SSBX
Load ASM with a value	LD #k, ASM LD Smem, ASM

7.5.7.2 Updating ARP in Compatibility Mode (CMPT = 1) and CMPT bit

A pipeline conflict can occur if two conditions are simultaneously met:

- ❑ An instruction updates ARP or CMPT.
- ❑ The next instruction uses ARP or CMPT to update the address pointer in indirect addressing mode.

The conflict occurs because the second instruction uses ARP or CMPT in a pipeline stage that occurs before the previous instruction updates ARP or CMPT.

Table 7–15 lists one instruction that does not have any latency in updating ARP when the CPU is in compatibility mode. Use this instruction wherever possible to avoid any conflicts.

Table 7–15. Pipeline-Protected Instruction to Update ARP in Compatibility Mode (CMPT = 1)

To do this:	Use this instruction:
Load ARP field of ST0 register	LD #k, ARP

Table 7–16 lists the latencies between instructions that update and use ARP or CMPT.

Notes:

1) You are responsible for rearranging instructions or inserting NOPs, if necessary, to accommodate latencies.
2) In compatibility mode (CMPT = 1), ARP is automatically updated by instructions that use indirect addressing mode. There is no pipeline conflict associated with such an ARP update.
3) ARP must always be set to 0 when the DSP is in standard mode (CMPT = 0). At reset, both ARP and CMPT are set to 0 automatically.

Table 7–16. Latencies for ARP in Compatibility Mode (CMPT = 1) and CMPT bit

(a) Latencies based on second-instruction category

First Instruction	Second Instruction: Category I	Second Instruction: Category II
STM #lk, status ST #lk, status	2	2
MVDK Smem, status MVMD MMR, status	2	2
MVKD dmad, status MVDM dmad, status MVPD pmad, status	3	2
MVPD pmad, status	3	3
POPM status POPD status DELAY status LTD status MVDD status	3	2
Store-type instruction (see Table 7–5)	3	2
SSBX statbit RSBX statbit	3	2

(b) Categories for the second instruction

Category I		Category II	
MVKD dmad, auxind MVDM dmad, auxind MVPD dmad, auxind MACP dmad, auxind, pmad, src MACD auxind, pmad, src	With a long-offset modifier	MVKD dmad, auxind MVDM dmad, auxind MVPD pmad, auxind MACP auxind, pmad, src MACD auxind, pmad, src	Without a long-offset modifier
ADD auxind, shift, src, dst LD auxind, shift, dst STH src, shift, auxind STL arc, shift, auxind SUB auxind, shift, src, dst	With an extended shift† and a long offset modifier	ADD auxind, shift, src, dst LD auxind, shift, dst STH src, shift, auxind STL src, shift, auxind SUB auxind, shift, src, dst	With an extended shift† and without a long-offset modifier
All other instructions that use ARP or CMPT in indirect addressing mode with or without a long offset modifier.			

Legend:
- status — Destination operand pointing to ST0 or ST1 to update ARP or CMPT respectively in either direct or indirect addressing modes
- MMR — Any memory-mapped register
- auxind — A read or write operand using indirect addressing mode
- statbit — Destination operand writing to a bit in ARP or CMPT
- † Shift value between –16 and 15.

Note: Any instruction that does not fit in either of the two categories has zero latency.

Example 7–41. ARP Load With No Latency in Compatibility Mode (CMPT = 1)

```
LD     #1h, ARP  ; This ARP load does not require any
                 ; latency.
LD     *AR0, A
```

Example 7–42. ARP Load With a 2-Cycle Latency in Compatibility Mode (CMPT = 1)

```
STLM   A, ST0    ; The ARP field of ST0 is updated here.
NOP
NOP
ADD    *AR0+,–3,B ; The new ARP value is used here
```

Example 7–43. ARP Load With a 3-Cycle Latency in Compatibility Mode (CMPT = 1)

```
POPM   ST0       ; The ARP field of ST0 is updated here.
NOP
NOP
NOP
LD     *AR0+, A  ; The new ARP value is used here.
```

7.5.7.3 Updating DP in Direct Addressing Mode (CPL = 0)

A pipeline conflict can occur if two conditions are simultaneously met:

- An instruction updates DP.
- The next instruction uses DP as the base address for direct addressing in noncompiler mode (CPL = 0).

The conflict occurs because the second instruction uses DP in a pipeline stage that occurs before the previous instruction updates it.

Table 7–17 lists instructions that do not have any latency in writing to DP. It is recommended that these instructions be used wherever possible to avoid conflicts.

Table 7–17. Recommended Instructions to Update DP in Noncompiler Mode (CPL = 0)

To do this:	Use this instruction:	
Load an immediate number to DP	LD	#k, DP
Copy contents of a memory location to DP	LD	Smem, DP

Table 7–18 lists the latencies between instructions that update DP and subsequently use it.

Note:

You are responsible for rearranging instructions or inserting NOPs, if necessary, to accommodate latencies.

Table 7–18. Latencies for DP in Noncompiler Mode (CPL = 0)

(a) Latencies based on second-instruction category

First Instruction	Second Instruction: Category I	Second Instruction: Category II
STM #lk, status ST #lk, status	2	2
MVDK Smem, status MVMD MMR, status	2	2
MVKD dmad, status MVDM dmad, status	3	2
MVPD pmad, status	3	3
POPM status POPD status MVDD status	3	2
Store-type instruction (see Table 7–5)	3	2
SSBX ST0, statbit RSBX ST0, statbit	3	2

(b) Categories for the second instruction

Category I	Category II
All instructions that use DP for direct addressing mode except those listed in Category II.	MVKD dmad, dirmem MVPD pmad, dirmem MACP dirmem, pmad, src MACD dirmem, pmad, src
	ADD dirmem, shift, src, dst LD dirmem, shift, dst STH src, shift, dirmem STL src, shift, dirmem SUB dirmem, shift, src, dst } With an extended shift†

Legend: status Destination operand pointing to ST0 to update DP in either direct or indirect addressing modes
MMR Any memory-mapped register
statbit Destination operand writing to a bit in DP field of ST0
dirmem A read or write operand using direct addressing mode when CPL = 0
† Shift value between –16 and 15.

Note: Any instruction that does not fit in either of the two categories has zero latency.

Example 7–44. DP Load With No Latency in Noncompiler Mode (CPL = 0)

(a)

```
LD     #2h, DP   ; This DP load does not require any
                 ; latency.
LD     27h, A
```

(b)

```
LD     100h, DP  ; This DP load does not require any
                 ; latency.
LD     27h, A
```

Example 7–45. DP Load With a 2-Cycle Latency in Noncompiler Mode (CPL = 0)

```
STLM   A, ST0    ; The DP field of ST0 is updated here.
NOP
NOP
STH    B,-3,27h  ; The new DP value is used here.
```

Example 7–46. DP Load With a 3-Cycle Latency in Noncompiler Mode (CPL = 0)

```
POPM   ST0       ; The DP field of ST0 is updated here.
NOP
NOP
NOP
LD     27h, A    ; The new DP value is used here
```

7.5.7.4 Updating CPL

A pipeline conflict can occur if two conditions are simultaneously met:

- ❑ An instruction modifies CPL.
- ❑ The next instruction uses direct addressing mode.

The conflict occurs because the second instruction reads CPL in a pipeline stage that occurs before the previous instruction updates it.

Table 7–19 lists the latencies between instructions that update CPL and subsequently use it.

Note:

You are responsible for rearranging instructions or inserting NOPs, if necessary, to accommodate latencies.

Table 7–19. Latencies for the CPL Bit

(a) Latencies based on second-instruction category

First Instruction	Second Instruction: Category I	Second Instruction: Category II
STM #lk, status ST #lk, status	2	1
MVDK Smem, status MVMD MMR, status	2	1
MVKD dmad, status MVDM dmad, status	3	2
MVPD pmad, status	3	3
POPM status POPD status MVDD status	3	2
Store-type instruction (see Table 7–5)	3	2
SSBX CPL RSBX CPL	3	2

(b) Categories for the second instruction

Category I	Category II	
All instructions that use direct addressing mode except those listed in Category II.	MVKD dmad, dirmem MVPD pmad, dirmem MACP dirmem, pmad, src MACD dirmem, pmad, src	
	ADD dirmem, shift, src, dst LD dirmem, shift, dst STH src, shift, dirmem STL arc, shift, dirmem SUB dirmem, shift, src, dst	With an extended shift†

Legend:
MMR Any memory-mapped register
dirmem A read or write operand using direct addressing mode when CPL = 0
status Destination operand pointing to ST1 to modify CPL in either direct, indirect, or memory-mapped addressing mode

† Shift value between –16 and 15.

Note: Any instruction that does not fit in either of the two categories has zero latency.

Example 7–47. CPL Update With a 1-Cycle Latency

```
    STM    #k, ST1       ; This instruction modifies the CPL
                         ; bit of ST1.
    NOP
    MVKD   1000h, 30h    ; Data read from 1000h is written to
                         ; (SP + 30h)
```

Example 7–48. CPL Update With a 2-Cycle Latency

```
    RSBX   CPL           ; CPL is changed from 1 to 0.
    NOP                  ; Two NOPs are required, since a LD
    NOP                  ; with an extended shift is being
                         ; used to access an operand.
    LD     27h, –1, A    ; The operand is read from the
                         ; current data page.
```

Example 7–49. CPL Update With a 3-Cycle Latency

```
    SSBX   CPL           ; CPL is changed from 0 to 1.
    NOP                  ; These NOPs can be replaced by
    NOP                  ; other instructions that do not use
    NOP                  ; direct addressing mode to access
                         ; operands.
    LD     27h, A        ; The operand is read at an offset
                         ; of 27h from SP.
```

7.5.7.5 Updating SXM

A pipeline conflict can occur if two conditions are simultaneously met:

- ❏ An instruction modifies SXM.
- ❏ The next instruction uses SXM to control sign extension.

The conflict occurs because the second instruction uses SXM in a pipeline stage that occurs before the previous instruction updates it.

Table 7–20 lists the latencies between instructions that update SXM and subsequently use it.

Note:

You are responsible for rearranging instructions or inserting NOPs, if necessary, to accommodate latencies.

Table 7–20. Latencies for the SXM Bit

(a) Latencies based on second-instruction category

First Instruction	Second Instruction Category I
MVKD dmad, status MVDM dmad, status POPM status POPD status MVDD Xmem, status	1
Store-type instruction (see Table 7–5)	1
SSBX SXM RSBX SXM	1

(b) Category for the second instruction

Category I
All instructions affected by the sign-extension mode bit, except those that require an *Smem* operand with a long offset modifier (for example, LD *+AR1 (100h), A)

Legend: status Destination operand pointing at ST1 to update SXM in either direct, indirect, or memory-mapped addressing mode

Note: Any instruction that does not fit in Category I has zero latency.

Example 7–50. SXM Update With No Latency

```
    RSBX   SXM    ; This instruction modifies the SXM bit of
                  ; ST1.
    ADD    *+AR1(100h),A
```

Example 7–51. SXM Update With a 1-Cycle Latency

(a)

```
    RSBX   SXM    ; This SXM load requires one cycle of
                  ; latency.
    NOP
    LD     *AR5+, A
```

(b)

```
    POPM   ST1    ; This instruction modifies the SXM bit of
    NOP           ; ST1.
    ADD    *AR2+,A
```

(c)

```
    STLM   A, ST1 ; This instruction modifies the SXM bit of
                  ; ST1.
    NOP
    SUB    *AR2-,A
```

7.5.7.6 ASM Field Used for Shift Operations

A pipeline conflict can occur if two conditions are simultaneously met:

- An instruction modifies ASM.
- The next instruction uses ASM as the shift-count value.

The conflict occurs because the second instruction reads ASM in a pipeline stage that occurs before the previous instruction updates it.

Table 7–21 lists instructions that do not have any latency for writing to the ASM bit field. Use these instructions wherever possible to avoid any conflicts.

Table 7–21. Pipeline-Protected Instructions for Writing to ASM

To do this:	Use this instruction:
Load an immediate number to ASM	LD #k, ASM
Copy contents of a memory location to ASM	LD Smem, ASM

Table 7–22 lists the latencies between instructions that write to ASM and those that subsequently use it.

Note:

You are responsible for rearranging instructions or inserting NOPs, if necessary, to accommodate latencies.

Table 7–22. Latencies for ASM Bit Field

(a) Latencies based on second-instruction category

First Instruction	Second Instruction Category I
MVKD dmad, status MVDM dmad, status POPM status POPD status MVDD Xmem, status	1
Store-type instruction (see Table 7–5)	1
SSBX STI, asmbit RSBX STI, asmbit	1

(b) Category for the second instruction

Category I	
STH src, ASM, Smem STL src, ASM, Smem	} Without a long-offset modifier
ST src, Ymem || LD/ADD/SUB/MAC/MAS/MPY	
SACCD src, Smem, cond	

LD src, ASM, dst
ADD src, ASM, dst
SUB src, ASM, dst	

Legend: asmbit Destination operand writing to a bit in ASM field of ST1
status Destination operand pointing at ST1 to update ASM in direct, indirect,or memory-mapped addressing mode

Note: Any instruction that does not fit in either of the two categories has zero latency.

Example 7–52. ASM Update With No Latency

(a)

```
    LD     #6, ASM   ; This instruction loads ASM with no
                     ; latency.
    STH    A,ASM,*AR1+
```

(b)

```
    LD     100h, ASM ; This instruction loads ASM with no
                     ; latency
    ADD    A,ASM,B
```

(c)

```
    STLM   A, ST1    ; This instruction modifies the ASM
                     ; field of ST1. No latency is needed
                     ; since STL uses a long offset
                     ; modifier.
    STL    A,ASM,*+AR5(100h)
```

Example 7–53. ASM Update With a 1-Cycle Latency

```
    POPM   ST1       ; This instruction modifies the ASM
    NOP              ; field of ST1
    SUB    A,ASM,B
```

7.5.8 Latencies in Repeat-Block Loops

The following status register fields and bits are affected by latency:

- ❏ BRC (block-repeat counter register)
- ❏ BRAF (block-repeat active flag)

7.5.8.1 Updating Block-Repeat Counter (BRC) Register

A pipeline conflict can occur if two conditions are simultaneously met:

- ❏ An instruction writes to the BRC register
- ❏ The next instruction is an RPTB[D]

The conflict occurs because the second instruction reads BRC in a pipeline stage that occurs before the previous instruction updates it.

There are certain instructions which do not cause any pipeline conflicts when updating BRC. Use these instructions wherever possible to avoid conflicts.

Table 7–23. Recommended Instructions for Writing to BRC Before an RPTB Loop

To do this	Use this instruction
Write an immediate value to BRC	STM #lk, BRC
Copy a memory location to BRC	MVDK Smem, BRC

Table 7–24 lists latencies between instructions that update BRC and an RPTB[D] instruction.

Notes:

1) Do not place instructions that modify BRC in the delay slots of a RPTBD instruction.

2) You are responsible for rearranging instructions or inserting NOPS, if necessary, to accommodate latencies.

Table 7–24. Latencies for Updating BRC Before an RPTB Loop

First Instruction	Latency if Second Instruction Is RPTB[D]
MVDK Smem, BRC MVMD MMR, BRC	0
STM #k, BRC ST #k, BRC	0
All other instructions that modify BRC	1

Example 7–54. Loading BRC Before Executing a New Repeat-Block Loop

(a)

```
    STM    #lk,BRC       ; There is no latency when BRC is
    RPTB   endloop–1     ; loaded via STM before a new RPTB
    ...                  ; loop.
endloop:
```

(b)

```
    MVDK      count,BRC ; There is no latency when BRC is
    RPTBD     endloop–1 ; loaded using MVDK before a new
    ...                 ; RPTB loop
endloop:
```

(c)

```
    STLM   A,BRC         ; There is a 1 cycle latency when
    NOP                  ; BRC is loaded using an STLM
    RPTB   endloop–1     ; instruction.
    ...
endloop:
```

(d)

```
    POPM   BRC           ; There is a 1 cycle latency when
    NOP                  ; BRC is loaded using a POPM
    RPTBD  endloop–1     ; instruction.
    ...
endloop:
```

In a repeat-block loop, BRC is decremented when the last instruction in the loop is in the decode stage of the pipeline. However, the SRCCD instruction writes the BRC's contents in the execute stage of the pipeline. This can result in an incorrect BRC value written by the SRCCD instruction. The pipeline conflict can be avoided by placing the SRCCD instruction at least three instruction words from the bottom of the loop, as shown in Example 7–55 and Example 7–56.

Example 7–55. SRCCD Instruction With No Latency

```
    RPTB   endloop–1
    ...
    SRCCD  *AR3, ALEQ    ; Placing the SRCCD instruction in
                         ; this position ensures that current
                         ; value of BRC will be written
                         ; to memory.
    ADD    *AR1+,A
    SUB    *AR2–,A
    STH    A, *AR1+
endloop:
```

Example 7–56. SRCCD Instruction With a 3-Cycle Latency

```
    RPTB   endloop–1
    ...
    SRCCD  *AR3, ALEQ    ;This ensures that current value of
    NOP                  ; BRC will be written to memory.
    NOP
    NOP
endloop:
```

There is also a 5-to-6-cycle latency when writing a new value to BRC from within a RPTB loop. The latencies described in Table 7–25 are relevant only if BRC is modified while a RPTB loop is active. See Example 7–57 for details.

Table 7–25. Latencies for Updating BRC From Within an RPTB Loop

Instruction	Latency
STM #lk, BRC ST #lk, BRC MVDK Smem, BRC MVMD MMR, BRC	The next 5 instruction words must not contain the last instruction in the RPTB loop.
All other instructions that modify BRC	The next 6 instruction words must not contain the last instruction in the RPTB loop.

Example 7–57. Modifying BRC From Within an RPTB Loop

```
    RPTB   endloop–1
    ...
    XC     2, Condition     ; If Condition is evaluated as
    MVDK   5h, BRC          ; true, write the new count value
                            ; to BRC.
    LD     *AR1+, A         ; These six instructions provide
    ADD    *AR2–, A         ; sufficient latency for the new
    SUB    *AR3+, A         ; BRC value to take effect before
    LD     *AR4+, T         ; the next iteration begins.
    MPYA   A
    STL    A, *AR3+
endloop:
```

7.5.8.2 Deactivating the Block-Repeat Active Flag (BRAF)

The ’54x sets the block-repeat active flag (BRAF) to 1 to indicate that the repeat-block loop is active. BRAF is set or cleared in the decode stage of the first instruction of the repeat-block loop during the last loop iteration (BRC = 0). BRAF is tested by the device at the end of each loop iteration to determine whether the next prefetch will be from the top of the loop or not.

BRAF can be deactivated in software to terminate the repeat-block prematurely. This, however, must be done early enough in the pipeline so that BRAF is cleared prior to the prefetch of the instruction at the top of the loop. Therefore, an instruction that clears BRAF (such as RSBX) must be placed at least six instructions words before the end of the repeat-block loop. This is shown in Example 7–58.

Example 7–58. BRAF Deactivation

```
    RPTB   endloop-1
    ...
    RSBX   BRAF      ; This ensures that the loop will
    NOP              ; terminate after completing this
    NOP              ; iteration.
    NOP
    NOP              ; These six NOPs may be replaced by
    NOP              ; other instructions.
    NOP
endloop:
```

7.5.9 Latencies for the PMST

PMST fields OVLY, DROM, MP/$\overline{\text{MC}}$, and IPTR configure the '54x memory space. When an instruction modifies one of these fields, a certain number of pipeline cycles must pass before the new memory space can be accessed. This latency is required because the PMST fields are updated in the read or execute stage of the pipeline while instructions in earlier pipeline stages may be accessing that memory space. In the case of program memory control via OVLY, IPTR, and MP/$\overline{\text{MC}}$ fields, the prefetch stage of the pipeline evaluates these bits. In the case of data memory control via the DROM field, the access or read stage evaluates this bit.

If an external memory access occurs, additional cycles are required for operations affected by the changing bit. Any change in PMST fields is delayed until an in-process external access is completed. For example, if an external memory access is occurring while an instruction in the execute stage of the pipeline tries to modify OVLY, the update is delayed until the external bus cycle is completed. This, in turn, requires additional cycles to those listed in the following tables.

Table 7–26 lists the latencies between instructions that write to the OVLY, IPTR, or MP/$\overline{\text{MC}}$ bit fields and those instructions that are subsequently fetched from the new memory space.

Note:

You are responsible for rearranging instructions or inserting NOPs, if necessary, to accommodate latencies.

Table 7–26. Latencies for OVLY, IPTR, and MP/$\overline{MC}$ Bits

(a) Latencies based on second-instruction category

	Second Instruction			
First Instruction	**Category I**	**Category II**	**Category III**	**Category IV**
STM #lk, pmst ST #lk, pmst MVDK dmad, pmst MVMD MMR, pmst	0	0	1	2
All other instructions that modify OVLY, IPTR, MP/$\overline{MC}$	0	1	2	3

(b) Categories for the second instruction

Category I	**Category II**	**Category III**	**Category IV**
BACC[D]	BC[D]	B[D]	INTR
CALA[D]	CC[D]	BANZ[D]	RETF[D]
FRET[D]	RC[D]	CALL[D]	TRAP
FRETE[D]	RET[D]	FB[D]	
FBACC[D]	RETE[D]	FCALL[D]	
FCALA[D]			

Legend: pmst Destination operand pointing at PMST to modify OVLY, IPTR, MP/$\overline{MC}$ in either direct, indirect, or memory-mapped addressing modes

Notes: 1. Additional latency cycles are required if an external memory access is in progress when an instruction is trying to modify OVLY, IPTR, or MP/$\overline{MC}$ bit fields.
2. The second instruction loads PC with a new value that points to the modified program address range.

Example 7–59. OVLY Setup Followed by an Unconditional Branch (DP = 0)

```
ORM    #20h, PMST    ; This instruction sets OVLY to 1.
NOP
NOP
B      onchip        ; Branch to on-chip dual-access
                     ; memory.
```

Example 7–60. OVLY Setup Followed by a Conditional Branch

```
MVDK   5h, PMST      ; This instruction sets OVLY to 1.
BC     onchip,AEQ    ; Branch to on-chip dual-access
                     ; memory.
```

Example 7–61. OVLY Setup Followed by a Return (DP = 0)

```
ANDM #0ffdfh, PMST  ; This instruction sets OVLY to 0.
NOP
RET                 ; Return to off-chip memory.
```

Example 7–62. MP/$\overline{MC}$ Setup Followed by an Unconditional Delayed Call

```
STLM   A, PMST      ;This instruction sets MP/MC to 1.
NOP
NOP
CALLD  offchip      ; Call a routine in external
                    ; program memory after executing
STM    #k, AR1      ; this 2-word instruction.
```

Example 7–63. IPTR Setup Followed by a Software Trap

```
STM    #k, PMST     ; This instruction relocates
NOP                 ; interrupt vectors by writing to
NOP                 ; the IPTR bit field.
TRAP                ; Fetch a TRAP vector from the
                    ; relocated vector table.
```

Table 7–27 lists the latencies between instructions that write to the DROM bit of PMST and those that subsequently read from or write to the DROM address range.

Note:

You are is responsible for rearranging instructions or inserting NOPs, if necessary, to accommodate latencies.

Table 7–27. Latencies for the DROM Bit

(a) Latencies based on second-instruction category

First instruction is...	And second instruction is Category I, the latency is...
STM #lk, drom ST #lk, drom MVDK dmad, drom MVMD MMR, drom	2
All other instructions that modify DROM	3

(b) Catagory for the second-instruction

Category I
All instructions that read from or write to the DROM address range

Legend: drom Destination operand pointing at PMST to modify DROM bit in either direct , indirect, or memory-mapped addressing modes

Notes:
1. Additional latency cycles are required if an external memory access is occurring at the time when an instruction is trying to modify the DROM bit field.
2. Any instruction not listed in this table that modifies DROM bit of PMST register has zero latency.

Example 7–64. DROM Setup Followed by a Read Access (DP = 0)

```
ORM    #8h, PMST        ; This instruction sets DROM = 1.
NOP
NOP
NOP
LD     *AR3, A          ; Reads from on-chip DROM
```

Example 7–65. DROM Setup Followed by a Dual-Read Access

```
STM    #k, PMST         ; This instruction sets DROM = 1.
NOP
NOP
MPY    *AR3+,*AR4+, A   ; This instruction reads from
                        ; on-chip DROM.
```

7.5.10 Latencies for Memory-Mapped Accesses to Accumulators

Accumulators A and B can be addressed as memory-mapped registers using memory-mapped, direct, or indirect addressing modes. Generally, it is not useful to access accumulators as memory-mapped registers, because the ’54x instruction set supports direct accesses to the accumulators. Some examples of the instructions that support direct access to accumulators are ADD, SUB, AND, OR, XOR, LD, STH, STL, MAC, and MPYA.

When the two accumulators are accessed using instructions that do not access them as memory-mapped registers, no pipeline latencies occur. In rare cases when you access the accumulators through the memory-mapped registers AG, AH, AL, BG, BH, and BL, you can use any instruction that uses memory-mapped, direct, or indirect addressing modes to access operands. Examples of such instructions are POPM AL and PSHM AH. Note that DP must be zero in order to access memory-mapped registers via direct addressing modes.

A pipeline conflict can occur when two conditions are simultaneously met:

- One instruction modifies an accumulator (either A or B) directly.
- The next instruction tries to read that accumulator as a memory-mapped register.

The conflict occurs because the first instruction updates an accumulator at the same time when the next instruction tries to read it as a memory-mapped register.

Example 7–66. Accumulator Access With a 1-Cycle Latency

```
ADD    Smem, A       ; A is updated directly by this
                     ; instruction.
NOP                  ; This conflict occurs because the
                     ; next instruction tries to read A
                     ; as a memory-mapped register. A
                     ; one-cycle latency required.
PSHM   AL            ; This instruction reads A as a
                     ; memory–mapped register.
```

Example 7–67. Accumulator Access With No Conflict

(a)

```
    ADD    Smem, A       ; A is updated directly by this
                         ; instruction. No conflict occurs
                         ; because the next instruction reads
                         ; accumulator A directly.
    NEG    A             ; This instruction reads A directly.
```

(b)

```
    STLM   A, BH         ; BH is written using memory-mapped
                         ; addressing here.
                         ; No conflict occurs because the
                         ; next instruction also accesses the
                         ; same accumulator as a memory-
                         ; mapped register.
    PSHM   BH            ; Reads BH as a memory–mapped
                         ; register.
```

(c)

```
    STLM   A, BH         ; BH is written using memory-mapped
                         ; addressing here.
                         ; No conflict occurs because the
                         ; next instruction accesses the same
                         ; accumulator directly.
    NEG    B             ; This instruction reads B directly.
```

Table 7–28 lists the latencies between instructions that update an accumulator directly and instructions that access the same accumulator as a memory-mapped register.

Note:

You are responsible for rearranging instructions or inserting NOPs, if necessary, to accommodate latencies.

Table 7–28. Latencies for Accumulators A and B When Used as Memory-Mapped Registers

(a) Latencies based on second-instruction category

First Instruction	Second Instruction Category I	Second Instruction Category II
All 1-word instructions that directly modify A or B without accessing them as memory-mapped registers	1	0
ADD Smem, shift†, src, dst LD Smem, shift†, dst SUB Smem, shift†, src, dst	1	0
All 2-word instructions that directly modify A or B without accessing them as memory-mapped registers	0	0

(b) Categories for the second instruction

Category I	Category II
All instructions that read an accumulator as a memory-mapped register (AG, AH, AL, BG, BH, and BL) without using a long-offset modifier.	All instructions that read an accumulator as memory-mapped register (AG, AH, AL, BG, BH, BL) using a long-offset modifier
	MVKD accum, Smem MVDM accum, MMR
	ADD accum, shift, src, dst LD accum, shift, dst SUB accum, shift, src, dst } With extended shift value of –16 to 15

Legend:
- MMR — Any memory-mapped register
- accum — Source operand pointing to AG, AH, AL, BG, BH, or BL using memory-mapped, direct, or indirect addressing modes
- † — Shift value between –16 and 15.

Example 7–68. Updating Accumulator With a 1-Cycle Latency

```
LD     Smem, –1, B      ; B is updated directly by this
                        ; instruction
NOP                     ; Conflict occurs because next
                        ; instruction tries to read B as
                        ; a memory-mapped register. A
                        ; one-cycle latency is required.
PSHM   BH               ; Reads BH as a memory-mapped
                        ; register.
```

Example 7–69. Updating Accumulator With No Latency

(a)

```
MAC    #K,A          ; A is updated directly by this
                     ; instruction. No latency is
                     ; required since MAC is a 2–word
                     ; instruction.
PSHM   AL            ; This instruction reads A as a
                     ; memory-mapped register.
```

(b)

```
ADD    Smem,A        ; A is updated directly by this
                     ; instruction. No latency is
                     ; required since the next
                     ; instruction uses a long offset
                     ; modifier.
LD     *(AL),ASM     ; This instruction reads A as a
                     ; memory–mapped register.
```

Chapter 8

On-Chip Peripherals

The on-chip peripherals for the ’54x are specific to the individual device. This chapter, along with Chapter 9, *Serial Ports*, and Chapter 10, *External Bus Operation*, describes all of the available on-chip peripherals, but your device may have only a subset of them.

All of the ’54x devices have general-purpose I/O pins, a timer, a clock generator, a software-programmable wait-state generator, and a programmable bank-switching module. Different types of serial ports, host port interface, and clock generator are device-specific peripherals. The serial ports are discussed in Chapter 9, *Serial Ports*, and the software-programmable wait-state generator and programmable bank-switching module are discussed in Chapter 10, *External Bus Operation*.

- ❑ General-purpose I/O pins: XF and $\overline{\text{BIO}}$
- ❑ Timer
- ❑ Clock generator
- ❑ Host port interface (’542, ’545, ’548, and ’549)
- ❑ Synchronous serial port (’541, ’545, and ’546)
- ❑ Buffered serial port (’542, ’543, ’545, ’546, ’548, and ’549)
- ❑ Time-division multiplexed (TDM) serial port (’542, ’543, ’548, and ’549)
- ❑ Software-programmable wait-state generator
- ❑ Programmable bank-switching module

Topic **Page**

8.1 Peripheral Memory-Mapped Registers

Peripherals are operated and controlled by accessing memory-mapped control and data registers. These registers can also transfer data to and from the peripherals. Setting and clearing bits in the control registers can enable, disable, initialize, and dynamically reconfigure the peripherals. The operations of the serial ports and the timer are synchronized to the CPU through interrupts or interrupt polling. When peripherals are not in use, the internal clocks are shut off; thus, the peripherals consume less power in normal run mode or in idle mode.

The peripheral registers are mapped into data page 0. Figure 8–1 shows the locations of the memory-mapped registers for each peripheral. Table 8–1 through Table 8–7 list the individual peripheral memory-mapped registers for each version of the ’54x. Note that all accesses to memory-mapped peripheral registers require two machine cycles.

Figure 8–1. TMS320C54x Peripheral Memory-Mapped Registers

	’541	’542	’543	’545	’546	’548/549
0020h	Synchronous serial port 0	Buffered serial port	Buffered serial port	Buffered serial port	Buffered serial port	Buffered serial port 0
0024h	Timer	Timer	Timer	Timer	Timer	Timer
0028h	External bus control	External bus control	External bus control	External bus control	External bus control	External bus control
002Ch	Reserved	Host port interface	Reserved	Host port interface	Reserved	Host port interface
0030h	Synchronous serial port 1	TDM serial port	TDM serial port	Synchronous serial port	Synchronous serial port	TDM serial port
0034h	Reserved			Reserved	Reserved	
0038h	Reserved	Autobuffering unit	Autobuffering unit	Autobuffering unit	Autobuffering unit	Autobuffering unit (BSP0)
003Ch	Reserved	Reserved	Reserved	Reserved	Reserved	Autobuffering unit (BSP1)
0040h	Reserved	Reserved	Reserved	Reserved	Reserved	Buffered serial port 1
0044h	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved
0048h	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved
004Ch	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved
0050h	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved
0054h	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved
0058h	Reserved	Reserved	Reserved	Reserved/Clock Mode†	Reserved/Clock Mode†	Clock Mode
005Ch	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved

† LP device only

Table 8–1. TMS320C541/541B Peripheral Memory-Mapped Registers

Address	Name	Description
20	DRR0	Serial port 0 data receive register
21	DXR0	Serial port 0 data transmit register
22	SPC0	Serial port 0 control register
23	–	Reserved
24	TIM	Timer register
25	PRD	Timer period register
26	TCR	Timer control register
27	–	Reserved
28	SWWSR	Software wait-state register
29	BSCR	Bank-switching control register
2A–2F	–	Reserved
30	DRR1	Serial port 1 data receive register
31	DXR1	Serial port 1 data transmit register
32	SPC1	Serial port 1 control register
33–57	–	Reserved
58	CLKMD	Clock mode register (’541B only)
59–57	–	Reserved

Table 8–2. TMS320C542 Peripheral Memory-Mapped Registers

Address	Name	Description
20	BDRR0	Buffered serial port data receive register
21	BDXR0	Buffered serial port data transmit register
22	BSPC0	Buffered serial port control register
23	BSPCE0	Buffered serial port control extension register
24	TIM	Timer register
25	PRD	Timer period register
26	TCR	Timer control register
27	–	Reserved
28	SWWSR	Software wait-state register
29	BSCR	Bank-switching control register
2A-2B	–	Reserved
2C	HPIC	Host port interface control register
2D-2F	–	Reserved
30	TRCV	TDM serial port data receive register
31	TDXR	TDM serial port data transmit register
32	TSPC	TDM serial port control register
33	TCSR	TDM serial port channel select register
34	TRTA	TDM serial port receive transmit register
35	TRAD	TDM serial port receive address register
36–37	–	Reserved
38	AXR0	ABU transmit address register
39	BKX0	ABU transmit buffer-size register
3A	ARR0	ABU receive address register
3B	BKR0	ABU receive buffer-size register
3C–5F	–	Reserved

Table 8–3. TMS320C543 Peripheral Memory-Mapped Registers

Address	Name	Description
20	BDRR0	Buffered serial port data receive register
21	BDXR0	Buffered serial port data transmit register
22	BSPC0	Buffered serial port control register
23	BSPCE0	Buffered serial port control extension register
24	TIM	Timer register
25	PRD	Timer period register
26	TCR	Timer control register
27	–	Reserved
28	SWWSR	Software wait-state register
29	BSCR	Bank-switching control register
2A-2F	–	Reserved
30	TRCV	TDM serial port data receive register
31	TDXR	TDM serial port data transmit register
32	TSPC	TDM serial port control register
33	TCSR	TDM serial port channel select register
34	TRTA	TDM serial port receive transmit register
35	TRAD	TDM serial port receive address register
36-37	–	Reserved
38	AXR0	ABU transmit address register
39	BKX0	ABU transmit buffer-size register
3A	ARR0	ABU receive address register
3B	BKR0	ABU receive buffer-size register
3C-5F	–	Reserved

Table 8–4. TMS320C545/545A Peripheral Memory-Mapped Registers

Address	Name	Description
20	BDRR0	Buffered serial port data receive register
21	BDXR0	Buffered serial port data transmit register
22	BSPC0	Buffered serial port control register
23	BSPCE0	Buffered serial port control extension register
24	TIM	Timer register
25	PRD	Timer period register
26	TCR	Timer control register
27	–	Reserved
28	SWWSR	Software wait-state register
29	BSCR	Bank-switching control register
2A-2B	–	Reserved
2C	HPIC	Host port interface control register
2D-2F	–	Reserved
30	DRR1	Serial port data receive register
31	DXR1	Serial port data transmit register
32	SPC1	Serial port control register
33-37	–	Reserved
38	AXR0	ABU transmit address register
39	BKX0	ABU transmit buffer-size register
3A	ARR0	ABU receive address register
3B	BKR0	ABU receive buffer-size register
3C-57	–	Reserved
58	CLKMD	Clock mode register (’545A only)
59-5F	–	Reserved

Table 8–5. TMS320C546/546A Peripheral Memory-Mapped Registers

Address	Name	Description
20	BDRR0	Buffered serial port data receive register
21	BDXR0	Buffered serial port data transmit register
22	BSPC0	Buffered serial port control register
23	BSPCE0	Buffered serial port control extension register
24	TIM	Timer register
25	PRD	Timer period register
26	TCR	Timer control register
27	–	Reserved
28	SWWSR	Software wait-state register
29	BSCR	Bank-switching control register
2A-2F	–	Reserved
30	DRR1	Serial port data receive register
31	DXR1	Serial port data transmit register
32	SPC1	Serial port control register
33-37	–	Reserved
38	AXR0	ABU transmit address register
39	BKX0	ABU transmit buffer-size register
3A	ARR0	ABU receive address register
3B	BKR0	ABU receive buffer-size register
3C-57	–	Reserved
58	CLKMD	Clock mode register (’546A only)
59-5F	–	Reserved

Table 8–6. TMS320C548 Peripheral Memory-Mapped Registers

Address	Name	Description
20	BDRR0	Buffered serial port 0 data receive register
21	BDXR0	Buffered serial port 0 data transmit register
22	BSPC0	Buffered serial port 0 control register
23	BSPCE0	Buffered serial port 0 control extension register
24	TIM	Timer register
25	PRD	Timer period register
26	TCR	Timer control register
27	–	Reserved
28	SWWSR	Software wait-state register
29	BSCR	Bank-switching control register
2A-2B	–	Reserved
2C	HPIC	Host port interface control register
2D-2F	–	Reserved
30	TRCV	TDM serial port data receive register
31	TDXR	TDM serial port data transmit register
32	TSPC	TDM serial port control register
33	TCSR	TDM serial port channel select register
34	TRTA	TDM serial port receive transmit register
35	TRAD	TDM serial port receive address register
36-37	–	Reserved
38	AXR0	ABU 0 transmit address register
39	BKX0	ABU 0 transmit buffer-size register
3A	ARR0	ABU 0 receive address register
3B	BKR0	ABU 0 receive buffer-size register
3C	AXR1	ABU 1 transmit address register
3D	BKX1	ABU 1 transmit buffer-size register
3E	ARR1	ABU 1 receive address register
3F	BKR1	ABU 1 receive buffer-size register
40	BDRR1	Buffered serial port 1 data receive register
41	BDXR1	Buffered serial port 1 data transmit register
42	BSPC1	Buffered serial port 1 control register
43	BSPCE1	Buffered serial port 1 control extension register
44-57	–	Reserved
58	CLKMD	Clock-mode register
59-5F	–	Reserved

Table 8–7. TMS320C549 Peripheral Memory-Mapped Registers

Address	Name	Description
20	BDRR0	Buffered serial port 0 data receive register
21	BDXR0	Buffered serial port 0 data transmit register
22	BSPC0	Buffered serial port 0 control register
23	BSPCE0	Buffered serial port 0 control extension register
24	TIM	Timer count register
25	PRD	Timer period register
26	TCR	Timer control register
27	–	Reserved
28	SWWSR	External interface software wait-state register
29	BSCR	External interface bank-switching control register
2A	–	Reserved
2B	XSWR	Extended software wait-state register
2C	HPIC	Host port interface control register
2D-2F	–	Reserved
30	TRCV	TDM serial port data receive register
31	TDXR	TDM serial port data transmit register
32	TSPC	TDM serial port control register
33	TCSR	TDM serial port channel select register
34	TRTA	TDM serial port receive transmit register
35	TRAD	TDM serial port receive address register
36-37	–	Reserved
38	AXR0	ABU 0 transmit address register
39	BKX0	ABU 0 transmit buffer-size register
3A	ARR0	ABU 0 receive address register
3B	BKR0	ABU 0 receive buffer-size register
3C	AXR1	ABU 1 transmit address register
3D	BKX1	ABU 1 transmit buffer-size register
3E	ARR1	ABU 1 receive address register
3F	BKR1	ABU 1 receive buffer-size register
40	BDRR1	Buffered serial port 1 data receive register
41	BDXR1	Buffered serial port 1 data transmit register
42	BSPC1	Buffered serial port 1 control register
43	BSPCE1	Buffered serial port 1 control extension register
44-57	–	Reserved
58	CLKMD	Clock-mode register
59-5F	–	Reserved

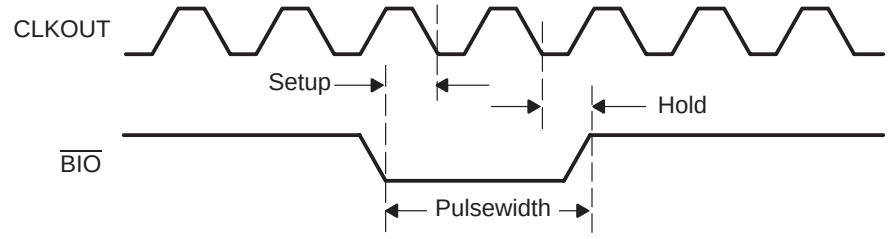
8.2 General-Purpose I/O

The ’54x offers general-purpose I/O through its I/O memory space and through two dedicated pins that are software controlled. The two dedicated pins are the branch control input pin ($\overline{\text{BIO}}$) and the external flag output pin (XF). For more information about the I/O memory space and accesses using the external bus, see Section 3.4, *I/O Memory*, on page 3-22, and Chapter 10, *External Bus Operation*.

8.2.1 Branch Control Input Pin ($\overline{\text{BIO}}$)

$\overline{\text{BIO}}$ can be used to monitor the status of peripheral devices. It is especially useful as an alternative to using an interrupt when time-critical loops must not be disturbed. A branch can be conditionally executed dependent upon the state of the $\overline{\text{BIO}}$ input. The timing diagram, shown in Figure 8–2, shows the operation of $\overline{\text{BIO}}$ (refer to the ’54x data sheet for timing specifications). Of the instructions that use $\overline{\text{BIO}}$, the execute conditionally (XC) instruction samples the condition of $\overline{\text{BIO}}$ during the decode phase of the pipeline; all other conditional instructions (branch, call, and return) sample $\overline{\text{BIO}}$ during the read phase of the pipeline.

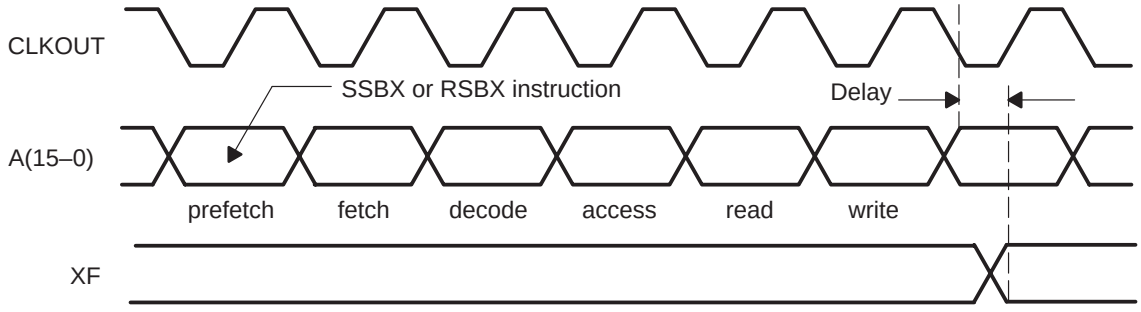
Figure 8–2. $\overline{\text{BIO}}$ Timing Diagram



8.2.2 External Flag Output Pin (XF)

XF can be used to signal external devices. The XF pin is controlled using software. It is driven high by setting the XF bit (in ST1) and is driven low by clearing the XF bit. The set status register bit (SSBX) and reset status register bit (RSBX) instructions set and clear XF, respectively. XF is also set high at device reset. Figure 8–3 shows the relationship between the time the SSBX or RSBX instruction is fetched and the time the XF pin is set or reset (refer to the ’54x data sheet for timing specifications). The XF timing shown is for a sequence of single-cycle instructions. Actual timing can vary with different instruction sequences.

Figure 8–3. External Flag Timing Diagram



8.3 Timer

The on-chip timer is a software-programmable timer that consists of three registers and that can be used to periodically generate interrupts. The timer resolution is the CLKOUT rate of the device, and it has up to a 20-bit dynamic range.

8.3.1 Timer Registers

The on-chip timer consists of three memory-mapped registers (TIM, PRD, and TCR). These three registers are listed in Table 8–8.

Table 8–8. Timer Registers

Address	Register	Description
0024h	TIM	Timer register
0025h	PRD	Timer period register
0026h	TCR	Timer control register

- Timer register (TIM). The 16-bit memory-mapped timer register (TIM) is loaded with the period register (PRD) value and decremented.
- Timer period register (PRD). The 16-bit memory-mapped timer period register (PRD) is used to reload the timer register (TIM).
- Timer control register (TCR). The 16-bit memory-mapped timer control register (TCR) contains the control and status bits of the timer. The TCR bit fields are shown in Figure 8–4 and described in Table 8–9.

Figure 8–4. Timer Control Register (TCR) Diagram

Table 8–9. Timer Control Register (TCR) Bit Summary

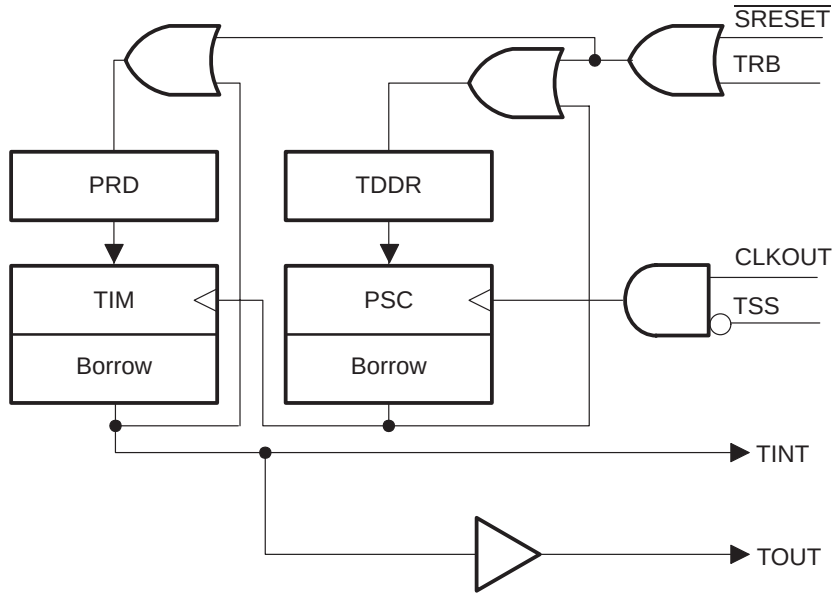
Bit	Name	Reset Value	Function
15–12	Reserved	—	Reserved; always read as 0.
11	Soft	0	Used in conjunction with the Free bit to determine the state of the timer when a breakpoint is encountered in the HLL debugger. When the Free bit is cleared, the Soft bit selects the timer mode. Soft = 0 The timer stops immediately. Soft = 1 The timer stops when the counter decrements to 0.
10	Free	0	Used in conjunction with the Soft bit to determine the state of the timer when a breakpoint is encountered in the HLL debugger. When the Free bit is cleared, the Soft bit selects the timer mode. Free = 0 The Soft bit selects the timer mode. Free = 1 The timer runs free regardless of the Soft bit.
9–6	PSC	—	Timer prescaler counter. Specifies the count for the on-chip timer. When PSC is decremented past 0 or the timer is reset, PSC is loaded with the contents of TDDR and the TIM is decremented.
5	TRB	—	Timer reload. Resets the on-chip timer. When TRB is set, the TIM is loaded with the value in the PRD and the PSC is loaded with the value in TDDR. TRB is always read as a 0.
4	TSS	0	Timer stop status. Stops or starts the on-chip timer. At reset, TSS is cleared and the timer immediately starts timing. TSS = 0 The timer is started. TSS = 1 The timer is stopped.
3–0	TDDR	0000	Timer divide-down ratio. Specifies the timer divide-down ratio (period) for the on-chip timer. When PSC is decremented past 0, PSC is loaded with the contents of TDDR.

8.3.2 Timer Operation

The timer is an on-chip down-counter that can be used to periodically generates CPU interrupts. The timer is driven by a prescaler that is decremented by 1 at every CLKOUT cycle. Each time the counter decrements to 0, a timer interrupt (TINT) is generated and the down-counter is reloaded with the period value. See Section 6.10, *Interrupts*, on page 6-26, for more details about interrupts.

Figure 8–5 shows a logical block diagram of the timer. It consists of two basic blocks: the main timer block, consisting of PRD and TIM; and a prescaler block, consisting of the TDDR and PSC bits in TCR. The timer is clocked by CLKOUT of the device.

Figure 8–5. Timer Block Diagram



Under normal operation, TIM is loaded with the contents of PRD when TIM decrements to 0. The contents of PRD are also loaded into TIM when the device is reset ($\overline{\text{SRESET}}$ input in Figure 8–5) or when the timer is individually reset (TRB input in Figure 8–5). TIM is clocked by the prescaler block. Each output clock from the prescaler block decrements TIM by 1. The output of the main timer block is the timer interrupt (TINT) signal that is sent to the CPU and to the timer output (TOUT) pin. The duration of the TOUT pulse is equal to the period of CLKOUT.

The prescaler block has two elements similar to the TIM and PRD. These are the prescale counter (PSC) and timer divide-down ratio (TDDR). Both PSC and TDDR are fields in the timer control register (TCR). Under normal operation, PSC is loaded with the contents of TDDR when PSC decrements to 0. The contents of TDDR are also loaded into PSC when the device is reset or when the timer is individually reset. PSC is clocked by the device output clock, CLKOUT. Each CLKOUT decrements PSC by 1. PSC can be read by reading TCR, but it cannot be written to directly.

The timer can be stopped by making use of the TSS input to turn off the clock input to the timer. Stopping the timer's operation allows the device to run in a low-power mode when the timer is not needed.

The timer interrupt (TINT) rate is equal to the CLKOUT frequency divided by two independent factors:

$$\text{TINT rate} = \frac{1}{t_{c(C)} \times u \times v} = \frac{1}{t_{c(C)} \times (\text{TDDR} + 1) \times (\text{PRD} + 1)}$$

In the equation, $t_{c(C)}$ is the period of CLKOUT, u is the sum of the TDDR contents plus 1, and v is the sum of the PRD contents plus 1.

The current value in the timer can be read by reading TIM; PSC can be read by reading TCR. Because it takes two instructions to read both registers, there may be a change between the two reads as the counter decrements. Therefore, when precise timing measurements are needed, it is more accurate to stop the timer before reading these two values. The timer can be stopped by setting the TSS bit and restarted by clearing it.

The timer can be used to generate a sample clock for peripheral circuits such as an analog interface. This can be accomplished by using the TOUT signal to clock a device or by making use of the interrupt to periodically read a register.

The timer is initialized with the following steps:

1) Stop the timer by writing a 1 to TSS in TCR.
2) Load PRD.
3) Start the timer by reloading TCR to initialize TDDR, enable CLKOUT to the timer by setting TSS to 0 and TRB to 1 to reload the timer period.

Optionally, the timer interrupt may be enabled by (assuming INTM = 1):

1) Clearing any pending timer interrupts by writing a 1 to TINT in the IFR.
2) Enabling the timer interrupt by writing a 1 to TINT in the IMR.
3) Enabling interrupts globally, if necessary, by clearing INTM to 0.

At reset, TIM and PRD are set to a maximum value of FFFFh. The timer divide-down ratio (TDDR) field of the TCR is cleared to 0 and the timer is started.

8.4 Clock Generator

The clock generator allows system designers to select the clock source. The sources that drive the clock generator are:

- A crystal resonator with the internal oscillator circuit. The crystal resonator circuit is connected across the X1 and X2/CLKIN pins of the ’54x to enable the internal oscillator.
- An external clock. The external clock source is directly connected to the X2/CLKIN pin, and X1 is left unconnected.

The clock generator on the ’54x devices consists of an internal oscillator and a phase-locked loop (PLL) circuit. Currently, there are two different types of PLL circuits on ’54x devices. Some devices have hardware-configurable PLL circuits while others have software-programmable PLL circuits. The ’541, ’542, ’543, ’545, and ’546 devices use a hardware-configurable PLL. The ’545A, ’546A, ’548 and ’549 devices use a software-programmable PLL and are referred to in this section as *LP* devices. Refer to Section C.2, *Part Order Information*, on page C-5.

8.4.1 Hardware-Configurable PLL

The PLL functions with a lower external frequency source than the machine cycle rate of the CPU. This feature reduces high-frequency noise from a high-speed switching clock. The internal oscillator or the external clock source is fed into the PLL. The internal CPU clock is generated by multiplying the external clock source or the internal oscillator frequency by a factor N (PLL $\times$ N). If you are using the internal oscillator circuit, the clock source is divided by 2 to generate the internal CPU clock. If you are using the external clock, the internal CPU clock is a factor of PLL $\times$ N.

The PLL has a maximum operating frequency of 40 MHz on a 25-ns ’54x device. The PLL requires a transitory locking time of 50 μs. The locking time is necessary during reset and recovery from the IDLE3 power-down mode. See Section 6.11, *Power-Down Modes*, on page 6-45, and subsection 10.5.2, *IDLE3*, on page 10-25, for more information.

The clock mode is determined by the CLKMD1, CLKMD2, and CLKMD3 pins. Table 8–10 shows how these pins select the clock mode. For non-PLL use, the frequency of the CPU clock is half the crystal’s oscillating frequency or the external clock frequency.

The clock mode must not be reconfigured with the clock mode pins during normal operation. During IDLE3 mode, the clock mode can be reconfigured after CLKOUT is set high.

Table 8–10. Clock Mode Configurations

Mode Select Pins			Clock Mode†	
CLKMD1	CLKMD2	CLKMD3	Option 1	Option 2
0	0	0	PLL × 3 with external source	PLL × 5 with external source
1	1	0	PLL × 2 with external source	PLL × 4 with external source
1	0	0	PLL × 3 with internal source	PLL × 5 with internal source
0	1	0	PLL × 1.5 with external source	PLL × 4.5 with external source
0	0	1	Divide-by-2 with external source	Divide-by-2 with external source
1	1	1	Divide-by-2 with internal source	Divide-by-2 with internal source
1	0	1	PLL × 1 with external source	PLL × 1 with external source
0	1	1	Stop mode‡	Stop mode‡

† An individual device is either an *Option 1* or *Option 2* clock-mode device.
‡ The PLL is disabled. The system clock is not provided to CPU/peripherals. The function of the stop mode is equivalent to that of the power-down mode of IDLE3; however, the IDLE 3 instruction is recommended rather than stop mode to realize full power saving, since IDLE3 stops clocks synchronously and can be exited with an interrupt.

8.4.2 Software-Programmable PLL (TMS320C541B/’545A/’546A/’548/’549)

The software-programmable PLL features a high level of flexibility, and includes a clock scaler that provides various clock multiplier ratios, capability to directly enable and disable the PLL, and a PLL lock timer that can be used to delay switching to PLL clocking mode of the device until lock is achieved.

Devices that have a built-in software-programmable PLL can be configured in one of two clock modes:

- PLL mode. The input clock (CLKIN) is multiplied by 1 of 31 possible ratios from 0.25 to 15. These ratios are achieved using the PLL circuitry.
- DIV (divider) mode. The input clock (CLKIN) is divided by 2 or 4. When DIV mode is used, all of the analog parts, including the PLL circuitry, are disabled in order to minimize power dissipation.

Immediately following reset, the clock mode is determined by the values of the three external pins, CLKMD1, CLKMD2, and CLKMD3. The modes corresponding to the CLKMD pins are shown in Table 8–11.

Table 8–11. Clock Mode Settings at Reset

CLKMD1	CLKMD2	CLKMD3	CLKMD Reset Value	Clock Mode
0	0	0	0000h	Divide-by-2 with external source
0	0	1	1000h	Divide-by-2 with external source
0	1	0	2000h	Divide-by-2 with external source
1	0	0	4000h	Divide-by-2 with internal source
1	1	0	6000h	Divide-by-2 with external source
1	1	1	7000h	Divide-by-2 with internal source†
1	0	1	0007h	PLL × 1 with external source
0	1	1	—	Stop mode

† Reserved on '549

Following reset, the software-programmable PLL can be programmed to any configuration desired. When the PLL × 1 with external source option, CLKMD(1–3) = 101, is selected during reset, the internal PLL lock-count timer is not active; therefore, the system must delay releasing reset in order to allow for the PLL lock-time delay.

The programming of the PLL is loaded in the 16-bit memory-mapped (address 58h) clock mode register (CLKMD). The CLKMD is used to define the clock configuration of the PLL clock module. The CLKMD bit fields are shown in Figure 8–6 and described in Table 8–12. Note that upon reset, the CLKMD is initialized with a predetermined value dependent only upon the state of the CLKMD(1–3) pins (see Table 8–11).

Figure 8–6. Clock Mode Register (CLKMD) Diagram

15–12	11	10–3	2	1	0
PLLMUL	PLLDIV	PLLCOUNT	PLLON/OFF	PLLNDIV	PLLSTATUS
R/W†	R/W†	R/W†	R/W†	R/W	R

† When in DIV mode (PLLSTATUS is low), PLLMUL, PLLDIV, PLLCOUNT, and PLLON/OFF are don't cares, and their contents are indeterminate.

Table 8–12. Clock Mode Register (CLKMD) Bit Summary

Bit	Name	Function
15–12	PLLMUL	PLL multiplier. Defines the frequency multiplier in conjunction with PLLDIV and PLLNDIV, as shown in Table 8–13 on page 8-21.
11	PLLDIV	PLL divider. Defines the frequency multiplier in conjunction with PLLMUL and PLLNDIV, as shown in Table 8–13 on page 8-21.
10–3	PLLCOUNT	PLL counter value. Specifies the number of input clock cycles (in increments of 16 cycles) for the PLL lock timer to count before the PLL begins clocking the processor after the PLL is started. The PLL counter is a down-counter, which is driven by the input clock divided by 16; therefore, for every 16 input clocks, the PLL counter decrements by 1. See subsection *Using the PLLCOUNT Programmable Lock Timer*, on page 8-21 for more information about PLLCOUNT. The PLL counter can be used to ensure that the processor is not clocked until the PLL is locked, so that only valid clock signals are sent to the device.
2	PLLON/OFF	PLL on/off. Enables or disables the PLL part of the clock generator in conjunction with PLLNDIV. PLLON/OFF and PLLNDIV both force the PLL to operate; when PLLON/OFF is high, the PLL runs independently of the state of PLLNDIV:

PLLON/OFF	PLLNDIV	PLL State
0	0	off
0	1	on
1	0	on
1	1	on

1	PLLNDIV	PLL clock generator select. Determines whether the clock generator works in PLL mode or in divider (DIV) mode, thus defining the frequency multiplier in conjunction with PLLMUL and PLLDIV. PLLNDIV = 0 Divider (DIV) mode is used. PLLNDIV = 1 PLL mode is used.
0	PLLSTATUS	PLL status. Indicates the mode that the clock generator is operating. PLLSTATUS = 0 Divider (DIV) mode PLLSTATUS = 1 PLL mode

Table 8–13. PLL Multiplier Ratio as a Function of PLLNDIV, PLLDIV, and PLLMUL

PLLNDIV	PLLDIV	PLLMUL	Multiplier†
0	x	0 – 14	0.5
0	x	15	0.25
1	0	0 – 14	PLLMUL + 1
1	0	15	1
1	1	0 or even	(PLLMUL + 1) ÷ 2
1	1	odd	PLLMUL ÷ 4

† CLKOUT = CLKIN × Multiplier

Programming Considerations When Using the Software-Programmable PLL

The software-programmable PLL offers many different options in startup configurations, operating modes, and power-saving features. Programming considerations and several software examples are presented here to illustrate the proper use of the software-programmable PLL at start-up, when switching between different clocking modes, and before and after IDLE 1/IDLE 2/IDLE 3 instruction execution.

Using the PLLCOUNT Programmable Lock Timer

During the lockup period, the PLL should not be used to clock the ’54x. The PLLCOUNT programmable lock timer provides a convenient method of automatically delaying clocking of the device by the PLL until lock is achieved.

The PLL lock timer is a counter, loaded from the PLLCOUNT field in the CLKMD register, that decrements from its preset value to 0. The timer can be preset to any value from 0 to 255, and its input clock is CLKIN divided by 16. The resulting lockup delay can therefore be set from 0 to 255 × 16 CLKIN cycles.

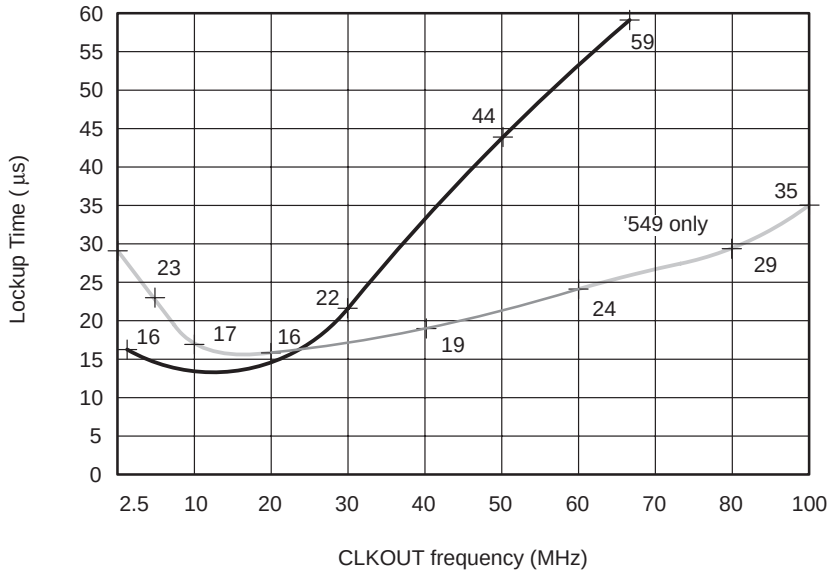
The lock timer is activated when the clock generator operating mode is switched from DIV to PLL (see subsection *Switching From DIV Mode to PLL Mode*, on page 8-22). During the lockup period, the clock generator continues to operate in DIV mode; after the PLL lock timer has decremented to 0, the PLL begins clocking the ’54x.

The decimal preset value, PLLCOUNT, is:

$$PLLCOUNT > \frac{LockupTime}{16 \times T_{CLKIN}}$$

where T_{CLKIN} is the input reference clock period and LockupTime is the required PLL lockup time as shown in Figure 8–7.

Figure 8–7. PLL Lockup Time Versus CLKOUT Frequency



Switching Clock Mode From DIV Mode to PLL Mode

Several circumstances may require switching from DIV mode to PLL mode; however, note that if the PLL is not locked when switching from DIV mode to PLL mode, the PLL lockup time delay must be observed before the mode switch occurs to ensure that only proper clock signals are sent to the device. It is, therefore, important to know whether or not the PLL is locked when switching operating modes.

The PLL is unlocked on power-up, after changing the PLLMUL or PLLDIV values, after turning off the PLL (PLLON/OFF = 0), or after loss of input reference clock. Once locked, the PLL remains locked even in DIV mode as long as the PLL had been previously locked and has not been turned off (PLLON/OFF stays 1), and the PLLMUL and PLLDIV values have not been changed since the PLL was locked.

Switching from DIV mode to PLL mode (setting PLLNDIV to 1) activates the PLLCOUNT programmable lock timer (when PLLCOUNT is preloaded with a nonzero value) and this can be used to provide a convenient method for implementing the lockup time delay. The PLLCOUNT lock timer feature should be used in the previously described situations where the PLL is unlocked unless a reset delay is used to implement the lockup delay, or the PLL is not used.

Switching from DIV mode to PLL mode is accomplished by loading CLKMD. The following procedure describes switching from DIV mode to PLL mode

when the PLL is not locked. When performing this mode switch with the PLL already locked, the effect is the same as when switching from PLL mode to DIV mode, but in the reverse order. In this case, the delays of when the new clock mode takes effect are the same.

When switching from DIV mode to PLL mode with the PLL unlocked, or when the mode change will result in unlocked operation, the PLLMUL, PLLDIV, and PLLNDIV bits are set to select the desired frequency multiplier as shown in Table 8–13 on page 8-21, and the PLLCOUNT bits are set to select the required lockup time delay. Note that PLLMUL, PLLDIV, PLLCOUNT, and PLLON/OFF can only be modified when in DIV mode.

Once the PLLNDIV bit is set, the PLLCOUNT timer begins being decremented from its preset value. When the PLLCOUNT timer reaches 0, the switch to PLL mode takes effect after 6 CLKIN cycles plus 3.5 PLL cycles (CLKOUT frequency). When the switch to PLL mode is completed, the PLLSTATUS bit in CLKMD is read as 1. Note that during the PLL lockup period, the ’54x continues operating in DIV mode.

The following code can be used to switch from DIV mode to PLL × 3 mode, with a CLKIN frequency of 13 MHz and PLLCOUNT = 41 (decimal):

```
STM    #0010000101001111b, CLKMD
```

Switching Clock Mode From PLL Mode to DIV Mode

When switching from PLL mode to DIV mode, the PLLCOUNT delay does not occur and the switch between the two modes takes place after a short transition delay.

The switch from PLL mode to DIV mode is also accomplished by loading CLKMD. The PLLNDIV bit is cleared to 0, selecting DIV mode, and the PLLMUL bits are set to select the desired frequency multiplier as shown in Table 8–13 on page 8-21.

The switch to DIV mode takes effect in 6 CLKIN cycles plus 3.5 PLL cycles (CLKOUT frequency) for all PLLMUL values except 1111b. For a PLLMUL value of 1111b, the switch to DIV mode takes effect in 12 CLKIN cycles plus 3.5 PLL cycles (CLKOUT frequency). When the switch to DIV mode is completed, the PLLSTATUS bit in CLKMD is read as 0.

Example 8–1 shows a code sequence that can be used to switch from PLL × 3 mode to divide-by-2 mode. Note that the PLLSTATUS bit is polled to determine when the switch to DIV mode has taken effect, and then the STM instruction is used to turn off the PLL at this point.

Example 8–1. Switching Clock Mode From PLL × 3 Mode to Divide-by-2 Mode

```
          STM    #0b, CLKMD    ;switch to DIV mode
TstStatu: LDM    CLKMD, A
          AND    #01b, A       ;poll STATUS bit
          BC     TstStatu, ANEQ
          STM    #0b, CLKMD    ;reset PLLON/OFF when STATUS
                               ;is DIV mode
```

Changing the PLL Multiplier Ratio

When switching from one PLL multiplier ratio to another multiplier ratio is required, the clock generator must first be switched from PLL mode to DIV mode before selecting the new multiplier ratio; switching directly from one PLL multiplier ratio to another multiplier ratio is not supported.

In order to switch from one PLL multiplier ratio to another multiplier ratio, the following steps must be followed:

1) Clear the PLLNDIV bit to 0, selecting DIV mode.

2) Poll the PLLSTATUS bit until a 0 is obtained, indicating that DIV mode is enabled.

3) Modify CLKMD to set the PLLMUL, PLLDIV, and PLLNDIV bits to the desired frequency multiplier as shown in Table 8–13 on page 8-21

4) Set the PLLCOUNT bits to the required lock-up time.

Once the PLLNDIV bit is set, the PLLCOUNT timer begins being decremented from its preset value. When the PLLCOUNT timer reaches 0, the new PLL mode takes effect after 6 CLKIN cycles plus 3.5 PLL cycles (CLKOUT frequency).

Note that a direct switch between divide-by-2 mode and divide-by-4 mode is not possible. To switch between these two modes, the clock generator must first be set to PLL mode with an integer-only (nonfractional) multiplier ratio and then set back to DIV mode in the desired divider configuration (see subsection *Switching From DIV Mode to PLL Mode*, on page 8-22).

Example 8–2 shows a code sequence that can be used to switch the clock mode from PLL × X mode to PLL × 1 mode.

Example 8–2. Switching Clock Mode From PLL × X Mode to PLL × 1 Mode

```
          STM    #0b, CLKMD              ;switch to DIV mode
TstStatu: LDM    CLKMD, A
          AND    #01b, A                 ;poll STATUS bit
          BC     TstStatu, ANEQ
          STM #0000001111101111b, CLKMD  ;switch to PLL × 1 mode
```

PLL Operation Immediately Following Reset

Immediately following reset, the clock mode is determined by the values of the three external pins, CLKMD1, CLKMD2, and CLKMD3 as shown in Table 8–11 on page 8-19. All but two of these operating modes are divide-by-2 with external source. Switching from divide-by-2 mode to a PLL mode can easily be accomplished by changing the contents of CLKMD. Note that if use of the internal oscillator is desired, either the CLKMD(1–3) = 100 or the CLKMD(1–3) = 111 must be selected at reset (as shown in Table 8–11) since the internal oscillator cannot be programmed using software.

The following code can be used to switch from divide-by-2 mode to PLL × 3 mode:

```
STM    #0010000101001111b, CLKMD
```

PLL Considerations When Using IDLE Instruction

When using one of the IDLE instructions to reduce power requirements, proper management of the PLL is important. The clock generator consumes the least power when operating in DIV mode with the PLL disabled. Therefore, if power dissipation is a significant consideration, it is desirable to switch from PLL mode to DIV mode and disable the PLL, before executing an IDLE 1, IDLE 2, or IDLE 3 instruction. This is accomplished as explained in subsection *Switching From PLL Mode to DIV Mode*, on page 8-23. After waking up from IDLE1/IDLE2/IDLE3, the clock generator can be reprogrammed to PLL mode as explained in subsection *Switching From DIV Mode to PLL Mode*, on page 8-22.

Note that when the PLL is stopped during an IDLE state and the ’54x device is restarted and the clock generator is switched back to PLL mode, the PLL lockup delay occurs in the same manner as in a normal device startup. Therefore, in this case, the lockup delay must also be accounted for, either externally or by using the PLL lockup counter timer.

Example 8–3 shows a code sequence that switches the clock generator from PLL × 3 mode to divide-by-2 mode, turns off the PLL, and enters IDLE3. After waking up from IDLE3, the clock generator is switched from DIV mode to PLL × 3 mode using a single STM instruction, with a PLLCOUNT of 64 (decimal) used for the lock timer value.

Example 8–3. Switching Clock From PLL × 3 Mode to Divide-by-2 Mode, Turning Off the PLL, and Entering IDLE3

```
          STM    #0b, CLKMD                ;switch to DIV mode
TstStatu: LDM    CLKMD, A
          AND    #01b, A                   ;poll STATUS bit
          BC     TstStatu, ANEQ
          STM    #0b, CLKMD                ;reset PLLON_OFF when STATUS
                                           ;is DIV mode
          IDLE3

   (After IDLE3 wake-up – switch the PLL from DIV mode to PLL × 3 mode)

          STM    #0010001000000111b, CLKMD  ;PLLCOUNT = 64 (decimal)
```

PLL Considerations When Using the Bootloader

The ROM on the ’545A and ’546A contains a bootloader program that can be used to load programs into RAM for execution following reset. When using this bootloader with the software-programmable PLL, several considerations are important for proper system operation.

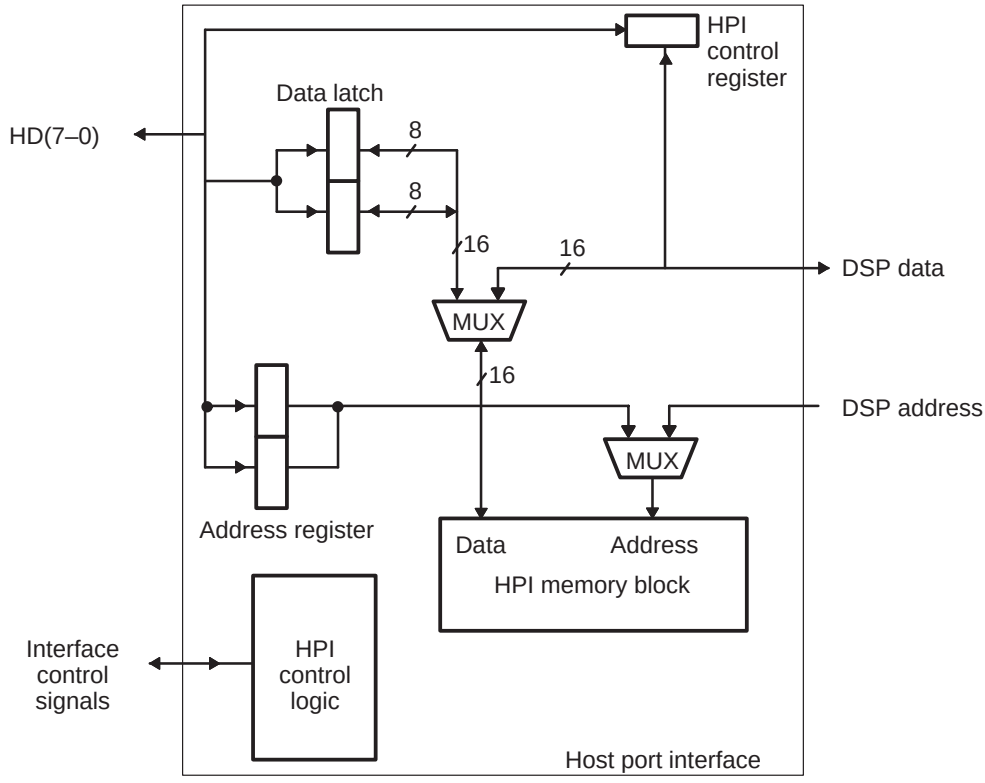
On the ’545A and ’546A, for compatibility, the bootloader configures the PLL to the same mode as would have resulted if the same CLKMD(1–3) input bits had been provided to the option-1 or option-2 hardware-programmable PLL (see Table 8–10 on page 8-18), according to whether the ’545A or ’546A is an option-1 or option-2 device. Once the bootloader program has finished executing and control is transferred to the user’s program, the PLL can be reprogrammed to any desired configuration.

8.5 Host Port Interface

The host port interface (HPI) is available on the ’542, ’545, ’548 and ’549 devices. The HPI is an 8-bit parallel port that interfaces a host device or host processor to the ’54x. Information is exchanged between the ’54x and the host device through on-chip ’54x memory that is accessible by both the host and the ’54x.

The HPI interfaces to the host device as a peripheral, with the host device as master of the interface, facilitating ease of access by the host. The host device communicates with the HPI through dedicated address and data registers, to which the ’54x does not have direct access, and the HPI control register, using the external data and interface control signals (see Figure 8–8). Both the host device and the ’54x have access to the HPI control register.

Figure 8–8. Host Port Interface Block Diagram



The HPI provides 16-bit data to the ’54x while maintaining the economical 8-bit external interface by automatically combining successive bytes transferred into 16-bit words. When the host device performs a data transfer with the HPI registers, the HPI control logic automatically performs an access to a dedicated 2K-word block of internal ’54x dual-access RAM to complete the transaction. The ’54x can then access the data within its memory space. The HPI RAM can also be used as general-purpose dual-access data or program RAM.

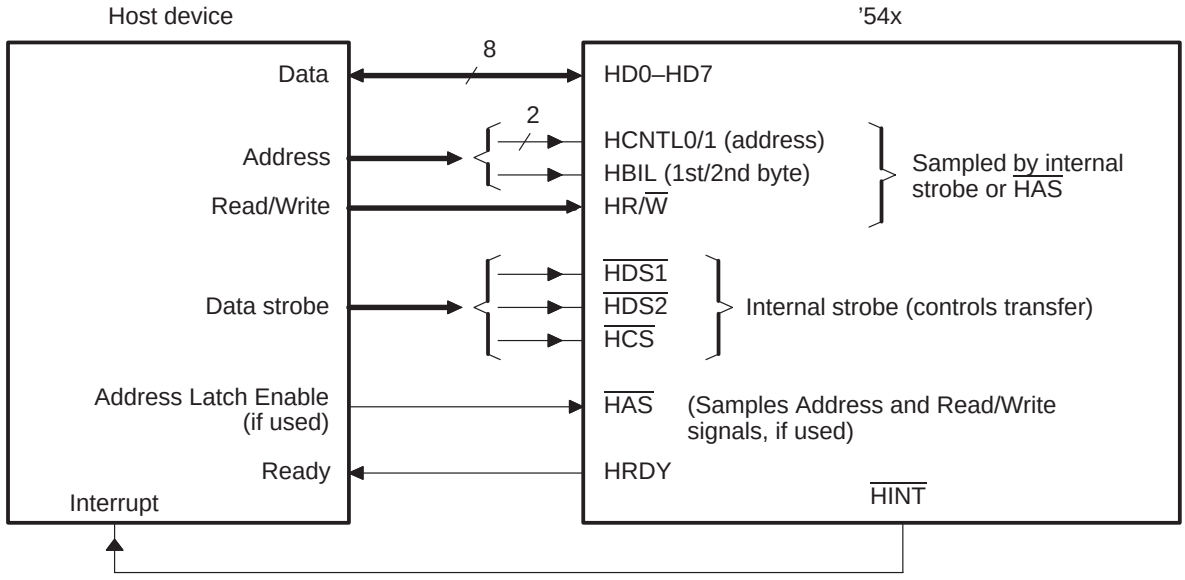
The HPI has two modes of operation, shared-access mode (SAM) and host-only mode (HOM). In shared-access mode (the normal mode of operation), both the ’54x and the host can access HPI memory. In this mode, asynchronous host accesses are resynchronized internally and, in the case of a conflict between a ’54x and a host cycle (where both accesses are reads or writes), the host has access priority and the ’54x waits one cycle. In host-only mode, only the host can access HPI memory while the ’54x is in reset or in IDLE2 with all internal and external clocks stopped. The host can therefore access the HPI RAM while the ’54x is in its minimum power consumption configuration.

The HPI supports high speed, back-to-back host accesses. In shared-access mode, the HPI can transfer one byte every five CLKOUT cycles (that is, 64M bps) with the ’54x running at a 40-MHz CLKOUT. The HPI is designed so the host can take advantage of this high bandwidth and run at frequencies up to (Fd*n)/5, where Fd is the ’54x CLKOUT frequency and n is the number of host cycles for an external access. Therefore, with a 40-MHz ’54x and common values of 4 (or 3) for n, the host can run at speeds of up to 32 (or 24) MHz without requiring wait states. In the host-only mode, the HPI supports even higher speed back-to-back host accesses on the order of one byte every 50 ns (that is, 160M bps), independent of the ’54x clock rate (refer to the TMS320C54x data sheet for specific detailed timing information).

8.5.1 Basic Host Port Interface Functional Description

The external HPI interface consists of the 8-bit HPI data bus and control signals that configure and control the interface. The interface can connect to a variety of host devices with little or no additional logic necessary. Figure 8–9 shows a simplified diagram of a connection between the HPI and a host device.

Figure 8–9. Generic System Block Diagram



The 8-bit data bus (HD0–HD7) exchanges information with the host. Because of the 16-bit word structure of the '54x, all transfers with a host must consist of two consecutive bytes. The dedicated HBIL pin indicates whether the first or second byte is being transferred. An internal control register bit determines whether the first or second byte is placed into the most significant byte of a 16-bit word. The host must not break the first byte/second byte (HBIL low/high) sequence of an ongoing HPI access. If this sequence is broken, data can be lost, and unpredictable operation can result.

The two control inputs (HCNTL0 and HCNTL1) indicate which internal HPI register is being accessed and the type of access to the register. These inputs, along with HBIL, are commonly driven by host address bus bits or a function of these bits. Using the HCNTL0/1 inputs, the host can specify an access to the HPI control (HPIC) register, the HPI address (HPIA) register (which serves as the pointer into HPI memory), or HPI data (HPID) register. The HPID register can also be accessed with an optional automatic address increment.

The autoincrement feature provides a convenient way of reading or writing to subsequent word locations. In autoincrement mode, a data read causes a postincrement of the HPIA, and a data write causes a preincrement of the HPIA. By writing to the HPIC, the host can interrupt the '54x CPU, and the $\overline{\text{HINT}}$ output can be used by the '54x to interrupt the host. The host can also acknowledge and clear $\overline{\text{HINT}}$ by writing to the HPIC.

Table 8–14 summarizes the three registers that the HPI utilizes for communication between the host device and the ’54x CPU and their functions.

Table 8–14. HPI Registers Description

Name	Address	Description
HPIA	–	HPI address register. Directly accessible only by the host. Contains the address in the HPI memory at which the current access occurs.
HPIC	002Ch	HPI control register. Directly accessible by either the host or by the ’54x. Contains control and status bits for HPI operations.
HPID	–	HPI data register. Directly accessible only by the host. Contains the data that was read from the HPI memory if the current access is a read, or the data that will be written to HPI memory if the current access is a write.

The two data strobes ($\overline{\text{HDS1}}$ and $\overline{\text{HDS2}}$), the read/write strobe (HR/$\overline{\text{W}}$), and the address strobe ($\overline{\text{HAS}}$) enable the HPI to interface to a variety of industry-standard host devices with little or no additional logic required. The HPI is easily interfaced to hosts with multiplexed address/data bus, separate address and data buses, one data strobe and a read/write strobe, or two separate strobes for read and write. This is described in detail later in this section.

The HPI ready pin (HRDY) allows insertion of wait states for hosts that support a ready input to allow deferred completion of access cycles and have faster cycle times than the HPI can accept due to ’54x operating clock rates. If HRDY, when used directly from the ’54x, does not meet host timing requirements, the signal can be resynchronized using external logic if necessary. HRDY is useful when the ’54x operating frequency is variable, or when the host is capable of accessing at a faster rate than the maximum shared-access mode access rate (up to the host-only mode maximum access rate). In both cases, the HRDY pin provides a convenient way to automatically (no software handshake needed) adjust the host access rate to a faster ’54x clock rate or switch the HPI mode.

All of these features combined allow the HPI to provide a flexible and efficient interface to a wide variety of industry-standard host devices. Also, the simplicity of the HPI interface greatly simplifies data transfers both from the host and the ‘54x sides of the interface. Once the interface is configured, data transfers are made with a minimum of overhead at a maximum speed.

8.5.2 Details of Host Port Interface Operation

This subsection includes a detailed description of each HPI external interface pin function, as well as descriptions of the register and control bit functions. Logical interface timings and initialization and read/write sequences are discussed in subsection 8.5.3, *Host Read/Write Access to HPI*, on page 8-37.

The external HPI interface signals implement a flexible interface to a variety of types of host devices. Devices with single or multiple data strobes and with or without address latch enable (ALE) signals can easily be connected to the HPI.

Table 8–15 gives a detailed description of the function of each of the HPI external interface pins.

Table 8–15. HPI Signal Names and Functions

HPI Pin	Host Pin	State†	Signal Function
$\overline{\text{HAS}}$	Address latch enable (ALE) or Address strobe or unused (tied high)	I	Address strobe input. Hosts with a multiplexed address and data bus connect $\overline{\text{HAS}}$ to their ALE pin or equivalent. HBIL, HCNTL0/1, and HR/$\overline{\text{W}}$ are then latched on $\overline{\text{HAS}}$ falling edge. When used, $\overline{\text{HAS}}$ must precede the later of $\overline{\text{HCS}}$, $\overline{\text{HDS1}}$, or $\overline{\text{HDS2}}$ (see '54x data sheet for detailed HPI timing specifications). Hosts with separate address and data bus can connect $\overline{\text{HAS}}$ to a logic-1 level. In this case, HBIL, HCNTL0/1, and HR/$\overline{\text{W}}$ are latched by the later of $\overline{\text{HDS1}}$, $\overline{\text{HDS2}}$, or $\overline{\text{HCS}}$ falling edge while $\overline{\text{HAS}}$ stays inactive (high).
HBIL	Address or control lines	I	Byte identification input. Identifies first or second byte of transfer (but not most significant or least significant — this is specified by the BOB bit in the HPIC register, described later in this section). HBIL is low for the first byte and high for the second byte.
HCNTL0, HCNTL1	Address or control lines	I	Host control inputs. Selects a host access to the HPIA register, the HPI data latches (with optional address increment), or the HPIC register.
$\overline{\text{HCS}}$	Address or control lines	I	Chip select. Serves as the enable input for the HPI and must be low during an access but may stay low between accesses. $\overline{\text{HCS}}$ normally precedes $\overline{\text{HDS1}}$ and $\overline{\text{HDS2}}$, but this signal also samples HCNTL0/1, HR/$\overline{\text{W}}$, and HBIL if $\overline{\text{HAS}}$ is not used and $\overline{\text{HDS1}}$ or $\overline{\text{HDS2}}$ are already low (this is explained in further detail later in this subsection). Figure 8–10 on page 8-33 shows the equivalent circuit of the $\overline{\text{HCS}}$, $\overline{\text{HDS1}}$ and $\overline{\text{HDS2}}$ inputs.

† I: Input
O: Output
Z: High impedance

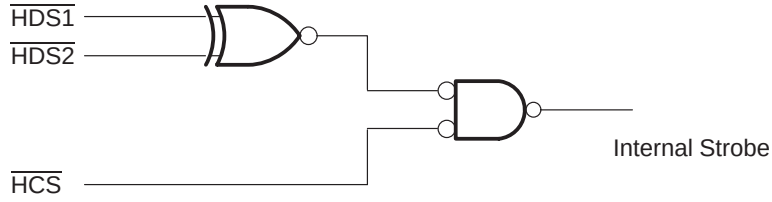
Table 8–15. HPI Signal Names and Functions (Continued)

HPI Pin	Host Pin	State†	Signal Function
HD0–HD7	Data bus	I/O/Z	Parallel bidirectional 3-state data bus. HD7 (MSB) through HD0 (LSB) are placed in the high-impedance state when not outputting (HDSx | HCS = 1) or when EMU1/OFF is active (low).
HDS1, HDS2	Read strobe and write strobe or data strobe	I	Data strobe inputs. Control transfer of data during host access cycles. Also, when HAS is not used, used to sample HBIL, HCNTL0/1, and HR/W when HCS is already low (which is the case in normal operation). Hosts with separate read and write strobes connect those strobes to either HDS1 or HDS2. Hosts with a single data strobe connect it to either HDS1 or HDS2, connecting the unused pin high. Regardless of HDS connections, HR/W is still required to determine direction of transfer. Because HDS1 and HDS2 are internally exclusive-NORed, hosts with a high true data strobe can connect this to one of the HDS inputs with the other HDS input connected low. Figure 8–10 on page 8-33 shows the equivalent circuit of the HDS1, HDS2, and HCS inputs.
HINT	Host interrupt input	O/Z	Host interrupt output. Controlled by the HINT bit in the HPIC. Driven high when the ’54x is being reset. Placed in high impedance when EMU1/OFF is active (low).
HRDY	Asynchronous ready	O/Z	HPI ready output. When high, indicates that the HPI is ready for a transfer to be performed. When low, indicates that the HPI is busy completing the internal portion of the previous transaction. Placed in high impedance when EMU1/OFF is active (low). HCS enables HRDY; that is, HRDY is always high when HCS is high.
HR/W	Read/Write strobe, address line, or multiplexed address/data	I	Read/write input. Hosts must drive HR/W high to read HPI and low to write HPI. Hosts without a read/write strobe can use an address line for this function.

† I: Input
O: Output
Z: High impedance

The $\overline{HCS}$ input serves primarily as the enable input for the HPI, and the $\overline{HDS1}$ and $\overline{HDS2}$ signals control the HPI data transfer; however, the logic with which these inputs are implemented allows their functions to be interchanged if desired. If $\overline{HCS}$ is used in place of $\overline{HDS1}$ and $\overline{HDS2}$ to control HPI access cycles, HRDY operation is affected (since $\overline{HCS}$ enables HRDY and HRDY is always high when $\overline{HCS}$ is high). The equivalent circuit for these inputs is shown in Figure 8–10. The figure shows that the internal strobe signal that samples the HCNTL0/1, HBIL, and HR/$\overline{W}$ inputs (when $\overline{HAS}$ is not used) is derived from all three of the input signals, as the logic illustrates. Therefore, the latest of $\overline{HDS1}$, $\overline{HDS2}$, or $\overline{HCS}$ is the one which actually controls sampling of the HCNTL0/1, HBIL, and HR/$\overline{W}$ inputs. Because $\overline{HDS1}$ and $\overline{HDS2}$ are exclusive-NORed, both these inputs being low does not constitute an enabled condition.

Figure 8–10. Select Input Logic



When using the $\overline{HAS}$ input to sample HCNTL0/1, HBIL, and HR/$\overline{W}$, this allows these signals to be removed earlier in an access cycle, therefore allowing more time to switch bus states from address to data information, facilitating interface to multiplexed address and data type buses. In this type of system, an ALE signal is often provided and would normally be the signal connected to $\overline{HAS}$.

The two control pins (HCNTL0 and HCNTL1) indicate which internal HPI register is being accessed and the type of access to the register. The states of these two pins select access to the HPI address (HPIA), HPI data (HPID), or HPI control (HPIC) registers. The HPIA register serves as the pointer into HPI memory, the HPIC contains control and status bits for the transfers, and the HPID contains the actual data transferred. Additionally, the HPID register can be accessed with an optional automatic address increment. Table 8–16 describes the HCNTL0/1 bit functions.

Table 8–16. HPI Input Control Signals Function Selection Descriptions

HCNTL1	HCNTL0	Description
0	0	Host can read or write the HPI control register, HPIC.
0	1	Host can read or write the HPI data latches. HPIA is automatically postincremented each time a read is performed and preincremented each time a write is performed.
1	0	Host can read or write the address register, HPIA. This register points to the HPI memory.
1	1	Host can read or write the HPI data latches. HPIA is not affected.

On the ’54x, HPI memory is a 2K × 16-bit word block of dual-access RAM that resides at 1000h to 17FFh in data memory space and optionally, depending on the state of the OVLY bit, in program memory space.

From the host interface, the 2K-word block of HPI memory can conveniently be accessed at addresses 0 through 7FFh; however, the memory can also be accessed by the host starting with any HPIA values with the 11 LSBs equal to 0. For example, the first word of the HPI memory block, addressed at 1000h by the ’54x in data memory space, can be accessed by the host with any of the following HPIA values: 0000h, 0800h,1000h,1800h, ··· F800h.

The HPI autoincrement feature provides a convenient way of accessing consecutive word locations in HPI memory. In the autoincrement mode, a data read causes a postincrement of the HPIA, and a data write causes a preincrement of the HPIA. Therefore, if a write is to be made to the first word of HPI memory with the increment option, due to the preincrement nature of the write operation, the HPIA should first be loaded with any of the following values: 07FFh, 0FFFh, 17FFh, ··· FFFFh. The HPIA is a 16-bit register and all 16 bits can be written to or read from, although with a 2K-word HPI memory implementation, only the 11 LSBs of the HPIA are required to address the HPI memory. The HPIA increment and decrement affect all 16 bits of this register.

HPI Control Register Bits and Function

Four bits control HPI operation. These bits are BOB (which selects first or second byte as most significant), SMOD (which selects host or shared-access mode), and DSPINT and HINT (which can be used to generate ’54x and host interrupts, respectively) and are located in the HPI control register (HPIC). A detailed description of the HPIC bit functions is presented in Table 8–17.

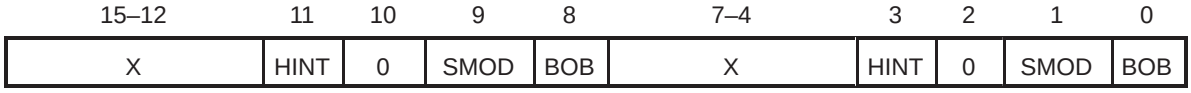
Table 8–17. HPI Control Register (HPIC) Bit Descriptions

Bit	Host Access	'54x Access	Description
BOB	Red/Write	–	If BOB = 1, first byte is least significant. If BOB = 0, first byte is most significant. BOB affects both data and address transfers. Only the host can modify this bit and it is not visible to the '54x. BOB must be initialized before the first data or address register access.
SMOD	Read	Read/Write	If SMOD = 1, shared-access mode (SAM) is enabled: the HPI memory can be accessed by the '54x. If SMOD = 0, host-only mode (HOM) is enabled: the '54x is denied access to the entire HPI RAM block. SMOD = 0 during reset; SMOD = 1 after reset. SMOD can be modified only by the '54x but can be read by both the '54x and the host.
DSPINT	Write	–	The host processor-to-'54x interrupt. This bit can be written only by the host and is not readable by the host or the '54x. When the host writes a 1 to this bit, an interrupt is generated to the '54x. Writing a 0 to this bit has no effect. Always read as 0. When the host writes to HPIC, both bytes must write the same value. See this subsection for a detailed description of DSPINT function.
HINT	Read/Write	Read/Write	This bit determines the state of the '54x $\overline{\text{HINT}}$ output, which can be used to generate an interrupt to the host. HINT = 0 upon reset, which causes the external $\overline{\text{HINT}}$ output to be inactive (high). The HINT bit can be set only by the '54x and can be cleared only by the host. The '54x writes a 1 to HINT, causing the $\overline{\text{HINT}}$ pin to go low. The HINT bit is read by the host or the '54x as a 0 when the external $\overline{\text{HINT}}$ pin is inactive (high) and as a 1 when the $\overline{\text{HINT}}$ pin is active (low). For the host to clear the interrupt, however, it must write a 1 to HINT. Writing a 0 to the HINT bit by either the host or the '54x has no effect. See this subsection for a detailed description of HINT function.

Because the host interface always performs transfers with 8-bit bytes and the control register is normally the first register accessed to set configuration bits and initialize the interface, the HPIC is organized on the host side as a 16-bit register with the same high and low byte contents (although access to certain bits is limited, as described previously) and with the upper bits unused on the '54x side. The control/status bits are located in the least significant four bits. The host accesses the HPIC register with the appropriate selection of HCNTL0/1, as described previously, and two consecutive byte accesses to the 8-bit HPI data bus. When the host writes to HPIC, both the first and second byte written must be the same value. The '54x accesses the HPIC at 002Ch in data memory space.

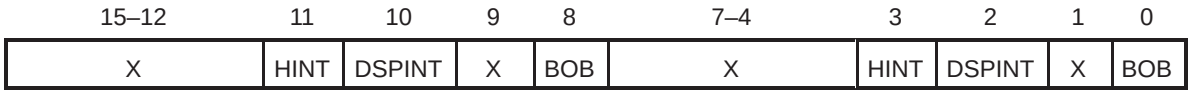
The layout of the HPIC bits is shown in Figure 8–11 through Figure 8–14. In the figures for read operations, if 0 is specified, this value is always read; if X is specified, an unknown value is read. For write operations, if X is specified, any value can be written. On a host write, both bytes must be identical. Note that bits 4–7 and 12–15 on the host side and bits 4–15 on the ’54x side are reserved for future expansion.

Figure 8–11. HPIC Diagram — Host Reads from HPIC



Note: X = Unknown value is read.

Figure 8–12. HPIC Diagram — Host Writes to HPIC



Note: X = Any value can be written.

Figure 8–13. HPIC Diagram — TMS320C54x Reads From HPIC

Note: X = Unknown value is read.

Figure 8–14. HPIC Diagram — TMS320C54x Writes to HPIC

Note: X = Any value can be written.

Because the ’54x can write to the SMOD and HINT bits, and these bits are read twice on the host interface side, the first and second byte reads by the host may yield different data if the ’54x changes the state of one or both of these bits in between the two read operations. The characteristics of host and ’54x HPIC read/write cycles are summarized in Table 8–18.

Table 8–18. HPIC Host/TMS320C54x Read/Write Characteristics

Device	Read	Write
Host	2 bytes	2 bytes (Both bytes must be equal)
’54x	16 bits	16 bits

8.5.3 Host Read/Write Access to HPI

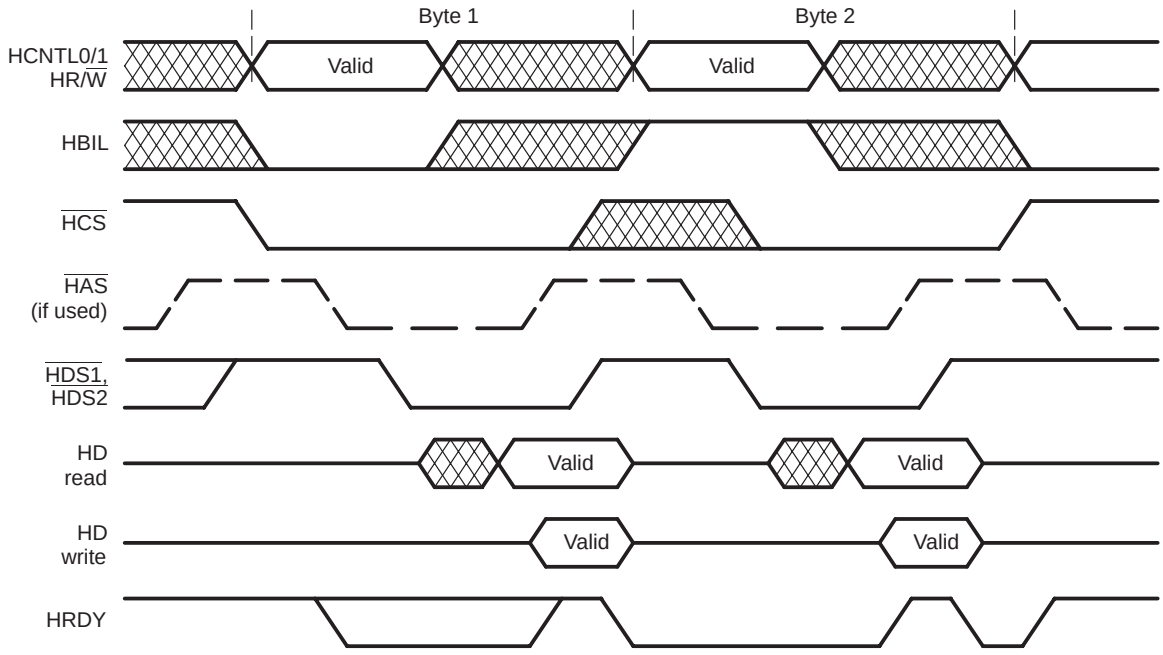
The host begins HPI accesses by performing the external interface portion of the cycle; that is, initializing first the HPIC register, then the HPIA register, and then writing data to or reading data from the HPID register. Writing to HPIA or HPID initiates an internal cycle that transfers the desired data between the HPID and the dedicated internal HPI memory. Because this process requires several ’54x cycles, each time an HPI access is made, data written to the HPID is not written to the HPI memory until after the host access cycle, and the data read from the HPID is the data from the previous cycle. Therefore, when reading, the data obtained is the data from the location specified in the previous access, and the current access serves as the initiation of the next cycle. A similar sequence occurs for a write operation: the data written to HPID is not written to HPI memory until after the external cycle is completed. If an HPID read operation immediately follows an HPID write operation, the same data (the data written) is read.

The autoincrement feature available for HPIA results in sequential accesses to HPI memory by the host being extremely efficient. During random (nonsequential) transfers or sequential accesses with a significant amount of time between them, it is possible that the ’54x may have changed the contents of the location being accessed between a host read and the previous host data read/write or HPIA write access, because of the prefetch nature of internal HPI operation. If this occurs, data different from the current memory contents may be read. Therefore, in cases where this is of concern in a system, two reads from the same address or an address write prior to the read access can be made to ensure that the most recent data is read.

When the host performs an external access to the HPI, there are two distinctly different types of cycles that can occur: those for which wait states are generated (the HRDY signal is active) and those without wait states. In general, when in shared-access mode (SAM), the HRDY signal is used; when in host-only mode (HOM), HRDY is not active and remains high; however, there are exceptions to this, which will be discussed.

For accesses utilizing the HRDY signal, during the time when the internal portion of the transfer is being performed (either for a read or a write), HRDY is low, indicating that another transfer cannot yet be initiated. Once the internal cycle is completed and another external cycle can begin, HRDY is driven high by the HPI. This occurs after a fixed delay following a cycle initiation (refer to the ’54x data sheet for detailed timing information for HPI external interface timings). Therefore, unless back-to-back cycles are being performed, HRDY is normally high when the first byte of a cycle is transferred. The external HPI cycle using HRDY is shown in the timing diagram in Figure 8–15.

Figure 8–15. HPI Timing Diagram



In a typical external access, as shown in Figure 8–15, the cycle begins with the host driving HCNTL0/1, HR/$\overline{W}$, HBIL, and $\overline{HCS}$, indicating specifically what type of transfer is to occur and whether the cycle is to be read or a write. Then the host asserts the $\overline{HAS}$ signal (if used) followed by one of the data strobe signals. If HRDY is not already high, it goes high when the previous internal cycle is complete, allowing data to be transferred, and the control signals are deasserted. Following the external HPI cycle, HRDY goes low and stays low for a period of approximately five CLKOUT cycles (refer to the ’54x data sheet for HPI timing information) while the ’54x completes the internal HPI memory access, and then HRDY is driven high again. Note, however, HRDY is always high when $\overline{HCS}$ is high.

As mentioned previously, SAM accesses generally utilize the HRDY signal. The exception to the HRDY-based interface timings when in SAM occurs when reading HPIC or HPIA or writing to HPIC (except when writing 1 to either DSPINT or HINT). In these cases, HRDY stays high; for all other SAM accesses, HRDY is active.

Host access cycles when in HOM have timings different from the SAM timings described previously. In HOM, the CPU is not involved (with one exception), and the access can be completed after a short, fixed delay time. The exception to this occurs when writing 1s to the DSPINT or HINT bits in HPIC. In this case, the host access takes several CPU clock cycles, and SAM timings apply. Besides the HRDY timings and a faster cycle time, HOM access cycles are logically the same as SAM access cycles. A summary of the conditions under which the HRDY signal is active (where SAM timings apply) for host accesses is shown in Table 8–19. When HRDY is not active (HRDY stays high), HOM timings apply. Refer to the ’54x data sheet for detailed HPI timing specifications.

Table 8–19. Wait-State Generation Conditions

	Wait State Generated	
Register	**Reads**	**Writes**
HPIC	No	1 to DSPINT/HINT – Yes All other cycles – No
HPIA	No	HOM – No SAM – Yes
HPID	HOM – No SAM – Yes	HOM – No SAM – Yes

Example Access Sequences

A complete host access cycle always involves two bytes, the first with HBIL low, and the second with HBIL high. This 2-byte sequence must be followed regardless of the type of host access (HPIA, HPIC, or data access) and the host must not break the first byte/second byte (HBIL low/high) sequence of an ongoing HPI access. If this sequence is broken, data may be lost, and unpredictable operation may result.

Before accessing data, the host must first initialize HPIC, in particular the BOB bit, and then HPIA (in this order, because BOB affects the HPIA access). After initializing BOB, the host can then write to HPIA with the correct byte alignment. On an HPI memory read operation, after completion of the HPIA write, the HPI memory is read and the contents at the given address are transferred to the two 8-bit data latches, the first byte data latch and the second byte data latch. Table 8–20 illustrates the sequence involved in initializing BOB and HPIA for an HPI memory read. In this example, BOB is set to 0 and a read is requested of the first HPI memory location (in this case 1000h), which contains FFFEh.

Table 8–20. Initialization of BOB and HPIA

Event	HD	HR/$\overline{W}$	HCNTL1/0	HBIL	HPIC	HPIA	latch1	latch2
Host writes HPIC, 1st byte	00	0	00	0	00xx	xxxx	xxxx	xxxx
Host writes HPIC, 2nd byte	00	0	00	1	0000	xxxx	xxxx	xxxx
Host writes HPIA, 1st byte	10	0	10	0	0000	10xx	xxxx	xxxx
Host writes HPIA, 2nd byte	00	0	10	1		1000	xxxx	xxxx
Internal HPI RAM read complete						1000	FF	FE

In the cycle shown in Table 8–20, BOB and HPIA are initialized, and by loading HPIA, an internal HPI memory access is initiated. The last line of Table 8–20 shows the condition of the HPI after the internal RAM read is complete; that is, after some delay following the end of the host write of the second byte to HPIA, the read is completed and the data has been placed in the upper and lower byte data latches. For the host to actually retrieve this data, it must perform an additional read of HPID. During this HPID read access, the contents of the first byte data latch appears on the HD pins when HBIL is low and the content of the second byte data latch appears on the HD pins when HBIL is high. Then the address is incremented if autoincrement is selected and the memory is read again into the data latches. The sequence involved in this access is shown in Table 8–21.

Table 8–21. Read Access to HPI With Autoincrement

Event	HD	HR/$\overline{W}$	HCNTL1/0	HBIL	HPIC	HPIA	latch1	latch2
Host reads data, 1st byte	FF	1	01	0	0000	1000	FF	FE
Host reads data, 2nd byte	FE	1	01	1	0000	1000	FF	FE
Internal HPI RAM read complete						1001	6A	BC

In the access shown in Table 8–21, the data obtained from reading HPID is the data from the read initiated in the previous cycle (the one shown in Table 8–20) and the access performed as shown in Table 8–21 also initiates a further read, this time at location 1001h (because autoincrement was specified in this access by setting HCNTL1/0 to 01). Also, when autoincrement is selected, the increment occurs with each 16-bit word transferred (not with each byte); therefore, as shown in Table 8–21, the HPIA is incremented by only 1. The last line of Table 8–21 indicates that after the second internal RAM read is complete, the contents of location 1001h (6ABCh) has been read and placed into the upper and lower byte data latches.

During a write access to the HPI, the first byte data latch is overwritten by the data coming from the host while the HBIL pin is low, and the second byte data latch is overwritten by the data coming from the host while the HBIL pin is high. At the end of this write access, the data in both data latches is transferred as a 16-bit word to the HPI memory at the address specified by the HPIA register. The address is incremented prior to the memory write because autoincrement is selected.

An HPI write access is illustrated in Table 8–22. In this example, after the internal portion of the write is completed, location 1002h of HPI RAM contains 1234h. If a read of the same address follows this write, the same data just written in the data latches (1234h) is read back.

Table 8–22. Write Access to HPI With Autoincrement

Event	HD	HR/$\overline{W}$	HCNTL1/0	HBIL	HPIC	HPIA	latch1	latch2
Host writes data, 1st byte	12	0	01	0	0000	1002	12	FE
Host writes data, 2nd byte	34	0	01	1	0000	1002	12	34
Internal HPI RAM write complete						1002	12	34

8.5.4 DSPINT and HINT Function Operation

The host and the ’54x can interrupt each other using bits in the HPIC register. This subsection presents more information about this process.

Host Device Using DSPINT to Interrupt the ’54x

A ’54x interrupt is generated when the host writes a 1 to the DSPINT bit in HPIC. This interrupt can be used to wake up the ’54x from IDLE. The host and the ’54x always read this bit as 0. A ’54x write has no effect. Once a 1 is written to DSPINT by the host, a 0 need not be written before another interrupt can be generated, and writing a 0 to this bit has no effect. The host should not write a 1 to the DSPINT bit while writing to BOB or HINT, or an unwanted ’54x interrupt is generated.

On the ’54x, the host-to-’54x interrupt vector address is xx64h. This interrupt is located in bit 9 of the IMR/IFR. Since the ’54x interrupt vectors can be remapped into the HPI memory, the host can instruct the ’54x to execute preprogrammed functions by simply writing the start address of a function to address xx65h in the HPI memory prior to interrupting the ’54x with a branch instruction located at address xx64h. If the interrupts are remapped to the host port accessible on-chip RAM, you must use SAM and the host must not write to location xx00h to xx7Fh, except for xx65h.

Host Port Interface ('54x) Using HINT to Interrupt the Host Device

When the '54x writes a 1 to the HINT bit in HPIC, the $\overline{\text{HINT}}$ output is driven low; the HINT bit is read as a 1 by the '54x or the host. The $\overline{\text{HINT}}$ signal can be used to interrupt the host device. The host device, after detecting the $\overline{\text{HINT}}$ interrupt line, can acknowledge and clear the '54x interrupt and the HINT bit by writing a 1 to the HINT bit. The HINT bit is cleared and then read as a 0 by the '54x or the host, and the $\overline{\text{HINT}}$ pin is driven high. If the '54x or the host writes a 0, the HINT bit remains unchanged. While accessing the SMOD bit, the '54x should not write a 1 to the HINT bit unless it also wants to interrupt the host.

8.5.5 Considerations in Changing HPI Memory Access Mode (SAM/HOM) and IDLE2/3 Use

The HPI host-only mode (HOM) allows the host to access HPI RAM while the '54x is in IDLE2/3 (that is, completely halted). Additionally, the external clock input to the '54x can be stopped for the lowest power consumption configuration. Under these conditions, random accesses can still be made without having to restart the external clock for each access and wait for its lockup time if the '54x on-chip PLL is used. The external clock need only be restarted before taking the '54x out of IDLE2/3.

The host cannot access HPI RAM in SAM when the '54x is in IDLE2/3, because CPU clocks are required for access in this mode of operation. Therefore, if the host requires access to the HPI RAM while the '54x is in IDLE2/3, the '54x must change HPI mode to HOM before entering IDLE2/3. When the HPI is in HOM, the '54x can access HPIC to toggle the SMOD bit or send an interrupt to the host, but cannot access the HPI RAM block; a '54x access to the HPI RAM is disregarded in HOM. In order for the '54x to again access the HPI RAM block, HPI mode must be changed to SAM after exiting IDLE2/3.

To select HOM, a 0 must be written to the SMOD bit in HPIC. To select SAM, a 1 must be written to SMOD. When changing between HOM and SAM, two considerations must be met for proper operation. First, the instruction immediately following the one that changes from SAM to HOM must not be an IDLE 2 or IDLE 3. This is because in this case, due to the '54x pipeline and delays in the SAM to HOM mode switch, the IDLE2/3 takes effect before the mode switch occurs, causing the HPI to remain in SAM; therefore, no host accesses can occur.

The second consideration is that when changing from HOM to SAM, the instruction immediately following the one that changes from HOM to SAM cannot read the HPI RAM block. This requirement is due to the fact that the mode has not yet changed when the HPI RAM read occurs and the RAM read is ignored because the mode switch has not yet occurred. HPI RAM writes are not included in this restriction because these operations occur much later in the pipeline, so it is possible to write to HPI RAM in the instruction following the one which changes from HOM to SAM.

On the host side, there are no specific considerations associated with the mode changes. For example, it is possible to have a third device wake up the ’54x from IDLE2/3 and the ’54x changing to SAM upon wake-up without a software handshake with the host. The host can continue accessing while the HPI mode changes. However, if the host accesses the HPI RAM while the mode is being changed, the actual mode change will be delayed until the host access is completed. In this case, a ’54x access to the HPI memory is also delayed.

Table 8–23 illustrates the sequence of events involved in entering and exiting an IDLE2/3 state on the ’54x when using the HPI. Throughout the process, the HPI is accessible to the host.

Table 8–23. Sequence for Entering and Exiting IDLE2 and IDLE3

Host or Other Device	’54x	Mode	’54x clock
	Switches mode to HOM	HOM	Running
	Executes a NOP	HOM	Running
	Executes IDLE 2 or IDLE 3 instruction	HOM	Running
May stop DSP clock	In IDLE2/3	HOM	Stopped or running
Turns on DSP clock if it was stopped†	In IDLE2/3	HOM	Running
Sends an interrupt to DSP	In IDLE2/3	HOM	Running
	’54x wakes up from IDLE2/3	HOM	Running
	’54x switches mode to SAM	SAM	Running

† Sufficient wake-up time must be ensured when the ’54x on-chip PLL is used.

8.5.6 Access of HPI Memory During Reset

The ’54x is not operational during reset, but the host can access the HPI, allowing program or data downloads to the HPI memory. When this capability is used, it is often convenient for the host to control the ’54x reset input. The sequence of events for resetting the ’54x and downloading a program to HPI memory while the ’54x is in reset is summarized in Table 8–24 and corresponds to the reset of the ’54x.

Initially, the host stops accessing the HPI at least six ’54x periods before driving the ’54x reset line low. The host then drives the ’54x reset line low and can start accessing the HPI after a minimum of four ’54x periods. The HPI mode is automatically set to HOM during reset, allowing high-speed program download. The ’54x clock can even be stopped at this time; however, the clock must be running when the reset line falls and rises for proper reset operation of the ’54x.

Once the host has finished downloading into HPI memory, the host stops accessing the HPI and drives the ’54x reset line high. At least 20 ’54x periods after the reset line rising edge, the host can again begin accessing the HPI. This number of periods corresponds to the internal reset delay of the ’54x. The HPI mode is automatically set to SAM upon exiting reset.

If the host writes a 1 to DSPINT while the ’54x is in reset, the interrupt is lost when the ’54x comes out of reset. The ’54x warm boot can use the HPI memory and start execution from the lowest HPI address.

Table 8–24. HPI Operation During RESET

Host	’54x	Mode	’54x CLK
Waits 6 ’54x clock periods	Running	X	Running
Brings RESET low and waits 4 clocks	Goes into reset	HOM	Running
Can stop ’54x clock	In reset	HOM	Stopped or running
Writes program and/or data in HPI memory	In reset	HOM	Stopped or running
Turns on DSP clock if it was stopped†	In reset	HOM	Running
Brings RESET high	In reset	HOM	Running
Waits 20 ’54x clock periods	Comes out of reset	SAM	Running
Can access HPI	Running	SAM	Running

† Sufficient wake-up time must be ensured when the ’54x on-chip PLL is used.

Chapter 9

Serial Ports

This chapter discusses the three serial port interfaces connected to the ’54x core CPU:

- Standard synchronous serial port interface
- Buffered serial port interface
- Time-division multiplexed serial port interface

These peripherals are controlled through registers that reside in the memory map. The serial ports are synchronized to the core CPU by way of interrupts.

Topic **Page**

9.1 Introduction to the Serial Ports

Three different types of synchronous serial port interfaces are available on ’54x devices:

- ❑ Standard synchronous serial port interface.
- ❑ Buffered serial port (BSP) interface.
- ❑ Time-division multiplexed (TDM) serial port interface.

Table 9–1 lists the serial ports available on the various ’54x devices.

Table 9–1. Serial Ports on the TMS320C54x Devices

Device	Standard Synchronous Serial Ports	Buffered Serial Ports	Time-Division Multiplexed Serial Ports
’541	2	0	0
’542	0	1	1
’543	0	1	1
’545	1	1	0
’546	1	1	0
’548	0	2	1
’549	0	2	1

Table 9–2 lists the sections that should be consulted for the various serial ports and their modes.

Table 9–2. Sections that Cover the Serial Ports

Serial Port	Mode	See . . .
Standard	–	Section 9.2, *Standard Serial Port Interface*, on page 9-3.
Buffered	Autobuffering	Section 9.3, *Buffered Serial Port (BSP) Interface*, on page 9-32.
	Nonbuffered (standard)	Section 9.2, *Standard Serial Port Interface*, on page 9-3.
TDM	TDM	Section 9.4, *Time-Division Multiplexed (TDM) Serial Port Interface*, on page 9-55.
	Non-TDM (standard)	Section 9.2, *Standard Serial Port Interface*, on page 9-3.

9.2 Serial Port Interface

Several ’54x devices implement a variety of types of flexible serial port interfaces. These serial port interfaces provide full duplex, bidirectional, communication with serial devices such as codecs, serial analog to digital (A/D) converters, and other serial systems. The serial port interface signals are directly compatible with many industry-standard codecs and other serial devices. The serial port may also be used for interprocessor communication in multiprocessing applications (the time-division multiplexed (TDM) serial port is especially optimized for multiprocessing).

Three different types of serial port interfaces are available on ’54x devices. The basic standard serial port interface is implemented on ’541, ’545, and ’546 devices. The TDM serial port interface is implemented on the ’542, ’543, ’548, and ’549 devices. The ’542, ’543, ’545, ’546, ’548, and ’549 devices include a buffered serial port (BSP) that implements an automatic buffering feature, which greatly reduces CPU overhead required in handling serial data transfers. See Table 9–1 for information about the features included in various ’54x devices.

The BSP operates in either autobuffering or nonbuffered mode. When operated in nonbuffered (or standard) mode, the BSP functions the same as the basic standard serial port (except where specifically indicated) and is described in this section. The TDM serial port operates in either TDM or non-TDM mode. When operated in non-TDM (or standard) mode, the TDM serial port also functions the same as the basic standard serial port and is described in this section.

The BSP also implements several enhanced features in standard mode. These features, together with operation of the BSP in autobuffering mode, are described in section 9.3, *Buffered Serial Port (BSP) Interface*, on page 9-32. Therefore, when using the ’542, ’543, ’545, ’546, ’548, and ’549 devices, you should consult section 9.3.

Operation of the TDM serial port in TDM mode is described in section 9.4, *Time-Division Multiplexed (TDM) Serial Port Interface*, on page 9-55. Note that the BSP and TDM serial ports initialize to a standard serial port compatible mode upon reset.

In all ’54x serial ports, both receive and transmit operations are double-buffered, thus allowing a continuous communications stream with either 8- or 16-bit data packets. The continuous mode provides operation that, once initiated, requires no further frame synchronization pulses (FSR and FSX) when transmitting at maximum packet frequency. The serial ports are fully static and thus will function at arbitrarily low clocking frequencies. The maximum operating frequency for the standard serial port of one-fourth of CLKOUT (10 Mbit/s at 25 ns, 12.5 Mbit/s at 20 ns) is achieved when using internal serial port clocks. The maximum operating frequency for the BSP is CLKOUT. When the serial ports are in reset, the device may be configured to turn off the internal serial port clocks, allowing the device to run in a lower power mode of operation.

9.2.1 Serial Port Interface Registers

The serial port operates through the three memory-mapped registers (SPC, DXR, and DRR) and two other registers (RSR and XSR) that are not directly accessible to the program, but are used in the implementation of the double-buffering capability. These five registers are listed in Table 9–3.

Table 9–3. Serial Port Registers

Address	Register	Description
†	DRR	Data receive register
†	DXR	Data transmit register
†	SPC	Serial port control register
—	RSR	Receive shift register
—	XSR	Data transmit shift register

† See Section 8.1, *Peripheral Memory-Mapped Registers*.

- Data receive register (DRR). The 16-bit memory-mapped data receive register (DRR) holds the incoming serial data from the RSR to be written to the data bus. At reset, the DRR is cleared.
- Data transmit register (DXR). The 16-bit memory-mapped data transmit register (DXR) holds the outgoing serial data from the data bus to be loaded in the XSR. At reset, the DXR is cleared.
- Serial port control register (SPC). The 16-bit memory-mapped serial port control register (SPC) contains the mode control and status bits of the serial port.

- Data receive shift register (RSR). The 16-bit data receive shift register (RSR) holds the incoming serial data from the serial data receive (DR) pin and controls the transfer of the data to the DRR.

- Data transmit shift register (XSR). The 16-bit data transmit shift register (XSR) controls the transfer of the outgoing data from the DXR and holds the data to be transmitted on the serial data transmit (DX) pin.

During normal serial port operation, the DXR is typically loaded with data to be transmitted on the serial port by the executing program, and its contents read automatically by the serial port logic to be sent out when a transmission is initiated. The DRR is loaded automatically by the serial port logic with data received on the serial port and read by the executing program to retrieve the received data.

At times during normal serial port operation, however, it may be desirable for a program to perform other operations with the memory-mapped serial port registers besides simply writing to DXR and reading from DRR.

On the SP, the DXR and DRR may be read or written at any time regardless of whether the serial port is in reset or not. On the BSP, access to these registers is restricted; the DRR can only be read, and the DXR can only be written when autobuffering is disabled (see subsection 9.3.2, *Autobuffering Unit (ABU) Operation*, on page 9-39). The DRR can only be written when the BSP is in reset. The DXR can be read at any time.

Note, however, that on both the SP and the BSP, care should be exercised when reading or writing to these registers during normal operation. With the DRR, since, as mentioned previously, this register is written automatically by the serial port logic when data is received, if a write to DRR is performed, subsequent reads may not yield the result written if a serial port receive occurs after the write but before the read is performed. With the DXR, care should be exercised when this register is written, since if previously written contents intended for transmission have not yet been sent, these contents will be overwritten and the original data lost. As mentioned previously, the DXR can be read at any time.

Alternatively, DXR and DRR may also serve as general purpose storage if they are not required for serial port use. If these registers are to be used for general purpose storage, the transmit and/or receive sections of the serial port should be disabled either by tying off (by pulling up or down, whichever is appropriate) external input pins which could spuriously cause serial port transfers, or by putting the port in reset.

9.2.2 Serial Port Interface Operation

This subsection describes operation of the basic standard serial port interface, which includes operation of the TDM and BSP serial ports when configured in standard mode. Table 9–4 lists the pins used in serial port operation. Figure 9–1 shows these pins for two ’54x serial ports connected for a one-way transfer from device 0 to device 1. Only three signals are required to connect from a serial port transmitter to a receiver for data transmission. The transmitted serial data signal (DX) sends the actual data. The transmit frame synchronization signal (FSX) initiates the transfer (at the beginning of the packet), and the transmit clock signal (CLKX) clocks the bit transfer. The corresponding pins on the receive device are DR, FSR and CLKR, respectively.

Table 9–4. Serial Port Pins

Pin	Description
CLKR	Receive clock signal
CLKX	Transmit clock signal
DR	Received serial data signal
DX	Transmitted serial data signal
FSR	Receive framing synchronization signal
FSX	Transmit frame synchronization signal

Figure 9–1. One-Way Serial Port Transfer

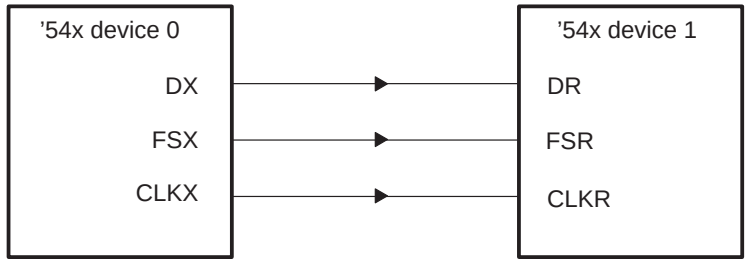


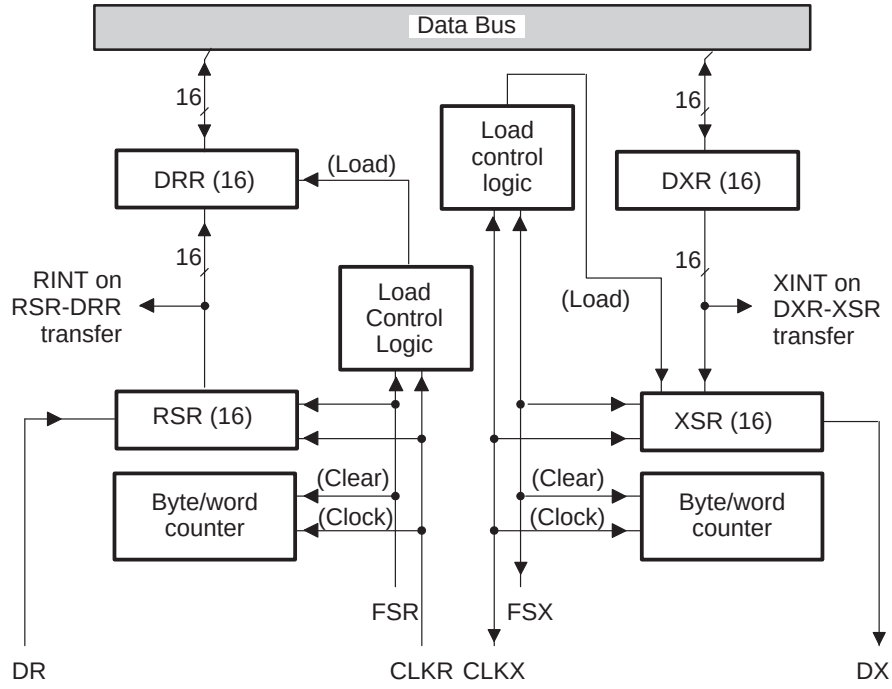
Figure 9–2 shows how the pins and registers are configured in the serial port logic and how the double-buffering is implemented.

Transmit data is written to the DXR, while received data is read from the DRR. A transmit is initiated by writing data to the DXR, which copies the data to the XSR when the XSR is empty (when the last word has been transmitted serially, that is, driven on the DX pin). The XSR manages shifting the data to the DX pin, thus allowing another write to DXR as soon as the DXR-to-XSR copy is completed.

During transmits, upon completion of the DXR-to-XSR copy, a 0-to-1 transition occurs on the transmit ready (XRDY) bit in the SPC. This 0-to-1 transition generates a serial port transmit interrupt (XINT) that signals that the DXR is ready to be reloaded. See Section 6.10, *Interrupts*, on page 6-26 for more information on ’54x interrupts.

The process is similar in the receiver. Data from the DR pin is shifted into the RSR, which is then copied into the DRR from which it may be read. Upon completion of the RSR-to-DRR copy, a 0-to-1 transition occurs on the receive ready (RRDY) bit in the SPC. This 0-to-1 transition generates a serial port receive interrupt (RINT). Thus, the serial port is double-buffered because data can be transferred to or from DXR or DRR while another transmit or receive is being performed. Note that transfer timing is synchronized by the frame sync pulse in burst mode (discussed in more detail in subsection 9.2.4, *Burst Mode Transmit and Receive Operations*, on page 9-17).

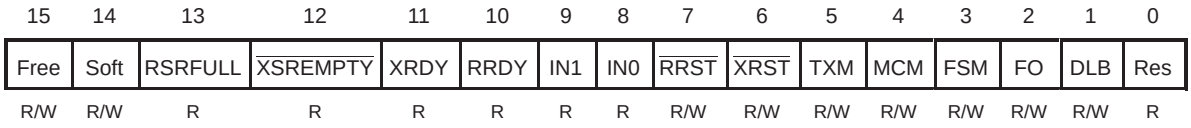
Figure 9–2. Serial Port Interface Block Diagram



9.2.3 Configuring the Serial Port Interface

The SPC contains control bits which configure the operation of the serial port. The SPC bit fields are shown in Figure 9–3 and described in Table 9–5. Note that seven bits in the SPC are read only and the remaining nine bits are read/write.

Figure 9–3. Serial Port Control Register (SPC) Diagram



Note: R = Read, W = Write

Table 9–5. Serial Port Control Register (SPC) Bit Summary

Bit	Name	Reset Value	Function
15	Free	0	This bit is used in conjunction with the Soft bit to determine the state of the serial port clock when a breakpoint is encountered in the HLL debugger. See Table 9–6 on page 9-17 for the serial port clock configurations. Free = 0 — The Soft bit selects the emulation mode. Free = 1 — The serial port clock runs free regardless of the Soft bit.
14	Soft	0	This bit is used in conjunction with the Free bit to determine the state of the serial port clock when a breakpoint is encountered in the HLL debugger. When the Free bit is cleared to 0, the Soft bit selects the emulation mode. See Table 9–6 on page 9-17 for the serial port clock configurations. Soft = 0 — The serial port clock stops immediately, thus aborting any transmission. Soft = 1 — The clock stops after completion of the current transmission.
13	RSRFULL	0	Receive Shift Register Full. This bit indicates whether the receiver has experienced overrun. Overrun occurs when RSR is full and DRR has not been read since the last RSR-to-DRR transfer. On the SP, when FSM = 1, the occurrence of a frame sync pulse on FSR qualifies the generation of RSRFULL = 1. When FSM = 0, and on the BSP, only the basic two conditions apply; that is, RSRFULL goes high without waiting for an FSR pulse. RSRFULL = 0 — Any one of the following three events clears the RSRFULL bit to 0: reading DRR, resetting the receiver ($\overline{\text{RRST}}$ bit to 0), or resetting the device. RSRFULL = 1 — The port has recognized an overrun. When RSRFULL = 1, the receiver halts and waits for DRR to be read, and any data sent on DR is lost. On the SP, the data in RSR is preserved; on the BSP, the contents of RSR are lost.

Table 9–5. Serial Port Control Register (SPC) Bit Summary (Continued)

Bit	Name	Reset Value	Function
12	$\overline{\text{XSREMPTY}}$	0	Transmit Shift Register Empty. This bit indicates whether the transmitter has experienced underflow. Underflow occurs when XSR is empty and DXR has not been loaded since the last DXR-to-XSR transfer. $\overline{\text{XSREMPTY}}$ = 0 — Any one of the following three events clears the $\overline{\text{XSREMPTY}}$ bit to 0: underflow has occurred, resetting the transmitter ($\overline{\text{XRST}}$ bit to 0), or resetting the device. $\overline{\text{XSREMPTY}}$ = 1 — On the SP, $\overline{\text{XSREMPTY}}$ is deactivated (set to 1) directly as a result of writing to DXR; on the BSP, $\overline{\text{XSREMPTY}}$ is only deactivated after DXR is loaded *followed* by the occurrence of an FSX pulse.
11	XRDY	1	Transmit Ready. A transition from 0 to 1 of the XRDY bit indicates that the DXR contents have been copied to XSR and that DXR is ready to be loaded with a new data word. A transmit interrupt (XINT) is generated upon the transition. This bit can be polled in software instead of using serial port interrupts. Note that on the SP, XRDY is generated directly as a result of writing to DXR; while on the BSP, XRDY is only generated after DXR is loaded *followed* by the occurrence of an FSX pulse. At reset or serial port transmitter reset ($\overline{\text{XRST}}$ = 0), the XRDY bit is set to 1.
10	RRDY	0	Receive Ready. A transition from 0 to 1 of the RRDY bit indicates that the RSR contents have been copied to the DRR and that the data can be read. A receive interrupt (RINT) is generated upon the transition. This bit can be polled in software instead of using serial port interrupts. At reset or serial port receiver reset ($\overline{\text{RRST}}$ = 0), the RRDY bit is cleared to 0.
9	IN1	x	Input 1. This bit allows the CLKX pin to be used as a bit input. IN1 reflects the current level of the CLKX pin of the device. When CLKX switches levels, there is a latency of between 0.5 and 1.5 CLKOUT cycles before the new CLKX value is represented in the SPC.
8	IN0	x	Input 0. This bit allows the CLKR pin to be used as a bit input. IN0 reflects the current level of the CLKR pin of the device. When CLKR switches levels, there is a latency of between 0.5 and 1.5 CLKOUT cycles before the new CLKR value is represented in the SPC.
7	$\overline{\text{RRST}}$	0	Receive Reset. This signal resets and enables the receiver. When a 0 is written to the $\overline{\text{RRST}}$ bit, activity in the receiver halts. $\overline{\text{RRST}}$ = 0 — The serial port receiver is reset. Writing a 0 to $\overline{\text{RRST}}$ clears the RSRFULL and RRDY bits to 0. $\overline{\text{RRST}}$ = 1 — The serial port receiver is enabled.

Table 9–5. Serial Port Control Register (SPC) Bit Summary (Continued)

Bit	Name	Reset Value	Function
6	$\overline{\text{XRST}}$	0	Transmitter Reset. This signal is used to reset and enable the transmitter. When a 0 is written to the $\overline{\text{XRST}}$ bit, activity in the transmitter halts. When the XRDY bit is 0, writing a 0 to $\overline{\text{XRST}}$ generates a transmit interrupt (XINT). $\overline{\text{XRST}}$ = 0 — The serial port transmitter is reset. Writing a 0 to $\overline{\text{XRST}}$ clears the $\overline{\text{XSREMPTY}}$ bit to 0 and sets the XRDY bit to 1. $\overline{\text{XRST}}$ = 1 — The serial port transmitter is enabled.
5	TXM	0	Transmit Mode. This bit configures the FSX pin as an input (TXM = 0) or as an output (TXM = 1). TXM = 0 — *External frame sync*. The transmitter idles until a frame sync pulse is supplied on the FSX pin. TXM = 1 — *Internal frame sync*. Frame sync pulses are generated internally when data is transferred from the DXR to XSR to initiate data transfers. The internally generated framing signal is synchronous with respect to CLKX.
4	MCM	0	Clock Mode. This bit specifies the clock source for CLKX. MCM = 0 — CLKX is taken from the CLKX pin. MCM = 1 — CLKX is driven by an on-chip clock source. For the SP and the BSP in standard mode, this on-chip clock source is at a frequency of one-fourth of CLKOUT. The BSP also allows the option of generating clock frequencies at additional ratios of CLKOUT. For a detailed description of this feature, see Section 9.3, *Buffered Serial Port (BSP) Interface*, on page 9-32. Note that if MCM = 1 and DLB = 1, a CLKR signal is also supplied by the internal source.
3	FSM	0	Frame Sync Mode. This bit specifies whether frame synchronization pulses (FSX and FSR) are required after the initial frame sync pulse for serial port operation. See subsection 9.2.2, *Serial Port Interface Operation*, on page 9-6 for more details on the frame sync signals. FSM = 0 — *Continuous mode*. Frame sync pulses are not required after the initial frame sync pulse, but they are not ignored; therefore, improperly timed frame syncs may cause errors in serial transfers. See subsection 9.2.6, *Serial Port Interface Exception Conditions*, on page 9-26 for information about serial port operation under various exception conditions. FSM = 1 — *Burst mode*. A frame sync pulse is required on FSX/FSR for the transmission/reception of each word.

Table 9–5. Serial Port Control Register (SPC) Bit Summary (Continued)

Bit	Name	Reset Value	Function	
2	FO	0	Format. This bit specifies the word length of the serial port transmitter and receiver.	
			FO = 0	The data is transmitted and/or received as 16-bit words.
			FO = 1	The data is transferred as 8-bit bytes. The data is transferred with the MSB first. The BSP also allows the capability of 10- and 12-bit transfers. For a detailed description of this feature, see Section 9.3, *Buffered Serial Port (BSP) Interface*, on page 9-32.
1	DLB	0	Digital Loopback Mode. This bit can be used to put the serial port in digital loopback mode.	
			DLB = 0	The digital loopback mode is disabled. The DR, FSR, and CLKR signals are taken from their respective device pins.
			DLB = 1	The digital loopback mode is enabled. The DR and FSR signals are connected to DX and FSX, respectively, through multiplexers, as shown in Figure 9–4(a) and (b) on page 9-12. Additionally, CLKR is driven by CLKX if MCM = 1. If DLB = 1 and MCM = 0, CLKR is taken from the CLKR pin of the device. This configuration allows CLKX and CLKR to be tied together externally and supplied by a common external clock source. The logic diagram for CLKR is shown in Figure 9–4(c) on page 9-12. Note also that in DLB mode, the FSX and DX signals appear on the device pins, but FSR and DR do not. Either internal or external FSX signals may be used in DLB mode, as defined by the TXM bit.
0	Res	0	Reserved. Always read as a 0 in the serial port. This bit performs a function in the TDM serial port discussed in Section 9.4, *Time-Division-Multiplexed (TDM) Serial Port Interface*, on page 9-55.	

Reserved Bit

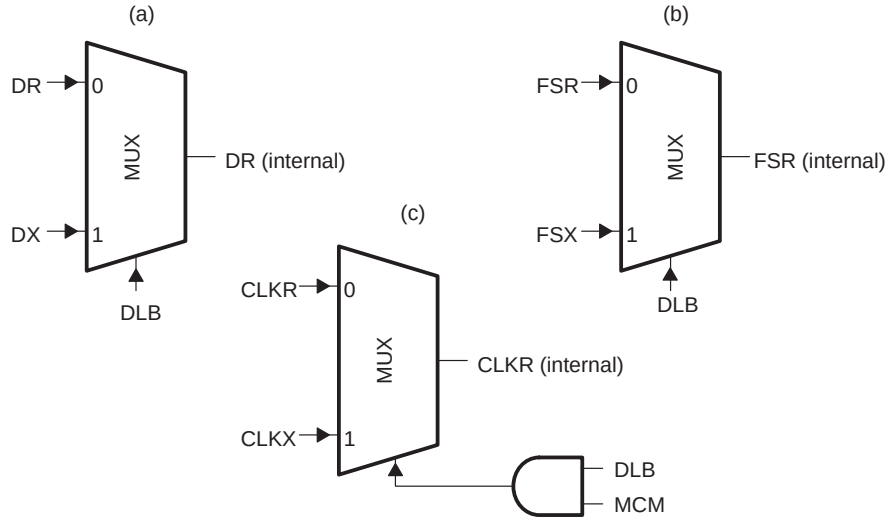
Bit 0 is reserved and is read as 0, although it performs a function in the TDM serial port (discussed in Section 9.4, *Time-Division-Multiplexed (TDM) Serial Port Interface*, on page 9-55).

DLB Bit

The DLB (bit 1) selects digital loopback mode, which allows testing of serial port code with a single ’54x device. When DLB = 1, DR and FSR are connected to DX and FSX, respectively, through multiplexers, as shown in Figure 9–4.

When in loopback mode, CLKR is driven by CLKX if on-chip serial port clock generation is selected (MCM = 1), but if MCM = 0, then CLKR is driven by the external CLKR signal. This allows for the capability of external serial port clock generation in digital loopback mode. If DLB = 0, then normal operation occurs where DR, FSR, and CLKR are all taken from their respective pins.

Figure 9–4. Receiver Signal Multiplexers



FO Bit

The FO (bit 2) specifies whether data is transmitted as 16-bit words (FO = 0) or 8-bit bytes (FO = 1). Note that in the latter case, only the lower byte of whatever is written to DXR is transmitted, and the lower byte of data read from DRR is what was received. To transmit a whole 16-bit word in 8-bit mode, two writes to DXR are necessary, with the appropriate shifts of the value because the upper eight bits written to DXR are ignored. Similarly, to receive a whole 16-bit word in 8-bit mode, two reads from DRR are required, with the appropriate shifts of the value. In the SP, the upper eight bits of DRR are indeterminate in 8-bit receptions; in the BSP, the unused bits of DRR are sign-extended. Additionally, in the BSP, transfers of 10- and 12-bit words are provided for additional flexibility. For a detailed description of this feature, refer to Section 9.3, *Buffered Serial Port (BSP) Interface*, on page 9-32.

FSM Bit

The FSM (bit 3) specifies whether or not frame sync pulses are required in consecutive serial port transmits. If FSM = 1, a frame sync must be present for every transfer, although FSX may be either externally or internally generated depending on TXM. This mode is referred to as burst mode, because there are normally periods of inactivity on the serial port between transmits.

The frequency with which serial port transmissions occur is called packet frequency, and data packets can be 8, 10, 12, or 16 bits long. Therefore, as packet frequency increases, it reaches a maximum that occurs when the time, in serial port clock cycles, from one packet to the next, is equal to the number of bits being transferred. If transmission occurs at the maximum rate for multiple transfers in a row, however, frame sync essentially becomes redundant. Note that frame sync actually becomes redundant in burst mode only at maximum packet frequency with FSX configured as an output (TXM = 1). When FSX is an input (TXM = 0), its presence is required for transmissions to occur.

FSM = 0 selects the continuous mode of operation which requires only an initial frame sync pulse as long as a write to DXR (for transmit), or a read from DRR (for receive), is executed during each transfer. Note that when FSM = 0, frame sync pulses are not required, but they are not ignored, therefore, improperly timed frame syncs may cause errors in serial transfers. The timing of burst and continuous modes is discussed in detail in subsections 9.2.4, 9.2.5, and 9.2.6.

MCM Bit

The serial port clock source is set by MCM (bit 4). If MCM = 0, CLKX is configured as an input and thus accepts an external clock. If MCM = 1, then CLKX is configured as an output, and is driven by an internal clock source. For the SP, and the BSP operating in standard mode, this on-chip clock is at a frequency of one-fourth of CLKOUT. The BSP also allows the option of generating clock frequencies at additional ratios of CLKOUT. For a detailed description of this feature, refer to Section 9.3, *Buffered Serial Port (BSP) Interface*, on page 9-32. Note that the CLKR pin is always configured as an input.

TXM Bit

The transmit frame synchronization pulse source is set by TXM (bit 5). Like MCM, if TXM = 1, FSX is configured as an output and generates a pulse at the beginning of every transmit. If TXM = 0, FSX is configured as an input, and accepts an external frame sync signal. Note that the FSR pin is always configured as an input.

$\overline{\text{XRST}}$ and $\overline{\text{RRST}}$ Bits

The serial port transmitter and receiver are reset with $\overline{\text{XRST}}$ (bit 6) and $\overline{\text{RRST}}$ (bit 7). These signals are active low, so that if $\overline{\text{XRST}}$ = $\overline{\text{RRST}}$ = 0, the serial port is in a reset state. To reset and reconfigure the serial port, a total of two writes to the SPC are required.

- The first write to the SPC should:
  - write a 0 to the $\overline{\text{XRST}}$ and $\overline{\text{RRST}}$ bits
  - write the desired configuration to the remainder of the bits.

- The second write to the SPC should:
  - write 1 to the $\overline{\text{XRST}}$ and $\overline{\text{RRST}}$ bits
  - resend the desired configuration to the remainder of the bits.

The second write takes the serial port out of reset. Note that the transmitter and receiver may be reset individually if desired. When a 0 is written to $\overline{\text{XRST}}$ or $\overline{\text{RRST}}$, activity in the corresponding section of the serial port stops. This minimizes the switching and allows the device to operate with lower power consumption. When $\overline{\text{XRST}}$ = $\overline{\text{RRST}}$ = MCM = 0, power requirements are further reduced since CLKX is no longer driven as an output.

In IDLE2 and IDLE3 mode, SP operation halts as with other parts of the ’54x device. On the BSP, however, if the external serial port clock is being used, operation continues after an IDLE2/3 is executed. This allows power savings to still be realized in IDLE2/3, while still maintaining operation of critical serial port functions if necessary (see Section 9.3, *Buffered Serial Port (BSP) Interface*, on page 9-32 for further information about BSP operation).

It should also be noted that, on the SP, the serial port may be taken out of reset at any time. Depending on the timing of exiting reset, however, a frame sync pulse may be missed. On the BSP, for receive and transmit with external frame sync, a setup of at least one CLKOUT cycle plus 1/2 serial port clock cycle is required prior to FSX being sampled active in standard mode. In autobuffering mode, additional setup is required (see Section 9.3, *Buffered Serial Port (BSP) Interface*, on page 9-32 for further information about BSP initialization timing requirements).

IN0 and IN1 Bits

IN0 (bit 8) and IN1 (bit 9) allow the CLKR and CLKX pins to be used as bit inputs. IN0 and IN1 reflect the current states of the CLKR and CLKX pins. The data on these pins can be sampled by reading the SPC. This can be accomplished using the BIT, BITF, BITT, or CMPM instruction. Note that there is a latency of between 0.5 and 1.5 CLKOUT cycles in duration from CLKR/CLKX switching to the new CLKR/CLKX value being available in the SPC. Note that even if the serial port is reset, IN0 and IN1 can still be used as bit inputs, and DRR and DXR as general-purpose registers.

RRDY and XRDY Bits

Bits 10–13 in the SPC are read-only status bits that indicate various states of serial port operation. Writes and reads of the serial port may be synchronized by polling RRDY (bit 10) and XRDY (bit 11), or by using the interrupts that they generate. A transition from 0 to 1 of the RRDY bit indicates that the RSR contents have been copied to the DRR and that the received data may be read. A receive interrupt (RINT) is generated upon this transition.

A transition from 0 to 1 of the XRDY bit indicates that the DXR contents have been copied to XSR and that DXR is ready to be loaded with a new data word. A transmit interrupt (XINT) is generated upon this transition. Polling XRDY and RRDY in software may either substitute for or complement the use of serial port interrupts (both polling and interrupts may be used together if so desired). Note that with external FSX, on the SP, XSR is loaded directly as a result of loading DXR, while on the BSP, XSR is not loaded until an FSX occurs.

$\overline{\text{XSREMPTY}}$ Bit

The $\overline{\text{XSREMPTY}}$ (bit 12) indicates whether the transmitter has experienced underflow. $\overline{\text{XSREMPTY}}$ is an active low bit; therefore, when $\overline{\text{XSREMPTY}}$ = 0, an underflow has occurred.

Any *one* of the following three conditions causes $\overline{\text{XSREMPTY}}$ to become active ($\overline{\text{XSREMPTY}}$ = 0):

- DXR has not been loaded since the last DXR-to-XSR transfer, and XSR empties (the actual transition of $\overline{\text{XSREMPTY}}$ occurs after the last bit has been shifted out of XSR),
- or the transmitter is reset ($\overline{\text{XRST}}$ = 0),
- or the ’54x device is reset ($\overline{\text{RS}}$ = 0).

When $\overline{\text{XSREMPTY}}$ = 0, the transmitter halts and stops driving DX (the DX pin is in a high-impedance state) until the next frame sync pulse. Note that underflow does not constitute an error condition in the burst mode, although it does in the continuous mode (error conditions are further discussed in subsection 9.2.6, *Serial Port Interface Exception Conditions*, on page 9-26).

The following condition causes $\overline{\text{XSREMPTY}}$ to become inactive ($\overline{\text{XSREMPTY}}$ = 1):

- A write to DXR occurs on the SP, or on the BSP a write to DXR occurs followed by an FSX pulse (see subsection 9.2.4, *Burst Mode Transmit and Receive Operations*, on page 9-17 for further information about transmit timing).

RSRFULL Bit

The RSRFULL (bit 13) indicates whether the receiver has experienced overrun. RSRFULL is an active high bit; therefore, when RSRFULL = 1, RSR is full.

In burst mode (FSM = 1), all three of the following must occur to cause RSRFULL to become active (RSRFULL = 1):

- The DRR has not been read since the last RSR-to-DRR transfer,
- RSR is full,
- and a frame sync pulse appears on FSR.

In continuous mode (FSM = 0), and on the BSP, only the first two conditions are necessary to set RSRFULL:

- The DRR has not been read since the last RSR-to-DRR transfer
- and RSR is full.

Therefore, in continuous mode, and on the BSP, RSRFULL occurs after the last bit has been received.

When RSRFULL = 1, the receiver halts and waits for the DRR to be read, and any data sent on DR is lost. On the SP, the data in RSR is preserved; on the BSP, the RSR contents are lost.

Any *one* of the following three conditions causes RSRFULL to become inactive (RSRFULL = 0):

- The DRR is read,
- or the serial port is reset ($\overline{\text{RRST}}$ = 0),
- or the ’54x device is reset ($\overline{\text{RS}}$ = 0).

SOFT and FREE Bits

Soft (bit 14) and Free (bit 15) are special emulation bits that determine the state of the serial port clock when a breakpoint is encountered in the high-level language (HLL) debugger. If the Free bit is set to 1, then upon a software breakpoint, the clock continues to run (free runs) and data is still shifted out. When Free = 1, the Soft bit is a *don't care*. If the Free bit is cleared to 0, then the Soft bit takes effect. If the Soft bit is cleared to 0, then the clock stops immediately, thus aborting any transmission. If the Soft bit is set to 1 and a transmission is in progress, the transmission continues until completion of the transfer, and then the clock halts. These options are listed in Table 9–6.

The receive side functions in a similar fashion. Note that if an option other than *immediate stop* (Soft = Free = 0) is chosen, the receiver continues running and an overflow error is possible. The default value for these bits is *immediate stop*.

Table 9–6. Serial Port Clock Configuration

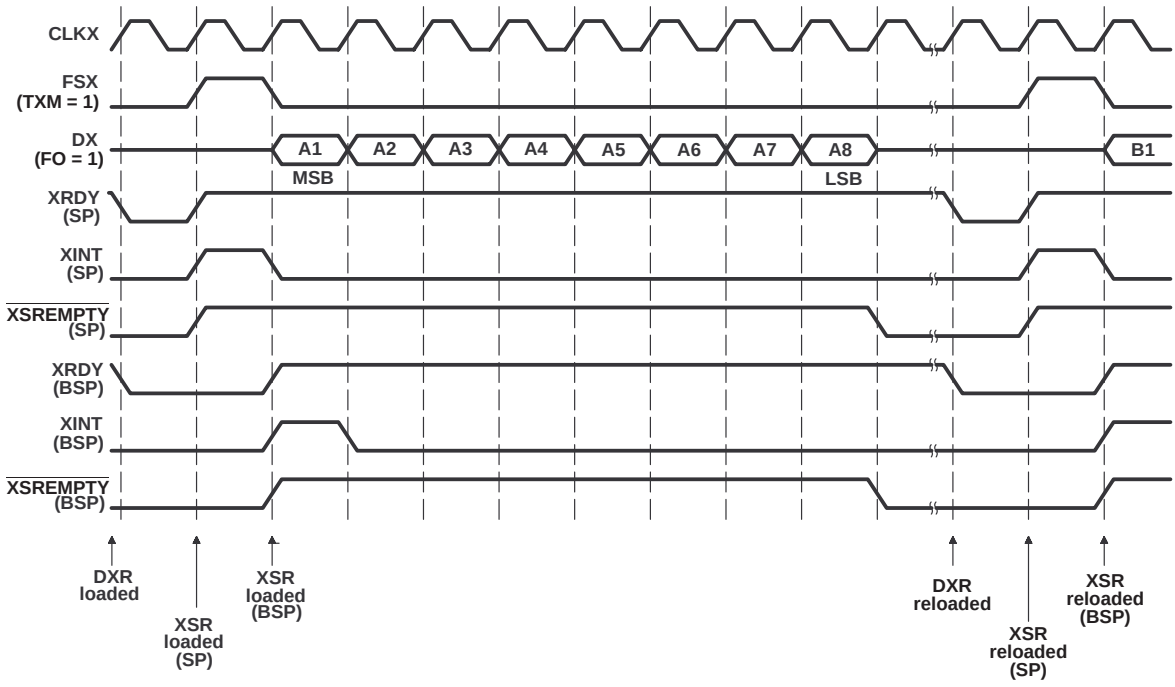
Free	Soft	Serial Port Clock Configuration
0	0	Immediate stop, clocks are stopped. (Reset values)
0	1	Transmitter stops after completion of the current word. The receiver is not affected.
1	X	Free run.

Note: X = Don't care

9.2.4 Burst Mode Transmit and Receive Operations

In burst mode operation, there are periods of serial port inactivity between packet transmits. The data packet is marked by the frame sync pulse occurring on FSX (see Figure 9–5). On the transmit device, the transfer is initiated by a write to DXR. The value in DXR is then transferred to XSR, and, upon a frame sync pulse on FSX (generated internally or externally depending on TXM), the value in XSR is shifted out and driven on the DX pin. Note that on the SP, the DXR to XSR transfer occurs on the second rising edge of CLKX after DXR is loaded, while on the BSP this transfer does not occur until an FSX occurs, when FSX is external. When FSX is internal on the BSP, the DXR to XSR transfer and generation of FSX occur directly after loading DXR. On both the SP and the BSP, once XSR is loaded with the value from DXR, XRDY goes high, generating a transmit interrupt (XINT) and setting $\overline{\text{XSREMPTY}}$ to a 1.

Figure 9–5. Burst Mode Serial Port Transmit Operation



Note that in both the SP and the BSP, DXR to XSR transfers occur only if the XSR is empty and the DXR has been loaded since the last DXR to XSR transfer. If DXR is reloaded before the old DXR contents have been transferred to XSR, the previous DXR contents are overwritten. Accordingly, unless overwriting DXR is intended, the DXR should only be loaded if XRDY = 1. This is assured if DXR writes are made only in response to a transmit interrupt or polling XRDY.

It should be noted that in the following discussions, the timings are slightly different for internally (TXM = 1, FSX is an output) and externally (TXM = 0, FSX is an input) generated frame syncs. This distinction is made because in the former case, the frame sync pulse is generated by the transmitting device as a direct result of a write to DXR. In the latter case, there is no such direct effect. Instead, the transmitting device must write to DXR and wait for an externally generated frame sync.

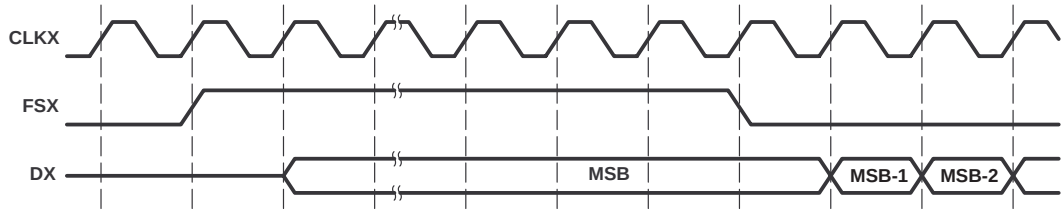
If internal frame sync pulse generation is selected (TXM = 1), a frame sync pulse is generated on the second rising edge of CLKX following a write to DXR. For externally generated frame syncs, the events described here will occur as soon as a properly timed frame sync pulse occurs (see the data sheet for detailed serial port interface timings).

On the next rising edge of CLKX after FSX goes high, the first data bit (MSB first) is driven on the DX pin. Thus, if the frame sync pulse is generated internally (TXM = 1), there is a 2-CLKX cycle latency (approximately) after DXR is loaded, before the data is driven on the line. If frame sync is externally generated, data transmission is delayed indefinitely after a DXR load until the FSX pulse occurs (this is described in further detail later in this subsection). With the falling edge of frame sync, the rest of the bits are shifted out. When all the bits are transferred, DX enters a high-impedance state.

At the end of each transmission, if DXR was not reloaded when XINT was generated, $\overline{\text{XSREMPTY}}$ becomes active (low) at this point, indicating underflow. With externally generated frame sync, if $\overline{\text{XSREMPTY}}$ is active and a frame sync pulse is generated, any old data in the DXR is transmitted. This is explained in detail in subsection 9.2.6, *Serial Port Interface Exception Conditions*, on page 9-26.

Note that the first data bit transferred could have variable length if frame sync is generated externally and does not fall within one CLKX cycle (this is illustrated in Figure 9–6). Internally generated frame syncs are assured by ’54x timings to be one CLKX cycle in duration.

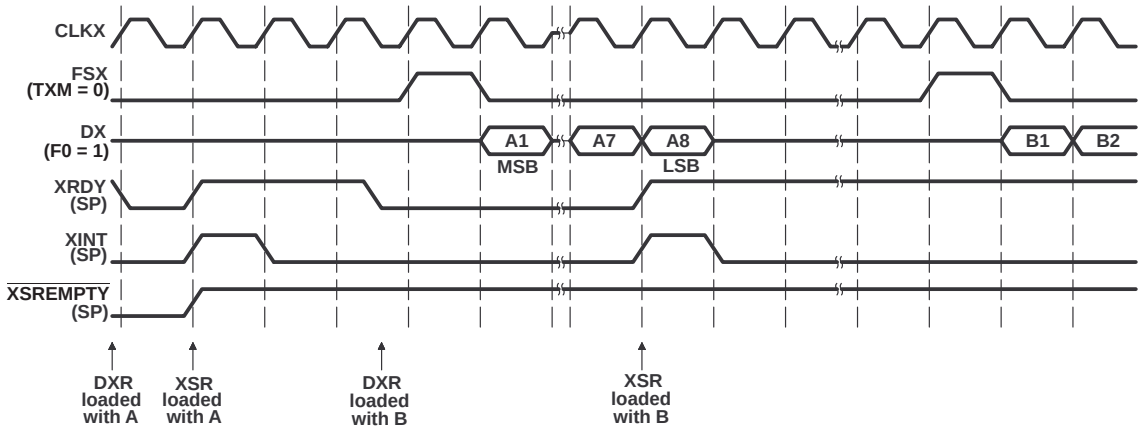
Figure 9–6. Serial Port Transmit With Long FSX Pulse



Serial port transmit with external frame sync pulses is similar to that with internal frame sync, with the exception that transfers do not actually begin until the external frame sync occurs. If the external frame sync occurs many CLKX cycles after DXR is loaded, however, the double buffer is filled and frozen until frame sync appears.

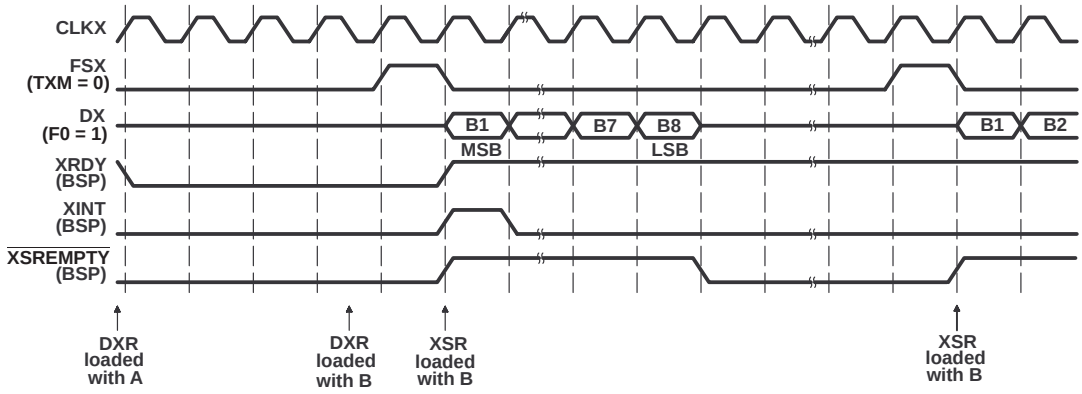
On the SP (Figure 9–7), when the delayed frame sync occurs, A is transmitted on DX; after the transmit, a DXR-to-XSR copy of B occurs, XINT is generated, and again, the transmitter remains frozen until the next frame sync. When frame sync finally occurs, B is transmitted on DX. Note that when B is loaded into DXR, a DXR-to-XSR copy of B does not occur immediately because A has not been transmitted, and no XINT is generated. Any subsequent writes to DXR before the next delayed frame sync occurs overwrite B in the DXR.

Figure 9–7. Burst Mode Serial Port Transmit Operation With Delayed Frame Sync in External Frame Sync Mode (SP)



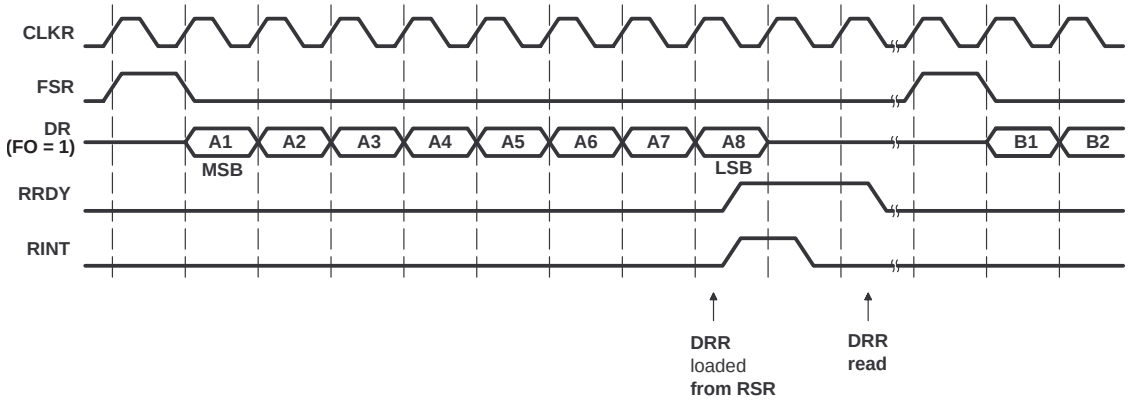
On the BSP (Figure 9–8), since DXR was reloaded with B shortly after being loaded with A when the delayed frame sync finally occurs, B is transmitted on DX. After the transmit, the transmitter remains frozen until the next frame sync. When frame sync finally occurs, B is again transmitted on DX. Note that when B is loaded into DXR, a DXR-to-XSR copy of B does not occur immediately since the BSP requires a frame sync to initiate transmitting. Any subsequent writes to DXR before the next delayed frame sync occurs overwrite B in the DXR.

Figure 9–8. Burst Mode Serial Port Transmit Operation With Delayed Frame Sync in External Frame Sync Mode (BSP)



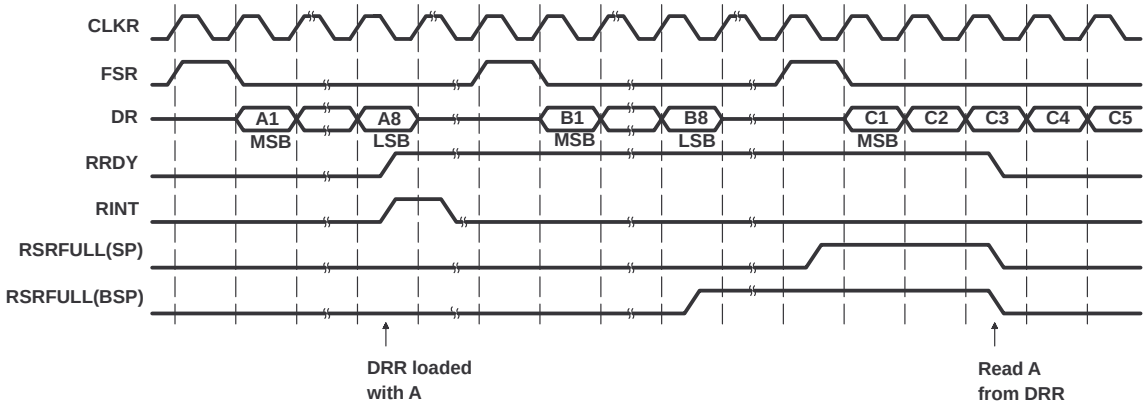
During a receive operation, shifting into RSR begins on the falling edge of the CLKR cycle after frame sync has gone low (as shown in Figure 9–9). Then, as the last data bit is being received, the contents of the RSR are transferred to the DRR on the falling edge of CLKR, and RRDY goes high, generating a receive interrupt (RINT).

Figure 9–9. Burst Mode Serial Port Receive Operation



If the DRR from a previous receive has not been read, and another word is received, no more bits can be accepted without causing data corruption since DRR and RSR are both full. In this case, the RSRFULL bit is set indicating this condition. On the SP, this occurs with the next FSR; on the BSP, RSRFULL is set on the falling edge of CLKR during the last bit received. RSRFULL timing on both the SP and BSP is shown in Figure 9–10.

Figure 9–10. Burst Mode Serial Port Receive Overrun



Unlike transmit underflow, overrun (RSRFULL = 1) constitutes an actual error condition. While DRR contents are preserved in overrun, its occurrence can often result in loss of other received data.

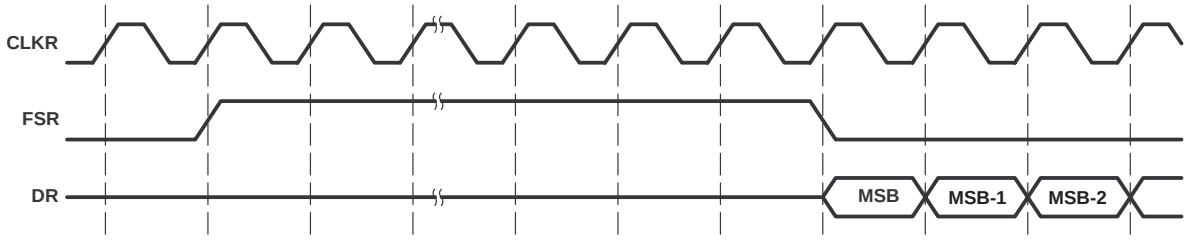
Overrun is handled differently on the SP and on the BSP. On the SP, the contents of RSR are preserved on overrun, but since RSRFULL is not set to 1 until the next FSR occurs after the overflowing reception, incoming data usually begins being lost as soon as RSRFULL is set. Data loss can only be avoided if RSRFULL is polled in software and the DRR is read immediately after RSRFULL is set to 1. This is normally possible only if the CLKR frequency is slow with respect to CLKOUT, since RSRFULL is set on the falling edge of CLKR during FSR, and data begins being received on the following rising edge of CLKR. The time available for polling RSRFULL and reading the DRR to avoid data loss is, therefore, only half of one CLKR cycle.

On the BSP, RSRFULL is set on the last valid bit received, but the contents of RSR are never transferred to DRR, therefore, the complete transferred word in RSR is lost. If the DRR is read (clearing RSRFULL) before the next FSR occurs, subsequent transfers can be received properly.

Overrun and various other serial port exception conditions such as the occurrence of frame sync during a receive are discussed in further detail in subsection 9.2.6, *Serial Port Interface Exception Conditions*, on page 9-26.

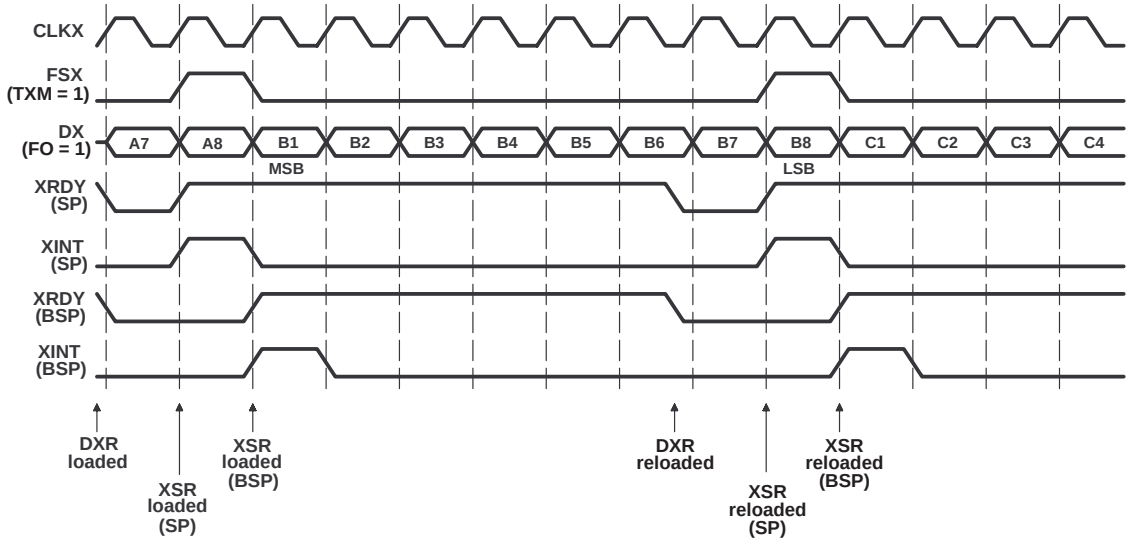
If the serial port receiver is provided with FSR pulses significantly longer than one CLKR cycle, timing of data reception is effected in a similar fashion as with long FSX pulses. With long FSR pulses, however, the reception of all bits, including the first one, is simply delayed until FSR goes low. Serial port receive operation with a long FSR pulse is illustrated in Figure 9–11.

Figure 9–11. Serial Port Receive With Long FSR Pulse



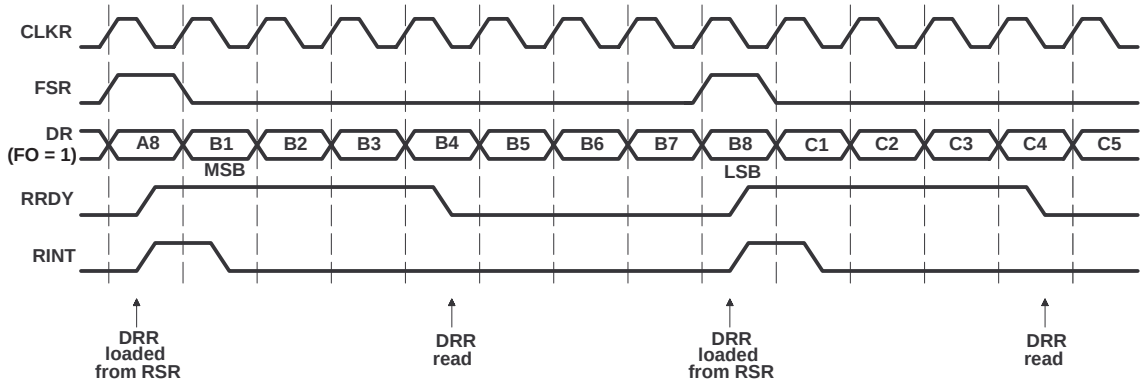
Note that if the packet transmit frequency is increased, the inactivity period between the data packets for adjacent transfers decreases to zero. This corresponds to a minimum period between frame sync pulses (equivalent to 8 or 16 CLKX/R cycles, depending on FO) that corresponds to a maximum packet frequency at which the serial port may operate. At maximum packet frequency, transmit timing is a compressed version of Figure 9–5, as shown in Figure 9–12.

Figure 9–12. Burst Mode Serial Port Transmit at Maximum Packet Frequency



At maximum packet frequency, the data bits in consecutive packets are transmitted contiguously with no inactivity between bits. The frame sync pulse overlaps the last bit transmitted in the previous packet. Maximum packet frequency receive timing is similar and is shown in Figure 9–13.

Figure 9–13. Burst Mode Serial Port Receive at Maximum Packet Frequency



As shown in Figure 9–12 and Figure 9–13, with the transfer of multiple data packets at maximum packet frequency in burst mode, packets are transmitted at a constant rate, and the serial port clock provides sufficient timing information for the transfer, which permits a continuous stream of data. Therefore, the frame sync pulses are essentially redundant. Theoretically, then, only an initial frame sync signal is required to initiate the multipacket transfer. The '54x does support operation of the serial port in this fashion, referred to as continuous mode, which is selected by clearing the FSM bit in the SPC to 0. Continuous mode serial port operation is described in detail in subsection 9.2.5, *Continuous Mode Transmit and Receive Operations*.

9.2.5 Continuous Mode Transmit and Receive Operations

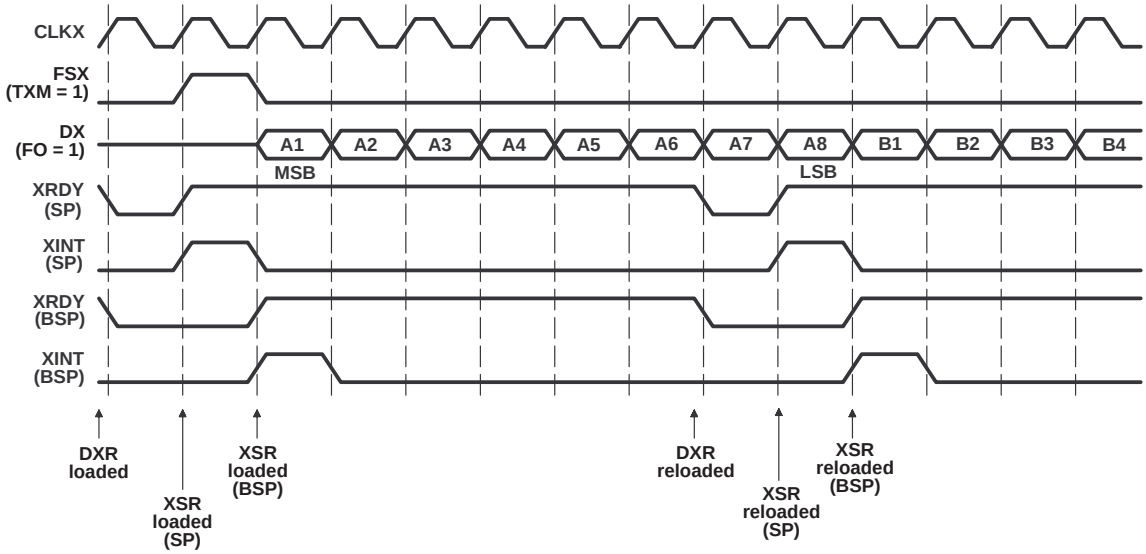
In continuous mode, a frame sync on FSX/FSR is not necessary for consecutive packet transfers at maximum packet frequency after the initial pulse. Continuous mode is selected by setting FSM = 0. Note that when FSM = 0, frame sync pulses are not required, but they are not ignored, therefore, improperly timed frame syncs may cause errors in serial transfers. Serial port operation under various error conditions is described in detail in subsection 9.2.6, *Serial Port Interface Exception Conditions*, on page 9-26.

In continuous mode transmission, one frame sync is generated following the first DXR load, and no further frame syncs are generated. As long as DXR is reloaded once every transmission, continuous transfers are maintained. Failing to update DXR causes the serial port to halt, as in the burst mode case ($\overline{\text{XSREMPTY}}$ becomes asserted, etc). If DXR is reloaded after a halt, the device begins continuous mode transmission again and generates a single FSX, assuming that internal frame sync generation is selected.

The distinction between internal and external frame syncs for continuous mode is similar to that of burst mode, as discussed in subsection 9.2.4, *Burst Mode Transmit and Receive Operations*, on page 9-17. If frame sync is externally generated (TXM = 0), then when DXR is loaded, the appearance of the frame sync pulse initiates continuous mode transmission. Continuous mode transmission may be discontinued (and burst mode resumed) only by reconfiguring and resetting the serial port (see subsection 9.2.2, *Serial Port Interface Operation*, on page 9-6). Simply changing the FSM bit during transmit or halt will not properly switch to burst mode.

Continuous mode transmit timing, shown in Figure 9–14, is similar to maximum packet frequency transmission in burst mode as shown in Figure 9–12. The major difference is the lack of a frame sync pulse after the initial one. As long as DXR is updated once per transmission, this mode will continue. Overwrites to DXR behave just as in burst mode; the last data written will be transmitted. XSR operation is the same as in burst mode. A new external FSX pulse will abort the present transmission, cause one data packet to be lost, and initiate a new continuous mode transmit. This is explained in more detail in subsection 9.2.6, *Serial Port Interface Exception Conditions*, on page 9-26.

Figure 9–14. Continuous Mode Serial Port Transmit



Continuous mode reception is similar to the transmit operation. After the initial frame sync pulse on FSR, no further frame syncs are required. This mode will continue as long as DRR is read every transmission. If DRR is not read by the end of the next transfer, the receiver will halt, and RSRFULL is set, indicating overrun. See subsection 9.2.6, *Serial Port Interface Exception Conditions*.

Overrun in continuous mode effects the SP and the BSP differently. On the SP, once overrun has occurred, reading DRR will restart continuous mode at the next word/byte boundary after DRR is read; no new FSR pulse is required. On the BSP, continuous mode reception does not resume until DRR is read and an FSR occurs.

Continuous mode reception may only be discontinued by reconfiguring and resetting the serial port. Simply changing the FSM bit during a reception or halt will not properly switch to burst mode. Continuous mode receive timing is shown in Figure 9–15.

Figure 9–15. Continuous Mode Serial Port Receive

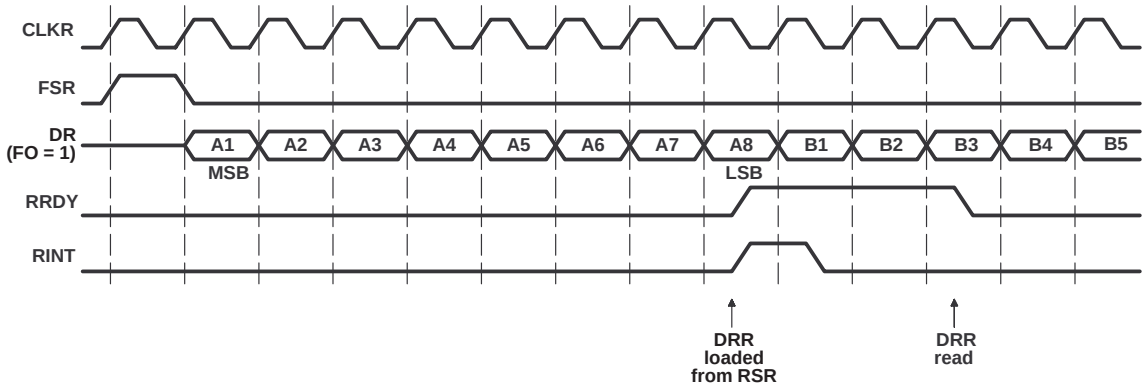


Figure 9–15 shows only one frame sync pulse; otherwise, it is similar to Figure 9–13. If a pulse occurs on FSR during a transfer (an error), then the receive operation is aborted, one packet is lost, and a new receive cycle is begun. This is discussed in more detail in subsection 9.2.2, *Serial Port Interface Operation*, on page 9-6 and in subsection 9.2.6, *Serial Port Interface Exception Conditions*.

9.2.6 Serial Port Interface Exception Conditions

Exception (or error) conditions result from an unexpected event occurring on the serial port. These conditions are operational aberrations such as overrun, underflow, or a frame sync pulse during a transfer. Understanding how the serial port handles these errors and the state it acquires during these error conditions is important for efficient use of the serial port. Because the error conditions differ slightly in burst and continuous modes, they are discussed separately.

Burst Mode

In burst mode, one type of error condition (presented in subsection 9.2.2, *Serial Port Interface Operation*) is receive overrun, indicated by the RSRFULL flag. This flag is set when the device has not read incoming data and more data is being sent. If this condition occurs, the processor halts serial port receives until DRR is read. Thus, any further data sent may be lost.

Overrun is handled differently on the SP and on the BSP. On the SP, the contents of RSR are preserved on overrun, but since RSRFULL is not set to 1 until the next FSR occurs after the overflowing reception, incoming data usually begins being lost as soon as RSRFULL is set. Data loss can only be avoided if RSRFULL is polled in software and the DRR is read immediately after

RSRFULL is set to 1. This is normally possible only if the CLKR frequency is slow with respect to CLKOUT, since RSRFULL is set on the falling edge of CLKR during FSR, and data begins being received on the following rising edge of CLKR. The time available for polling RSRFULL and reading the DRR to avoid data loss is, therefore, only half of one CLKR cycle.

On the BSP, RSRFULL is set on the last valid bit received, but the contents of RSR are never transferred to DRR, therefore, the complete transferred word in RSR is lost. If the DRR is read (clearing RSRFULL) before the next FSR occurs, subsequent transfers can be received properly.

Another type of receive error is caused if frame sync occurs during a receive (that is, data is being shifted into RSR from DR). If this happens, the present receive is aborted and a new one begins. Thus, the data that was being loaded into RSR is lost, but the data in DRR is not (no RSR-to-DRR copy occurs). Burst mode serial port receiver behavior under normal and error conditions for the SP is shown in Figure 9–16 and for the BSP is shown in Figure 9–17.

Figure 9–16. SP Receiver Functional Operation (Burst Mode)

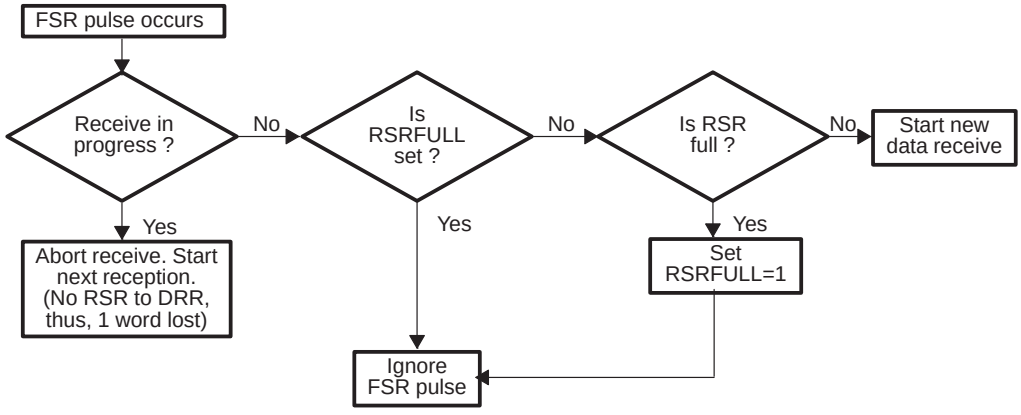
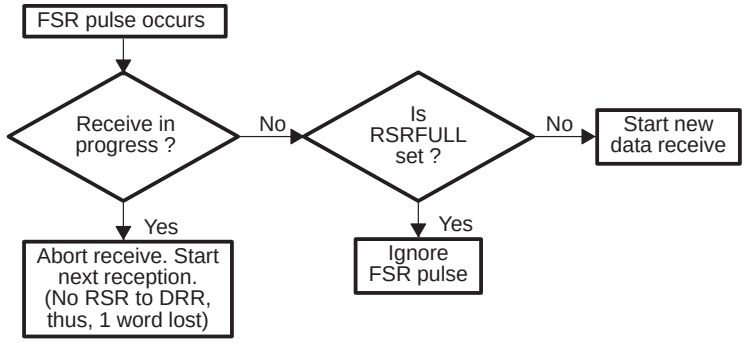
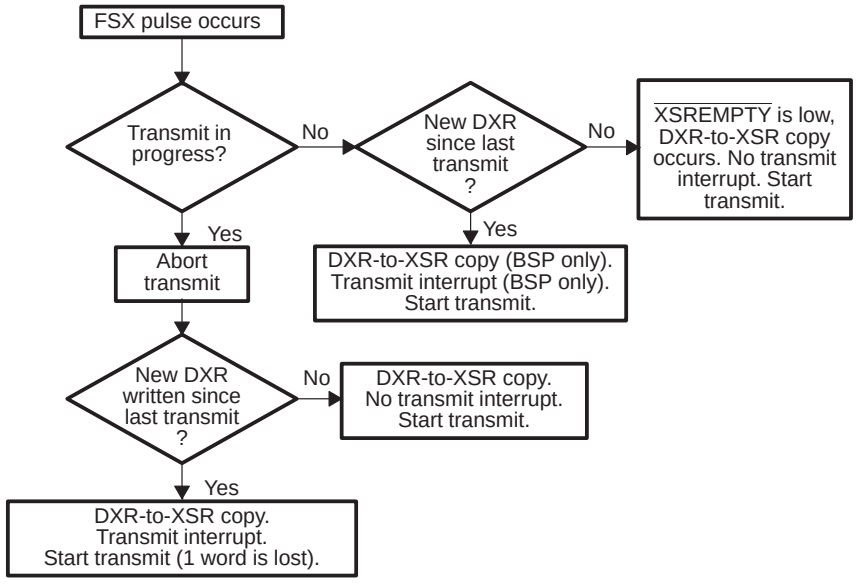


Figure 9–17. BSP Receiver Functional Operation (Burst Mode)



Transmitter exception conditions in burst mode may occur for several possible reasons. Underflow, which is described in subsection 9.2.3, *Configuring the Serial Port Interface*, on page 9-7 is an exception condition that may occur in burst mode, however, underflow is not normally considered an error. An exception condition that causes errors in transmitted data occurs when frame sync pulses occur at inappropriate times during a transfer. If a transmit is in progress (that is, XSR data is being driven on DX) when a frame sync pulse occurs, the transmission is aborted, and the data in XSR is lost. Then, whatever data is in DXR at the time of the frame sync pulse is transferred to XSR (DXR-to-XSR copy) and is transmitted. Note, however, that in this case an XINT is generated only if the DXR has been written to since the last transmit. Also, if $\overline{\text{XSREMPTY}}$ is active and a frame sync pulse occurs, the old data in DXR is shifted out. Figure 9–18 summarizes serial port transmit behavior under error and nonerror conditions. Note that if an FSX occurs when no transmit is in progress, and DXR has been reloaded since the last transmit, the DXR-to-XSR copy and generation of transmit interrupt occur at this point only on the BSP. On the SP, these two events occur at the time the DXR was reloaded.

Figure 9–18. SP/BSP Transmitter Functional Operation (Burst Mode)

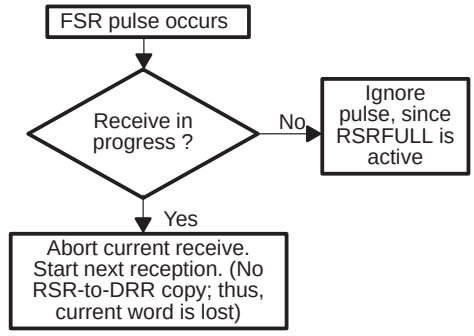


Continuous Mode

In continuous mode, errors take on a broader meaning, since data transfer is intended to occur at all times. Thus, underflow ($\overline{\text{XSREMPTY}}$ = 0) constitutes an error in continuous mode because data will not be transmitted. As in burst mode, overrun (RSRFULL = 1) is also an error, and in continuous mode, both overrun and underflow cause the serial port receive or transmit sections, respectively, to halt (see subsection 9.2.3, *Configuring the Serial Port Interface*, on page 9-7 for a description of these conditions). Fortunately, underflow and overrun errors may not be catastrophic; they can often be corrected simply by reading DRR or writing to DXR.

The SP and the BSP are affected differently when overrun occurs in continuous mode. In the SP, when DRR is read to deactivate RSRFULL, a frame sync pulse is not required in order to resume continuous mode operation. The receiver keeps track of the transfer word boundary, even though it is not receiving data. Therefore, when the RSRFULL flag is deactivated by a read from DRR, the receiver begins reading from the correct bit. On the BSP, since an FSR pulse is required to restart continuous reception, this also reestablishes the proper bit alignment, in addition to restarting reception. Figure 9–19 shows receiver functional operation in continuous mode.

Figure 9–19. SP/BSP Receiver Functional Operation (Continuous Mode)

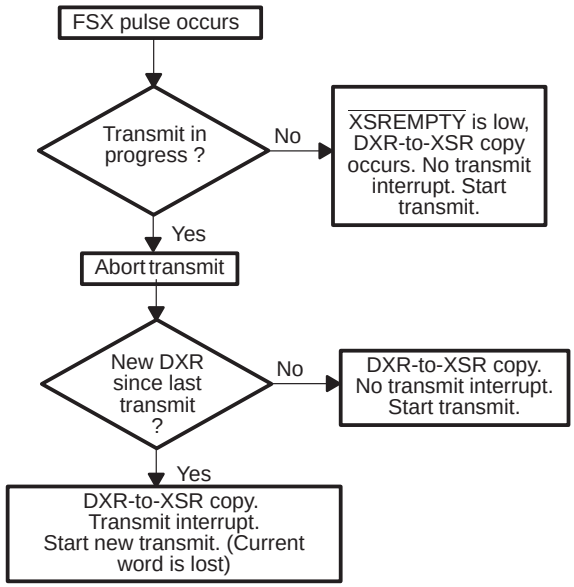


During a receive in continuous mode, if a frame sync pulse occurs, this causes a receive abort condition, and one packet of data is lost (this is caused because the frame sync pulse resets the RSR bit counter). The data present on DR then begins being shifted into RSR, starting again from the first bit. Note that if a frame sync occurs after deactivating the RSRFULL flag by reading DRR, but before the beginning of the next word boundary, this also creates a receive abort condition.

Another cause for error is the appearance of extraneous frame syncs during a transmission. After the initial frame sync in continuous mode, no others are required; if an improperly timed frame sync pulse occurs during a transmit, the current transfer (that is, serially driving XSR data onto DX) is aborted, and data in XSR is lost. A new transmit cycle is initiated, and transfers continue as long as the DXR is updated once per transmission afterward. Figure 9–20 shows continuous mode transmitter functional operation.

Note that if $\overline{\text{XSREMPTY}}$ is active in continuous mode and an external frame sync occurs, the previous DXR data is transmitted as in burst mode operation.

Figure 9–20. SP/BSP Transmitter Functional Operation (Continuous Mode)



9.2.7 Example of Serial Port Interface Operation

As an illustration of the proper operation of a standard serial port, Example 9–1 and Example 9–2 define a sequence of actions. This illustration is based on the use of interrupts to handle the normal I/O between the serial port and CPU. The ’545 peripheral configuration has been used as a reference for these examples.

Example 9–1. Serial Port Initialization Routine

	Action	Description
1)	Reset and initialize the serial port by writing 0038h (or 0008h) to SPC.	This places both the transmit and receive portions of the serial port in reset and sets up the serial port to operate with internally generated FSX and CLKX signals and FSX/FSR required for transmit/receive of each 16-bit word. (The alternative is used if another device will provide FSX and CLKX.)
2)	Clear any pending serial port interrupts by writing 00C0h to IFR.	Eliminate any interrupts that may have occurred before initialization.
3)	Enable the serial port interrupts by ORing 00C0h with IMR.	Enable both transmit and receive interrupts. A common alternative when transmit and receive are synchronized to one another is to enable only one or the other, by ORing 0080h or 0040h with IMR, and performing both I/O operations with the same interrupt service routine (ISR).
4)	Enable interrupts globally (if necessary) by clearing the INTM bit in ST1.	Interrupts must be globally enabled for the CPU to respond.
5)	Start the serial port by writing 00F8h (or 00C8h) to SPC.	This takes both the transmit and receive portions of the serial port out of reset and starts operations with the conditions defined in step 1.
6)	Write the first data value to DXR. (If the serial port is connected to the serial port of another processor and this processor will be generating FSX, a handshake must be performed prior to writing the first data value to DXR.)	This initiates serial port transmit operations if FSX and CLKX are internally generated or prepares the serial port transmit for operation when the first FSX arrives.

Example 9–2. Serial Port Interrupt Service Routine

	Action	Description
1)	Save any context that may be modified on the stack.	The operating context of the interrupted code must be maintained.
2)	Read the DRR or write the DXR or both. The data read form DRR should be written to a predetermined location in memory. The data written to DXR should be read from a predetermined location in memory.	Read the received data for the receive ISR. Write the new transmit data for the transmit ISR. Or, do both if the ISR is combined for transmit and receive.
3)	Restore the context that was saved in step 1.	The operating context of the interrupted code must be maintained.
4)	Return from the ISR with an RETE to reenable interrupts.	Interrupts must be reenabled for the CPU to respond to the next interrupt.

9.3 Buffered Serial Port (BSP) Interface

The buffered serial port (BSP) is made up of a full-duplex, double-buffered serial port interface, which functions in a similar manner to the ’54x standard serial port, and an autobuffering unit (ABU) (see Figure 9–21). The serial port section of the BSP is an enhanced version of the ’54x standard serial port. The ABU is an additional section of logic which allows the serial port section to read/write directly to ’54x internal memory independent of the CPU. This results in a minimum overhead for serial port transfers and faster data rates.

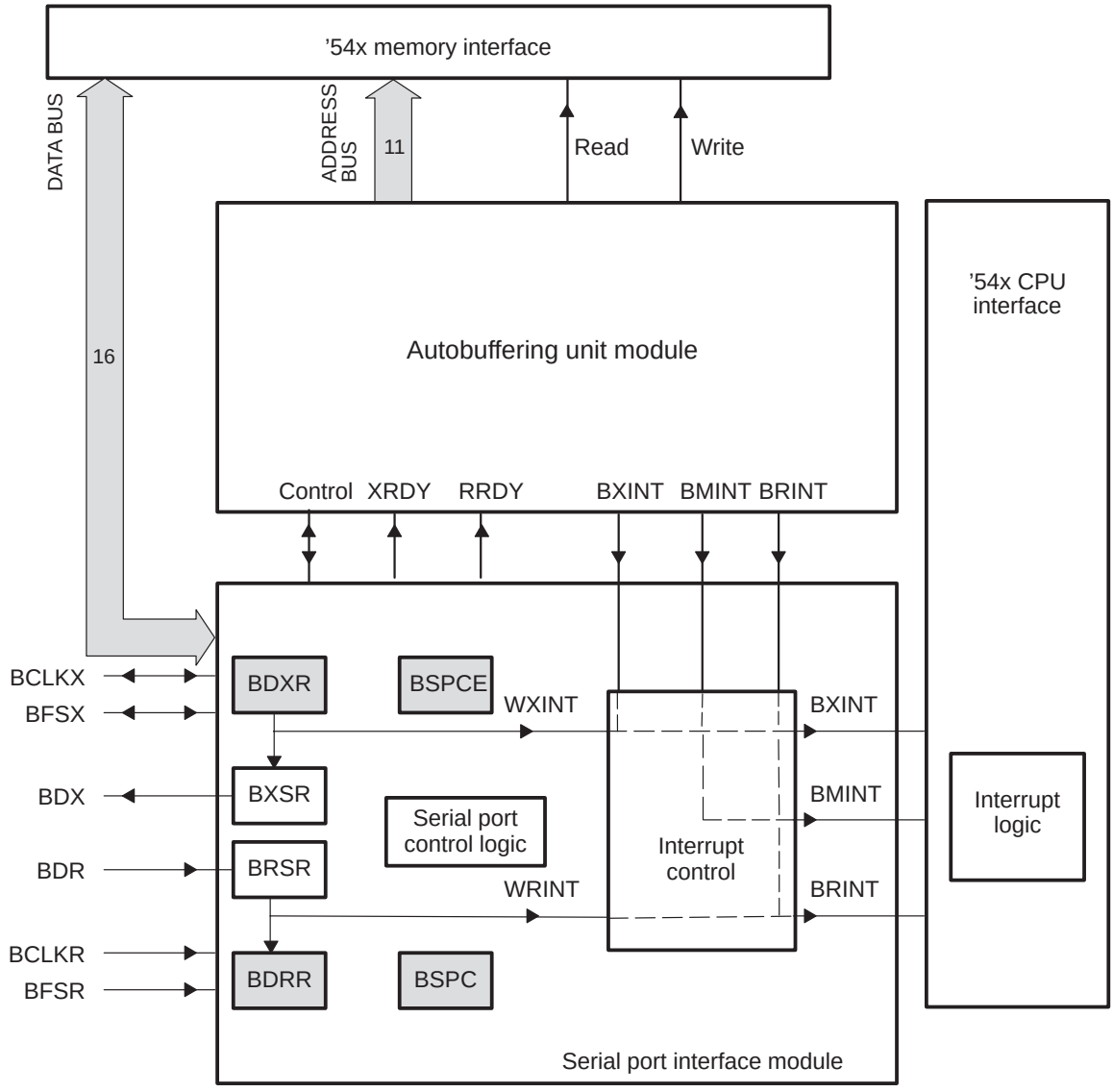
The full duplex BSP serial interface provides direct communication with serial devices such as codecs, serial A/D converters, and other serial devices with a minimum of external hardware. The double-buffered BSP allows transfer of a continuous communication stream in 8-,10-,12- or 16-bit data packets. Frame synchronization pulses as well as a programmable frequency serial clock can be provided by the BSP for transmission and reception. The polarity of frame sync and clock signals are also programmable. The maximum operating frequency is CLKOUT (40 Mbit/s at 25 ns, 50 Mbit/s at 30 ns). The BSP transmit section includes a pulse code modulation (PCM) mode that allows easy interface with a PCM line. Operation of the BSP in standard (nonbuffered) mode is detailed in subsection 9.3.1 on page 9-34.

The ABU has its own set of circular addressing registers, each with corresponding address generation units. Memory for transmit and receive buffers resides within a special 2K word block of ’54x internal memory. This memory can also be used by the CPU as general purpose storage, however, this is the only memory block in which autobuffering can occur.

Using autobuffering, word transfers occur directly between the serial port section and the ’54x internal memory automatically using the ABU embedded address generators. The length and starting addresses of the buffers within the 2K block are programmable, and a buffer empty/full interrupt can be generated to the CPU. Buffering can easily be halted using the autodisabling capability. ABU operation is detailed in subsection 9.3.2 on page 9-39.

The BSP autobuffering capability can be separately enabled for the transmit and receive sections. When autobuffering is disabled (standard mode), data transfers with the serial port section occur under software control in the same fashion as with the standard ’54x serial port. In this mode, the ABU is transparent, and the WXINT and WRINT interrupts generated each time a word is transmitted or received are sent to the CPU as transmit interrupt (BXINT) and receive interrupt (BRINT). When autobuffering is enabled, the BXINT and BRINT interrupts are only generated to the CPU each time half of the buffer is transferred.

Figure 9–21. BSP Block Diagram



As mentioned previously, most aspects of BSP operation are similar to that of the '54x standard serial port. Section 9.2, *Serial Port Interface*, on page 9-3 discusses operation of both the '54x standard serial port and the BSP in standard mode. Since standard mode BSP operation is a superset of standard serial port operation, Section 9.2, *Serial Port Interface*, should first be studied before the rest of this section is read.

System considerations of BSP operation such as initialization and low power modes are discussed in subsection 9.3.3 on page 9-48.

9.3.1 BSP Operation in Standard Mode

BSP operation in standard mode is discussed in Section 9.2, *Serial Port Interface*, on page 9-3. This subsection summarizes the differences between serial port operation and standard mode BSP operation and describes the enhanced features that the BSP offers. The enhanced BSP features are available both in standard mode and in autobuffering mode. ABU is discussed in subsection 9.3.2 on page 9-39. Information presented in this section assumes familiarity with standard mode operation as described in Section 9.2, *Serial Port Interface*.

The BSP uses its own dedicated memory-mapped data transmit, data receive and serial port control registers (BDXR, BDRR, and BSPC). The BSP also utilizes an additional control register, the BSP control extension register (BSPCE), in implementing its enhanced features and controlling the ABU. The BDRR, BDXR, and BSPC registers function similarly to their counterparts in the serial port as described in Section 9.2, *Serial Port Interface*. As with the serial port, the BSP transmit and receive shift registers (BXSR and BRSR) are not directly accessible in software but facilitate the double-buffering capability. If the serial port is not being used, the BDXR and the BDRR registers can be used as general purpose registers. In this case, BFSR should be set to an inactive state to prevent a possible receive operation from being initiated. Note, however, that program access to BDXR or BDRR is limited when autobuffering is enabled for transmit or receive, respectively. BDRR can only be read, and BDXR can only be written when the ABU is disabled. BDRR can only be written when the BSP is in reset. BDXR can be read any time.

The buffered serial port registers are summarized in Table 9–7. The ABU utilizes several additional registers which are discussed in subsection 9.3.2, *Autobuffering Unit (ABU) Operation*, on page 9-39.

Table 9–7. Buffered Serial Port Registers

Address	Register	Description
†	BDRR	16-bit BSP data receive register
†	BDXR	16-bit BSP data transmit register
†	BSPC	16-bit BSP control register
†	BSPCE	16-bit BSP control extension register
—	BRSR	16-bit BSP data receive shift register
—	BXSR	16-bit BSP data transmit shift register

† See Section 8.1, *Peripheral Memory-Mapped Registers*.

9.3.1.1 Differences Between Serial Port and BSP Operation in Standard Mode

The differences between serial port and BSP operation in standard mode are discussed in detail in the standard mode serial port operation (Section 9.2 on page 9-3). These differences relate primarily to boundary conditions, however, in some systems, these differences may be significant. The differences are summarized in Table 9–8.

Table 9–8. Differences Between Serial Port and BSP Operation in Standard Mode

Condition	Serial Port	BSP
RSRFULL is set.	RSRFULL is set when RSR is full and then an FSR occurs, except in continuous mode where RSRFULL is set as soon as RSR is full.	RSRFULL is set as soon as BRSR is full.
Preservation of data in RSR on overrun.	RSR contents are preserved on overrun.	BRSR contents are not preserved on overrun.
Continuous mode receive restart after overrun.	Receive restarts as soon as DRR is read (see subsection 9.2.6, *Serial Port Interface Exception Conditions*, on page 9-26).	Receive does not restart until BDRR is read and then a BFSR occurs.
Sign extension in DRR on 8-, 10-, or 12-bit transfers.	No	Yes
XSR load, $\overline{\text{XSREMPTY}}$ clear, XRDY/XINT generation.	Occur when DXR is loaded.	Occur when when a BFSX occurs after BDXR is loaded.
Program accessibility to DXR and DRR.	DRR and DXR can be read or written under program control at any time. Note that caution should be exercised when reads and writes of the DRR may be close in time to serial port receptions. In this case, a DRR read may not yield the result that was previously written by the program. Also note that rewrites of DXR may cause loss (and therefore non-transmission) of previously written data depending on the relative timing of the writes and FSX (see subsection 9.2.4, *Burst Mode Transmit and Receive Operations*, on page 9-17).	BDRR can only be read and BDXR can only be written when the ABU is disabled. BDRR can only be written when the BSP is in reset. BDXR can be read any time. The same precautions with regard to reads and writes to these registers apply as in serial port.
Maximum serial port clock rate.	CLKOUT/4	CLKOUT

Table 9–8. Differences Between Serial Port and BSP Operation in Standard Mode (Continued)

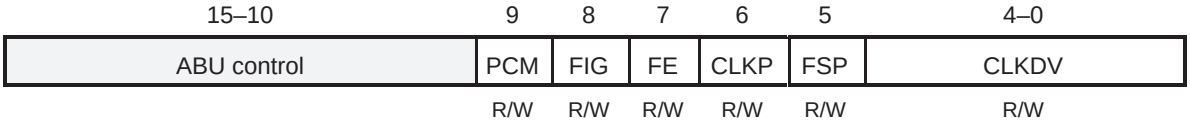
Condition	Serial Port	BSP
Initialization timing requirements.	On the serial port, the serial port may be taken out of reset at any time with respect to FSX/FSR, however, if $\overline{\text{XRST}}$/$\overline{\text{RRST}}$ go high during or after the frame sync, the frame sync may be ignored.	On the BSP, exiting serial port reset under certain conditions must precede FSX timing by one CLKOUT cycle in standard mode and by six CLKOUT cycles in autobuffering mode (see subsection 9.3.3, *System Considerations of BSP Operation*, on page 9-48).
Operates in IDLE2/3 mode.	No	Yes (see subsection 9.3.3, *System Considerations of BSP Operation*, on page 9-48).

9.3.1.2 Enhanced BSP Features

The enhanced features that the BSP offers include the capability to generate programmable rate serial port clocks, select positive or negative polarities for clock and frame sync signals, and to perform transfers of 10- and 12-bit words, in addition to the 8- and 16-bit transfers offered by the serial port. Additionally, the BSP implements the capability to specify that frame sync signals be ignored until instructed otherwise, and provides a dedicated operating mode which facilitates its use with PCM interfaces.

The BSPCE contains the control and status bits that are used in the implementation of these enhanced BSP features and the ABU. The 10 LSBs of BSPCE are dedicated to the enhanced features control, whereas the 6 MSBs are used for ABU control, which is discussed in subsection 9.3.2, *Autobuffering Unit (ABU) Operation*, on page 9-39. Figure 9–22 shows the BSPCE bit positions and Table 9–9 summarizes the function of the BSPCE bits. The value of the BSPCE upon reset is 3. This results in standard mode operation compatible with the serial port.

Figure 9–22. BSP Control Extension Register (BSPCE) Diagram — Serial Port Control Bits



Note: R = Read, W = Write

Table 9–9. BSP Control Extension Register (BSPCE) Bit Summary — Serial Port Control Bits

Bit	Name	Reset value	Function
15–10	ABU control	—	Reserved for autobuffering unit control (see subsection 9.3.2, *Autobuffering Unit (ABU) Operation*, on page 9-39).
9	PCM	0	Pulse Code Modulation Mode. This control bit puts the serial port in pulse code modulation (PCM) mode. The PCM mode only affects the transmitter. BDXR-to-BXSR transfer is not affected by the PCM bit value. PCM = 0 Pulse code modulation mode is disabled. PCM = 1 Pulse code modulation mode is enabled. In PCM mode, BDXR is transmitted only if its most significant (2^{15}) bit is set to 0. If this bit is set to 1, BDXR is not transmitted and BDX is put in high impedance during the transmission period.
8	FIG	0	Frame Ignore. This control bit operates only in transmit continuous mode with external frame and in receive continuous mode. FIG = 0 Frame sync pulses following the first frame pulse restart the transfer. FIG = 1 Frame sync pulses following the first frame pulse that initiates a transfer operation are ignored.
7	FE	0	Format Extension. The FE bit in conjunction with FO in SPC (subsection 9.2.3, *Setting the Serial Port Configuration*, on page 9-7) specifies the word length. When FO FE = 00, the format is 16-bit words; when FO FE = 01, the format is 10-bit words; when FO FE = 10, the format is 8-bit words; and when FO FE = 11, the format is 12-bit words. Note that for 8-, 10-, and 12-bit words, the received words are right justified and the sign bit is extended to form a 16-bit word. Words to transmit must be right justified. See Table 9–10 for the word length configurations.
6	CLKP	0	Clock Polarity. This control bit specifies when the data is sampled by the receiver and transmitter. CLKP = 0 Data is sampled by the receiver on BCLKR falling edge and sent by the transmitter on BCLKX rising edge. CLKP = 1 Data is sampled by the receiver on BCLKR rising edge and sent by the transmitter on BCLKX falling edge.
5	FSP	0	Frame Sync Polarity. This control bit specifies whether frame sync pulses (BFSX and BFSR) are active high or low. FSP = 0 Frame sync pulses (BFSX and BFSR) are active high. FSP = 1 Frame sync pulses (BFSX and BFSR) are active low.

Table 9–9. BSP Control Extension Register (BSPCE) Bit Summary — Serial Port Control Bits (Continued)

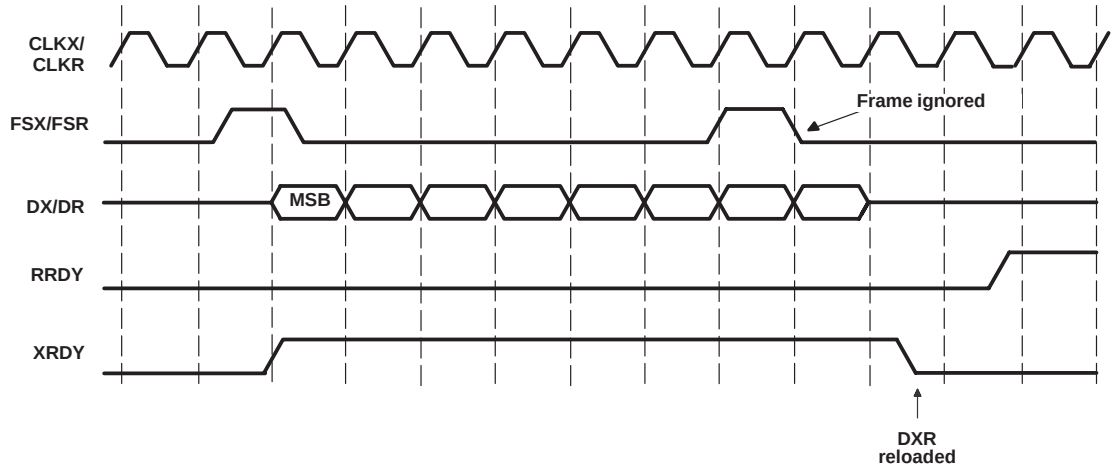
Bit	Name	Reset value	Function
4–0	CLKDV	00011	Internal Transmit Clock Division factor. When the MCM bit of BSPC is set to 1, CLKX is driven by an on-chip source having a frequency equal to 1/(CLKDV+1) of CLKOUT. CLKDV range is 0–31. When CLKDV is odd or equal to 0, the CLKX duty cycle is 50%. When CLKDV is an even value (CLKDV=2p), the CLKX high and low state durations depend on CLKP. When CLKP is 0, the high state duration is p+1 cycles and the low state duration is p cycles; when CLKP is 1, the high state duration is p cycles and the low state duration is p+1 cycles.

Table 9–10. Buffered Serial Port Word Length Configuration

FO	FE	Buffered Serial Port Word Length Configuration
0	0	16-bit words transmitted and received. (Reset values)
0	1	10-bit words transmitted and received.
1	0	8-bit words transmitted and received.
1	1	12-bit words transmitted and received.

These enhanced features allow greater flexibility in serial port interface in a variety of areas. In particular, the frame ignore feature offers a capability which allows a mechanism for effectively compressing transferred data packets if they are not transferred in 16 bit format. This feature is used with continuous receptions and continuous transmits with external frame sync. When FIG = 0, if a frame sync pulse occurs after the initial one, the transfer is restarted; when FIG = 1, this frame sync is ignored. Setting FIG to 1 allows, for example, effectively achieving continuous 16-bit transfers under circumstances where frame sync pulses occur every 8-, 10- or 12-bits. Without using FIG, each transfer of less than 16 bits requires an entire 16-bit memory word, and each 16 bits transferred as two 8-bit bytes requires two memory words and two transfer operations, rather than one of each. Using FIG, therefore, can result in a significant improvement in buffer size requirement in both autobuffered and standard mode, and a significant improvement in CPU cycle overhead required to handle serial port transfers in standard mode. Figure 9–23 shows an example with the BSP configured in 16-bit format but with a frame sync after 8 bits.

Figure 9–23. Transmit Continuous Mode with External Frame and FIG = 1 (Format Is 16 Bits)



9.3.2 Autobuffering Unit (ABU) Operation

Since ABU functionality is a superset of standard mode serial port operation, Section 9.2, *Serial Port Interface*, on page 9-3 and subsection 9.3.1, *BSP Operation in Standard Mode,* on page 9-34 should first be studied before this subsection is read. Also, note that when operating in autobuffering mode, the serial port control and status bits in BSPC and BSPCE function in the same fashion as in standard mode.

The ABU implements the capability to move data transferred on the serial port to and from internal ’54x memory independent of CPU intervention.

The ABU utilizes five memory-mapped registers: the address transmit register (AXR), the block size transmit register (BKX), the address receive register (ARR), and the block size receive register (BKR), along with the BSPCE. These registers are summarized in Table 9–11.

Table 9–11. Autobuffering Unit Registers

Address	Register	Description
†	BSPCE	16-bit BSP control extension register
†	AXR	11-bit BSP address transmit register (ABU)
†	BKX	11-bit BSP transmit buffer size register (ABU)
†	ARR	11-bit BSP address receive register (ABU)
†	BKR	11-bit BSP receive buffer size register (ABU)

† See Section 8.1, *Peripheral Memory-Mapped Registers*.

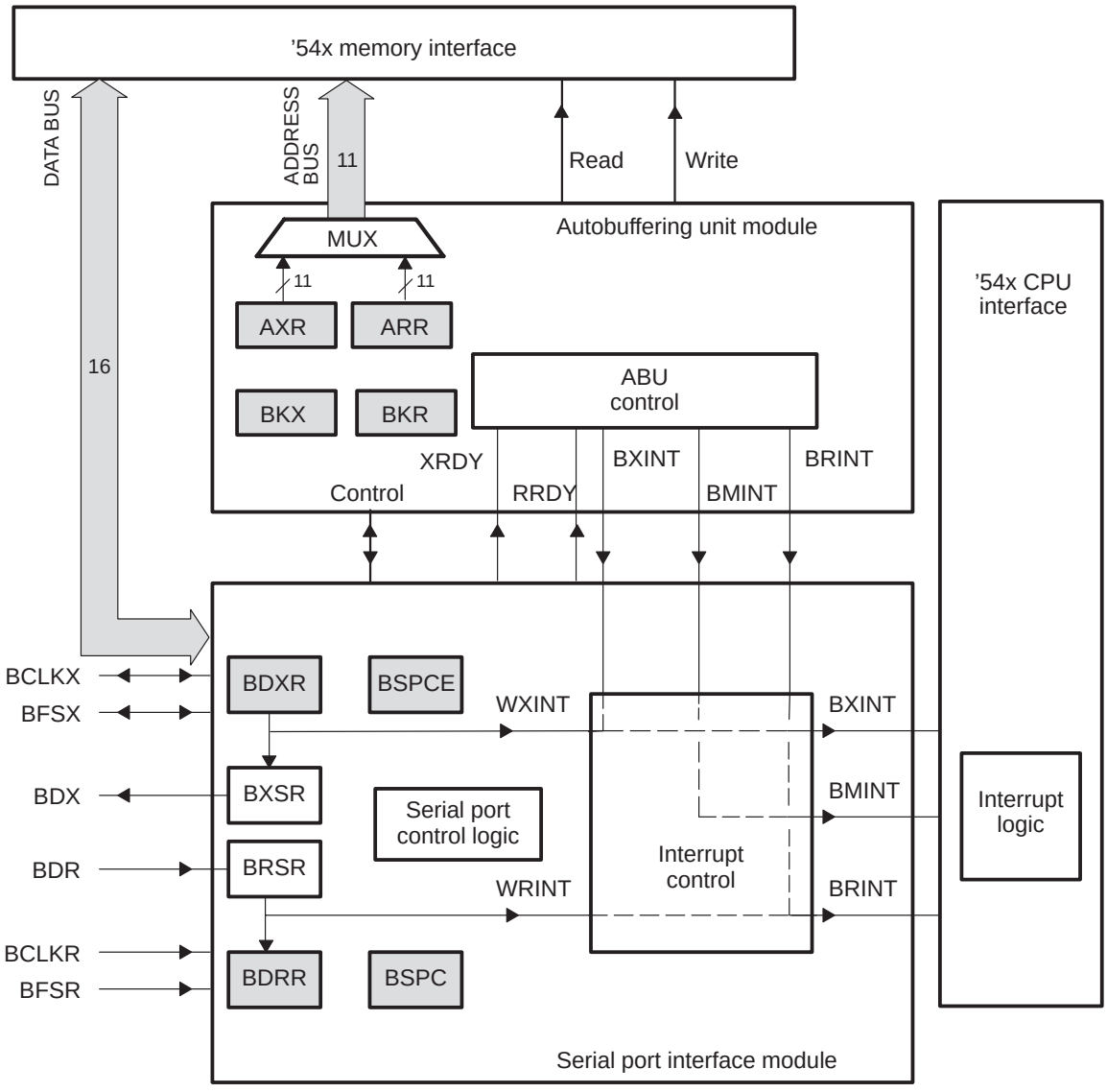
Figure 9–24 shows the block diagram of the ABU. The BSPCE contains bits which control ABU operation and will be discussed in detail later in this subsection. AXR, BKX, ARR, and BKR, along with their associated circular addressing logic, allow address generation for accessing words to be transferred between the ’54x internal memory and the BSP data transmit register (BDXR) and BSP data receive register (BDRR) in autobuffering mode. The address and block size registers as well as circular addressing are also discussed in detail later in this subsection.

Note that the 11-bit memory mapped AXR, BKX, ARR, and BKR registers are read as 16-bit words, with the five most significant bits read as zeroes and the 11-bit register contents right justified in the least significant 11 bits. If autobuffering is not used, these registers can be used for general purpose storage of 11-bit data.

The transmit and receive sections of the ABU can be enabled separately. When either section is enabled, access to its corresponding serial port data register (BDXR or BDRR) through software is limited. The BDRR can only be read, and the BDXR can only be written when the ABU is disabled. The BDRR can only be written when the BSP is in reset. The BDXR can be read any time. When either transmit or receive autobuffering is disabled, that section operates in standard mode, and its portion of the ABU is transparent.

The ABU also implements CPU interrupts when transmit and receive buffers have been halfway or entirely filled or emptied. These interrupts take the place of the transmit and receive interrupts in standard mode operation (the receive interrupt is the CPU). They are not generated in autobuffering mode. This mechanism features an autodisabling capability that can be used to automatically terminate autobuffering when either the half-of-buffer or bottom-of-buffer boundary is crossed. These features are described in detail later in this section.

Figure 9–24. ABU Block Diagram



Burst or continuous mode, as described in Section 9.2, *Serial Port Interface*, can be used in conjunction with the autobuffering capability. Note that due to the nature of autobuffering mode, however, if burst mode with internal frame sync is selected, this will effectively result in continuous transmission with FSX generated by the BSP at the start of each transmission.

The internal ’54x memory used for autobuffering consists of a 2K-word block of dual-access memory that can be configured as data, program, or both (as with other dual-access memory blocks). This memory can also be used by the CPU as general purpose storage, however, this is the only memory block in which autobuffering can occur. Since the BSP is implemented on several different TMS320 devices, the actual base address of the ABU memory may not be the same in all cases. The memory map for the particular device being used should be consulted for the actual base address of its ABU memory.

When the ABU is enabled, this 2K-word block of memory can still be accessed by the CPU within data and/or program spaces. Conflicts may therefore occur between the CPU and the ABU if the 2K-word block is accessed at the same time by both. If a conflict does occur, priority is given to the ABU, resulting in the CPU access being delayed by one cycle. Accordingly, the worst case situation is that a CPU access could be delayed one cycle each time the ABU accesses the memory block, that is, for every new word transmitted or received. Note that when on-chip program memory is secured using the ROM protection feature, the 2K-word block of ABU memory cannot be mapped to program memory. For further information regarding the ROM protection feature, see Section 3.5, *Program and Data Security*, on page 3-22.

When the ABU is enabled for both transmit and receive, if transmit and receive requests from the serial port interface occur at same time, the transmit request takes priority over the receive request. In this case, the transmit memory access occurs first, delaying the receive memory access by generating a wait state. When the transmit memory access is completed, the receive memory access takes place.

9.3.2.1 Autobuffering Control Register

The most-significant six bits in the BSPCE constitute the ABU control register (ABUC). Some of these bits are read only, while others are read/write. Figure 9–25 shows the ABUC bit positions and Table 9–12 summarizes the function of each ABUC bit in the BSPCE. The value of the BSPCE upon reset is 3.

Figure 9–25. BSP Control Extension Register (BSPCE) Diagram — ABU Control Bits

15	14	13	12	11	10	9–0
HALTR	RH	BRE	HALTX	XH	BXE	Serial Port control
R/W	R	R/W	R/W	R	R/W	

Note: R = Read, W = Write

Table 9–12. BSP Control Extension Register (BSPCE) Bit Summary — ABU Control Bits

Bit	Name	Reset value	Function
15	HALTR	0	Autobuffering Receive Halt. This control bit determines whether autobuffering receive is halted when the current half of the buffer has been received. HALTR = 0 Autobuffering continues to operate when the current half of the buffer has been received. HALTR = 1 Autobuffering is halted when the current half of the buffer has been received. When this occurs, the BRE bit is cleared to 0 and the serial port continues to operate in standard mode.
14	RH	0	Receive Buffer Half Received. This read-only bit indicates which half of the receive buffer has been filled. Reading RH when the RINT interrupt occurs (seen either as a program interrupt or by polling IFR) is a convenient way to identify which boundary has just been crossed. RH = 0 The first half of the buffer has been filled and that receptions are currently placing data in the second half of the buffer. RH = 1 The second half of the buffer has been filled and that receptions are currently placing data in the first half of the buffer.
13	BRE	0	Autobuffering Receive Enable. This control bit enables autobuffering receive. BRE = 0 Autobuffering is disabled and the serial port interface operates in standard mode. BRE = 1 Autobuffering is enabled for the receiver.
12	HALTX	0	Autobuffering Transmit Halt. This control bit determines whether autobuffering transmit is halted when the current half of the buffer has been transmitted. HALTX = 0 Autobuffering continues to operate when the current half of the buffer has been transmitted. HALTX = 1 Autobuffering is halted when the current half of the buffer has been transmitted. When this occurs, the BXE bit is cleared to 0 and the serial port continues to operate in standard mode.

Table 9–12. BSP Control Extension Register (BSPCE) Bit Summary — ABU Control Bits (Continued)

Bit	Name	Reset value	Function
11	XH	0	Transmit Buffer Half Transmitted. This read-only bit indicates which half of the transmit buffer has been transmitted. Reading XH when the XINT interrupt occurs (seen either as a program interrupt or by polling IFR) is a convenient way to identify which boundary has just been crossed. XH = 0 The first half of the buffer has been transmitted and transmissions are currently taking data from the second half of the buffer. XH = 1 The second half of the buffer has been transmitted and transmissions are currently taking data from the first half of the buffer.
10	BXE	0	Autobuffering Transmit Enable. This control bit enables the autobuffering transmit. BXE = 0 Autobuffering is disabled and the serial port operates in standard mode. BXE = 1 Autobuffering is enabled for the transmitter.
9–0	Serial Port control	—	Serial Port Interface Control bits (see subsection 9.3.1.2, *Enhanced BSP Features*, on page 9-36).

9.3.2.2 Autobuffering Process

The autobuffering process occurs between the ABU and the 2K-word block of ABU memory. Each time a serial port transfer occurs, the data involved is automatically transferred to or from a buffer in the 2K-word block of memory under control of the ABU. During serial port transfers in autobuffering mode, interrupts are not generated with each word transferred as they are in standard mode operation. This prevents the overhead of having the CPU directly involved in each serial port transfer. Interrupts are generated to the CPU only each time one of the half-buffer boundaries is crossed.

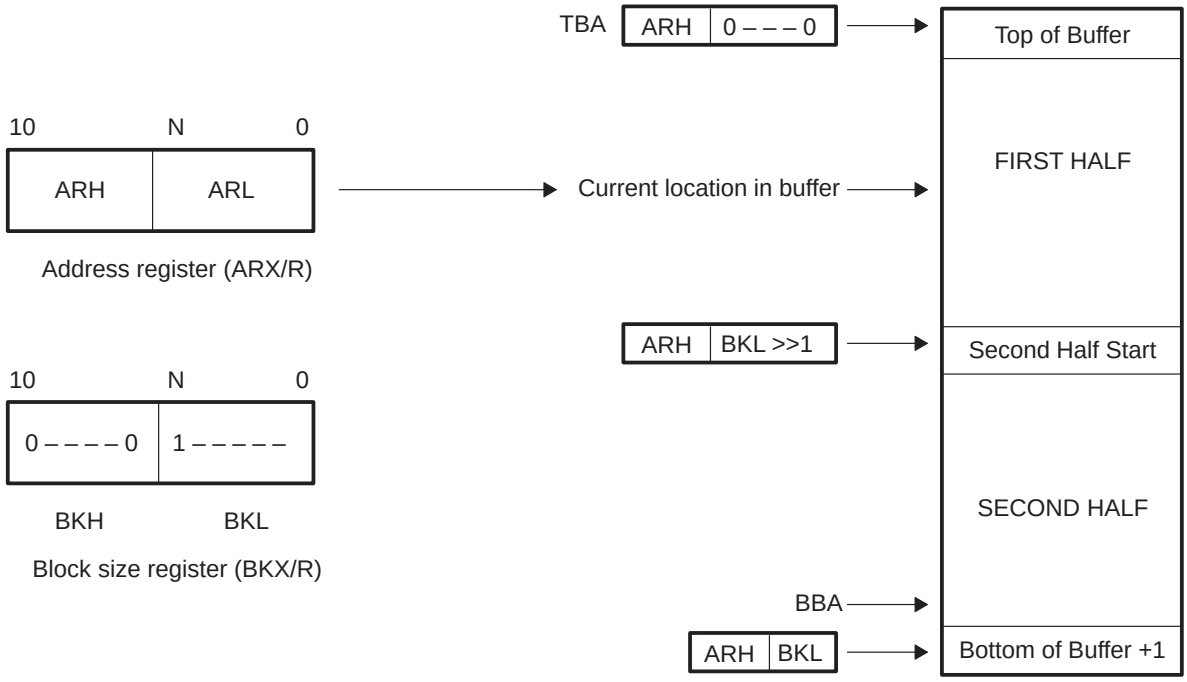
Within the 2K-word block of ABU memory, the starting address and size of the buffers allocated is programmable using the 11-bit address registers (AXR and ARR) and the 11-bit block size registers (BKX and BKR). The transmit and receive buffers can reside in independent areas, overlapping areas or the same area, which allows transmitting from a buffer while receiving into the same buffer if desired.

The autobuffering process utilizes a circular addressing mechanism to access buffers within the 2K word block of ABU memory. This mechanism operates in the same fashion for transmit and receive. For each direction (transmit or receive), two registers specify the buffer size and the current address in the buffer. These registers are the block size and address register for transmit and receive (BKX, BKR, ARX, ARR, respectively). Each of the block size and address register pairs fully specify the top and bottom of buffer addresses for transmit and receive. Note that this circular addressing mechanism only effects accesses into the 2K-word block by the ABU. Accesses to this memory by the CPU are performed strictly according to the addressing mode(s) selected in the assembly language instructions which perform the memory access.

The circular addressing mechanism automatically recirculates ABU memory accesses through the specified buffer, returning to the top of the buffer each time the bottom of the buffer is reached. The circular addressing mechanism is initialized by loading BKX/R with the exact size of the desired buffer (as opposed to size–1) and ARX/R with a value which contains both the base address of the buffer within the 2K word block and the initial starting address within this buffer (this is explained in detail below). Often the initial starting address within the buffer is 0, indicating the start of the buffer (the top-of-buffer address), but the initial starting address may be any point within the defined buffer range.

Once initialized, BKX/R can be considered to consist of two parts; the most significant or higher part (BKH), which corresponds to the all of the most significant 0 bits of BKX/R, and the lower part (BKL), which is the remaining bits, of which the most significant bit is a 1 and whose bit position is designated bit position N. The N bit position also defines the two parts (ARH and ARL) of the address register. The top of buffer address (TBA) is defined by the concatenation of ARH with N+1 least significant 0 bits. The bottom of buffer address (BBA) is defined by the concatenation of ARH and BKL–1, and the current address within the buffer is specified by the complete contents of ARX/R. A circular buffer of size BKX/R must therefore start on an N-bit boundary (the N least significant bits of the address register are 0) where N is smallest integer that satisfies 2^N > BKX/R, or at the lowest address within the 2K memory block. The buffer consists of two halves: the address range for the first half is TBA $\geq$ (BKL/2) – 1 and for the second half BKL/2 $\geq$ (BKL – 1). Figure 9–26 illustrates all of the relationships between the defined buffer and the BKX/R and ARX/R registers, the bottom of circular buffer address (BBA), and the top of circular buffer address (TBA).

Figure 9–26. Circular Addressing Registers

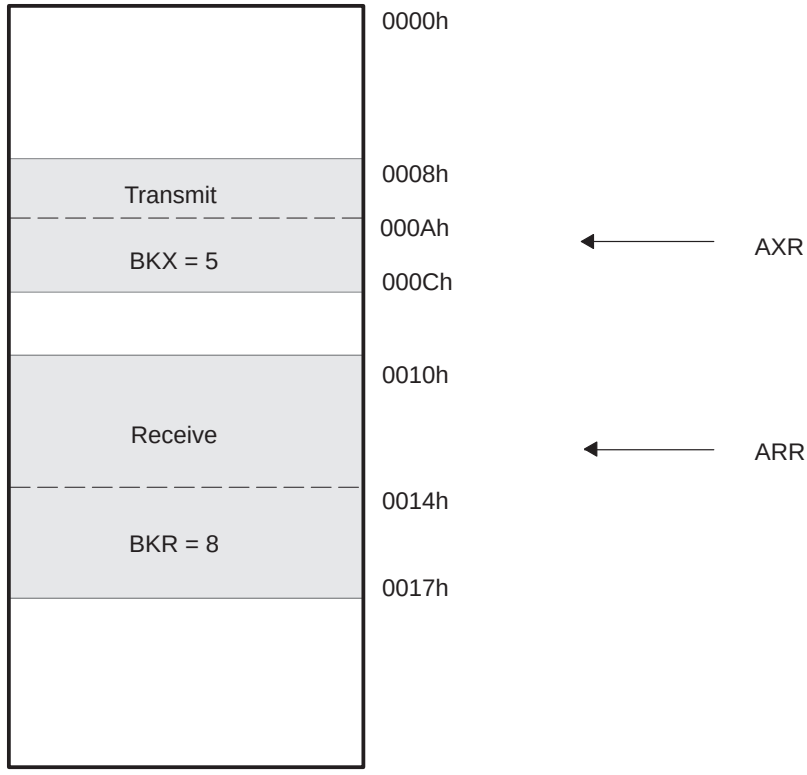


The minimum block size for an ABU buffer is two; the maximum block size is 2047, and any buffer of 2047 to 1024 words must start at a relative address of 0x0000 with respect to the base address of the 2K block of ABU memory. If either of the address registers (AXR or ARR) is loaded with a value specifying a location that is outside the range of the currently allocated buffer size as defined by BKX/R, improper operation may result. Subsequent memory accesses will be performed starting at the location specified, despite the fact that they will be to locations which are outside the range of the desired buffer, and the ARX/R will be incremented with each access until its contents reach the next permitted buffer start address. Any further accesses are then performed using the correct circular buffering algorithm with the new ARX/R contents as the updated buffer start address. It should be noted that any accesses performed with improperly loaded ARX/Rs may therefore unexpectedly corrupt some memory locations.

The following example illustrates some of these functional aspects of the auto-buffering process. Consider a transmit buffer of size 5 (BKX = 5) and a receive buffer of size 8 (BKR = 8) as shown in Figure 9–27. The transmit buffer may start at any relative address that is a multiple of 8 (address 0x0000, 0x0008, 0x0010, 0x0018, ···, 0x07F8), and the receive buffer may start at any relative address that is a multiple of 16 (0x0000, 0x0010, 0x0020, ···, 0x07F0). In this

example, the transmit buffer starts at relative address 0x0008 and the receive buffer starts at relative address 0x0010. AXR may therefore contain any value in the range 0x0008–0x000C and ARR may contain any value in the range 0x0010–0x0017. If AXR in this example had been loaded with the value 0x000D (not acceptable in a modulo 5 buffer), memory accesses would be performed and AXR incremented until it reaches address 0x0010 which is an acceptable starting address for a modulo 5 buffer. Note, however, that if this had occurred, AXR would then specify a transmit buffer starting at the same base address as the receive buffer, which may cause improper buffer operation.

Figure 9–27. Transmit Buffer and Receive Buffer Mapping Example



The autobuffering process is activated upon request from serial port interface when XRDY or RRDY goes high, indicating that a word has been received. The required memory access is then performed, following which an interrupt is generated if half of the defined buffer (first or second) has been processed. The RH and XH flags in BSPCE indicate which half has been processed when the interrupt occurs.

When autodisabling is selected (HALTX or HALTR bit is set), then when the next half (first or second) buffer boundary is encountered, the autobuffering enable bit in the BSPCE (BXE or BRE) is cleared so that autobuffering is disabled and does not generate any further requests. When transmit autobuffering is halted, transmission of the current XSR contents and the last value loaded in DXR are completed, since these transfers have already been initiated. Therefore, when using the HALTX function, some delay will normally occur between crossing a buffer boundary and transmission actually stopping. If it is necessary to identify when transmission has actually ended, software should poll for the condition of XRDY = 1 and $\overline{\text{XSREMPTY}}$ = 0, which occurs after last bit has been transmitted.

In the receiver, when using HALTR, since autobuffering is stopped when the most recent buffer boundary is crossed, future receptions may be lost, unless software begins servicing receive interrupts at this point, since BDRR is no longer being read and transferred to memory automatically by the ABU. For explanation of how the serial port operates in standard mode when DRR is not being read, refer to subsection 9.2.6, *Serial Port Interface Exception Conditions*, on page 9-26.

The sequence of events involved in the autobuffering process is summarized as follows:

1) The ABU performs the memory access to the buffer.

2) The appropriate address register is incremented unless the bottom of buffer has been reached, in which case the address register is modified to point to the top of buffer address.

3) Generate an BXINT or BRINT and update XH/RH if the half buffer or bottom of buffer boundary has been crossed.

4) Autodisable the ABU if this function has been selected and if the half buffer or bottom of buffer boundary has been crossed.

9.3.3 System Considerations for BSP Operation

This subsection discusses several system-level considerations of BSP operation. These considerations include initialization timing issues, software initialization of the ABU, and power down mode operation.

9.3.3.1 Timing of Serial Port Initialization

The ’54x device utilizes a fully static design, and accordingly, in both the serial port and the BSP, serial port clocks need not be running between transfers or prior to initialization. Therefore, proper operation can still result if FSX/FSR occurs simultaneously with CLKX/CLKR starting. Regardless of whether serial port clocks have been running previously, however, the timing of serial port initialization, and most importantly, when the port is taken out of reset, can be critical for proper serial port operation. The most significant consideration of this is when the port is taken out of reset with respect to when the first frame sync pulse occurs.

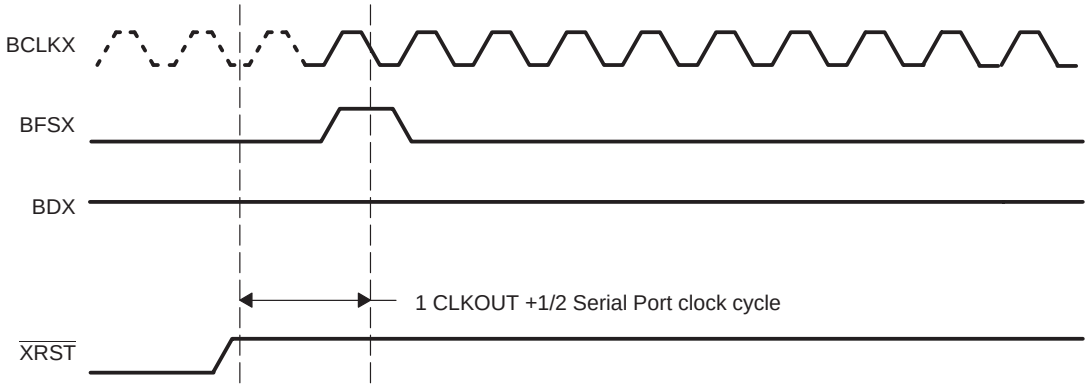
Initialization timing requirements differ on the serial port and the BSP. On the serial port, the serial port may be taken out of reset at any time with respect to FSX/FSR, however, if $\overline{\text{XRST}}$/$\overline{\text{RRST}}$ go high during or after the frame sync, the frame sync may be ignored. In standard mode operation on the BSP for receive, and for transmit with external frame sync (TXM = 0), the BSP must be taken out of reset at least two full CLKOUT cycles plus 1/2 serial port clock cycle prior to the edge of the clock which detects the active frame sync pulse (whether the clock has been running previously or not) for proper operation. See Figure 9–28.

Transmit operations with internal clock and frame sync are not subject to this requirement since frame sync is internally generated automatically (after $\overline{\text{XRST}}$ is cleared (set to 1)) when BDXR is loaded.

Note, however, that if external serial port clock is used with internal frame sync, frame sync generation may be delayed depending on the timing of clearing $\overline{\text{XRST}}$ with respect to the clock.

Figure 9–28 illustrates the standard mode BSP initialization timing requirements for the transmitter. The figure shows standard mode operation with external frame (TXM = 0) and clock (MCM = 0), active high frame sync (FSP = 0), and data sampled on rising edge (CLKP = 0). In this example, if the BFSX pulse occurs during the first two BCLKXs after the transmit section is taken out of reset, the transmit frame is ignored and BDX is placed in the high impedance state.

Figure 9–28. Standard Mode BSP Initialization Timing



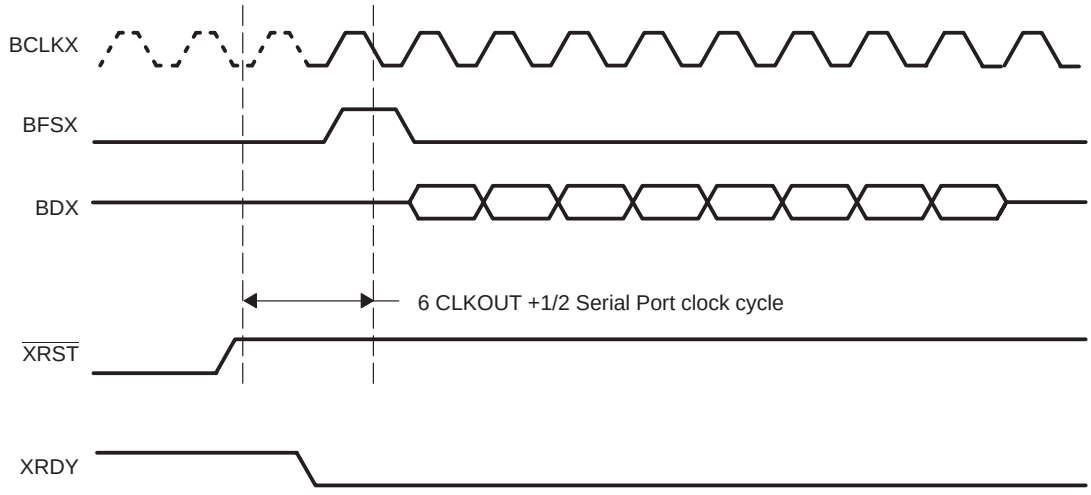
In autobuffering mode, for receive, and transmit with external frame sync (TXM = 1), the BSP must be taken out of reset at least six CLKOUT cycles plus 1/2 serial port clock cycle prior to the edge of the clock which detects the active frame sync pulse (whether the clock has been running previously or not) for proper operation. This is due to the time delay for the ABU logic to be activated. See Figure 9–29.

Transmit operations with internal clock and frame sync are not subject to this requirement since frame sync is internally generated automatically after $\overline{\text{XRST}}$ is cleared.

Note, however, that if external serial port clock is used with internal frame sync, and if the clock is not running when $\overline{\text{XRST}}$ is cleared, frame sync generation may be delayed depending on the timing of clearing $\overline{\text{XRST}}$ with respect to the clock.

Figure 9–29 illustrates autobuffering mode initialization timing requirements for the transmitter with external clock and frame sync. The figure shows standard mode operation with external frame (TXM = 0) and clock (MCM = 0), active high frame sync (FSP = 0), and data sampled on rising edge (CLKP = 0).

Figure 9–29. Autobuffering Mode Initialization Timing



9.3.3.2 Initialization Examples

In order to start or restart BSP operation in standard mode, the same steps are performed in software as with initializing the serial port (see Section 9.2, *Serial Port Interface*, on page 9-3), in addition to which, the BSPCE must be initialized to configure any of the enhanced features desired. To start or restart the BSP in autobuffering mode, a similar set of steps must also be performed, in addition to which, the autobuffering registers must be initialized.

As an illustration of the proper operation of a buffered serial port, Example 9–3 and Example 9–4 define a sequence of actions. This illustration is based on the use of interrupts to handle the normal I/O between the serial port and CPU. The ’545 peripheral configuration has been used as a reference for these examples. The examples illustrate initializing the buffered serial port for autobuffering mode operation. In both cases, assume that transmit and receive interrupts are used to service the ABU interrupts, however, polling of the interrupt flag register (IFR) could also be used. Both the transmit and receive sections can be initialized at the same time or separately depending upon system requirements.

Example 9–3 initializes the serial port for transmit operations only, with burst mode, external frame sync, and external clock selected. The selected data format is 16 bits, with frame sync and clock polarities selected to be high true. Transmit autobuffering is enabled by setting the BXE bit in the ABUC section of BSPCE, and HALTX has been set to 1, which causes transmission to halt when half of the defined buffer is transmitted.

Example 9–4 initializes the serial port for receive operations only, with continuous mode selected. Frame sync and clock polarities are selected to be

low true, data format is 16 bits, and frame ignore is selected so that two received data bytes are packed into a single received word to minimize memory requirements. Receive autobuffering is enabled by setting the BRE bit in the ABUC section of BSPCE.

In Example 9–3 and Example 9–4, the transmit and receive interrupts used are those that the BSP occupies on the ’542, ’543, ’545, ’546, ’548, and ’549, the devices that include the BSP. However, on other devices that use the BSP, different interrupts may be used; and therefore, you should consult the appropriate device documentation.

Example 9–3. BSP Transmit Initialization Routine

Action	Description
1) Reset and initialize the serial port by writing 0008h to BSPC.	This places both the transmit and receive portions of the serial port in reset and sets up the serial port to operate with externally generated FSX and CLKX signals and FSX required for transmit/receive of each 16-bit word.
2) Clear any pending serial port interrupts by writing 0020h to IFR.	Eliminate any interrupts that may have occurred before initialization.
3) Enable the serial port interrupts by ORing 0020h with IMR.	Enable transmit interrupts.
4) Enable interrupts globally (if necessary) by clearing the INTM bit in ST1.	Interrupts must be globally enabled for the CPU to respond.
5) Initialize the ABU transmit by writing 1400h to BSPCE.	This causes the BSP to stop transmitting at the end of the buffer until another FSX is received.
6) Write the buffer start address to AXR.	Identify the first buffer address to the ABU.
7) Write the buffer size to BKX.	Identify the buffer size to the ABU.
8) Start the serial port by writing 0048h to BSPC.	This takes the transmit portion of the serial port out of reset and starts operations with the conditions defined in steps 1 and 5.

Example 9–4. BSP Receive Initialization Routine

	Action	Description
1)	Reset and initialize the serial port by writing 0000h to BSPC.	This places the receive portion of the serial port in reset and sets up the serial port to operate in continuous receive mode with 16-bit words.
2)	Clear any pending serial port interrupts by writing 0010h to IFR.	Eliminate any interrupts that may have occurred before initialization.
3)	Enable the serial port interrupts by ORing 0010h with IMR.	Enable receive interrupts.
4)	Enable interrupts globally (if necessary) by clearing the INTM bit in ST1.	Interrupts must be globally enabled for the CPU to respond.
5)	Initialize the ABU receive by writing 2160h to BSPCE.	This causes the BSP to receive continuously and not restart if a new FSR is received.
6)	Write the buffer start address to ARR.	Identify the first buffer address to the ABU.
7)	Write the buffer size to BKR.	Identify the buffer size to the ABU.
8)	Start the serial port by writing 0080h to BSPC.	This takes the receive portion of the serial port out of reset and starts operations with the conditions defined in steps 1 and 5.

9.3.4 Buffer Misalignment Interrupt (BMINT) – ’549 only

BMINT is generated when a frame sync occurs and the ABU transmit or receive buffer pointer is not at the top of the the buffer address. This is useful for detecting several potential error conditions on the serial interface, including extraneous and missed clocks and frame sync pulses. A BMINT interrupt, therefore, indicates that one or more words may have been lost on the serial interface.

BMINT is useful for detecting buffer misalignment only when the buffer pointer(s) are initially loaded with the top of the buffer address and a frame of data contains the same number of words as the buffer length. These are the only conditions under which a frame sync occurring at a buffer address other than the top of the buffer constitute an error condition. In cases where these conditions are met, a frame sync always occurs when the buffer pointer is at the top of the buffer address (if the interface is functioning properly).

If BMINT is enabled under conditions other than those described, interrupts can be generated under circumstances other than actual buffer misalignment. In these cases, BMINT should generally be masked in the IMR register so that the processor ignores this interrupt.

BMINT is available when the device is operating in the auto-buffering mode with continuous transfers, the FIG bit cleared to 0, and with external serial clocks or frames.

9.3.5 BSP Operation in Power-Down Mode

The ’54x offers several power down modes which allow part or all of the device to enter a dormant state and dissipate considerably less power than when running normally. Power down mode may be invoked in several ways, including either executing the IDLE instruction or driving the $\overline{\text{HOLD}}$ input low with the HM status bit set to 1. The BSP, like other peripherals (timer, standard serial port), can take the CPU out of IDLE using the transmit interrupt (BXINT) or receive interrupt (BRINT).

When in IDLE or HOLD mode, the BSP continues to operate, as is the case with the serial port. When in IDLE2/3, unlike the serial port and other on-chip peripherals which are stopped with this power-down mode, the BSP can still be operated.

In standard mode, if the BSP is using external clock and frame sync while the device is in IDLE2/3, the port will continue to operate, and a transmit interrupt (BXINT) or receive interrupt (BRINT) will take the device out of IDLE2/3 mode if INTM = 0 before the device executes the IDLE 2 or IDLE 3 instruction. With internal clock and/or frame sync, the BSP remains in IDLE2/3 until the CPU resumes operation.

In autobuffering mode, if the BSP is using external clock and frame sync while the device is in IDLE2/3, a transmit/receive event will cause the internal BSP clock to be turned on for the cycles required to perform the DXR (or DRR) to memory transfer. The internal BSP clock is then turned off automatically as soon as the transfer is complete so the device will remain in IDLE2/3. The device is awakened from IDLE2/3 by the ABU transmit interrupt (BXINT) or receive interrupt (BRINT) when the transmit/receive buffer has been halfway or entirely emptied or filled if INTM = 0 before the device executes the IDLE 2 or IDLE 3 instruction.

9.4 Time-Division Multiplexed (TDM) Serial Port Interface

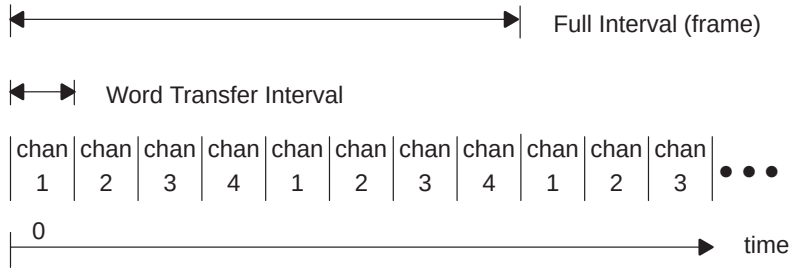
The time-division multiplexed (TDM) serial port allows the ’54x device to communicate serially with up to seven other devices. The TDM port, therefore, provides a simple and efficient interface for multiprocessing applications.

The TDM serial port is a superset of the serial port described in Section 9.2 on page 9-3. By means of the TDM bit in the TDM serial port control register (TSPC), the port can be configured in multiprocessing mode (TDM = 1) or stand-alone mode (TDM = 0). When in stand-alone mode, the port operates as described in Section 9.2. When in multiprocessing mode, the port operates as described in this section. The port can be shut down for low power consumption via the $\overline{\text{XRST}}$ and $\overline{\text{RRST}}$ bits, as described in Section 9.2.

9.4.1 Basic Time-Division Multiplexed Operation

Time-division multiplexing is the division of time intervals into a number of subintervals, with each subinterval representing a communications channel according to a prespecified arrangement. Figure 9–30 shows a 4-channel TDM scheme. Note that the first time slot is labeled chan 1 (channel 1), the next chan 2 (channel 2), etc. Channel 1 is active during the first communications period and during every fourth period thereafter. The remaining three channels are interleaved in time with channel 1.

Figure 9–30. Time-Division Multiplexing



The ’54x TDM port uses eight TDM channels. Which device is to transmit and which device or devices is/are to receive for each channel may be independently specified. This results in a high degree of flexibility in interprocessor communications.

9.4.2 TDM Serial Port Interface Registers

The TDM serial port operates through six memory-mapped registers and two other register (TRSR and TXSR) that are not directly accessible to the program, but are used in the implementation of the double-buffering capability. These eight registers are listed in Table 9–13.

Table 9–13. TDM Serial Port Registers

Address	Register	Description
†	TRCV	TDM data receive register
†	TDXR	TDM data transmit register
†	TSPC	TDM serial port control register
†	TCSR	TDM channel select register
†	TRTA	TDM receive/transmit address register
†	TRAD	TDM receive address register
—	TRSR	TDM data receive shift register
—	TXSR	TDM data transmit shift register

† See Section 8.1, *Peripheral Memory-Mapped Registers*.

- TDM data receive register (TRCV). The 16-bit TDM data receive register (TRCV) holds the incoming TDM serial data. The TRCV has the same function as the DRR, described on page 9-4.
- TDM data transmit register (TDXR). The 16-bit TDM data transmit register (TDXR) holds the outgoing TDM serial data. The TDXR has the same function as the DXR, described on page 9-4.
- TDM serial port control register (TSPC). The 16-bit TDM serial port control register (TSPC) contains the mode control and status bits of the TDM serial port interface. The TSPC is identical to the SPC (Figure 9–3) except that bit 0 serves as the TDM mode enable control bit in the TSPC. The TDM bit configures the port in TDM mode (TDM = 1) or stand-alone mode (TDM = 0). In stand-alone mode, the port operates as a standard serial port as described on page 9-3.
- TDM channel select register (TCSR). The 16-bit TDM channel select register (TCSR) specifies in which time slot(s) each ’54x device is to transmit.
- TDM receive/transmit address register (TRTA). The 16-bit TDM receive/transmit address register (TRTA) specifies in the eight LSBs (RA0–RA7) the receive address of the ’54x device and in the eight MSBs (TA0–TA7) the transmit address of the ’54x device.
- TDM receive address register (TRAD). The 16-bit TDM receive address register (TRAD) contains various information regarding the status of the TDM address line (TADD).

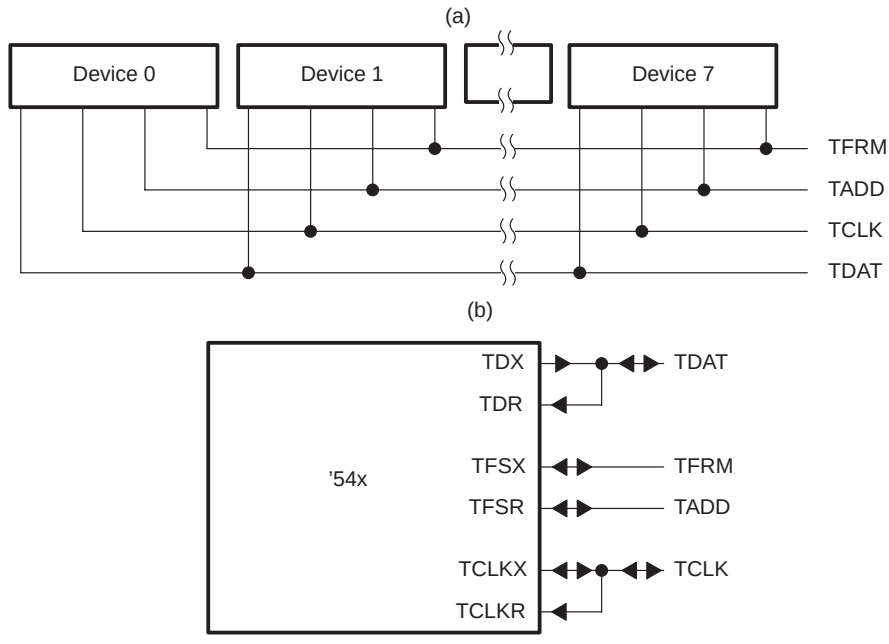
- TDM data receive shift register (TRSR). The 16-bit TDM data receive shift register (TRSR) controls the storing of the data, from the input pin, to the TRCV. The TRSR has the same function as the RSR, described on page 9-4.

- TDM data transmit shift register (TXSR). The 16-bit TDM data transmit shift register (TXSR) controls the transfer of the outgoing data from the TDXR and holds the data to be transmitted on the data-transmit (TDX) pin. The TXSR has the same function as the XSR, described on page 9-4.

9.4.3 TDM Serial Port Interface Operation

Figure 9–31(a) shows the '54x TDM port architecture. Up to eight devices can be placed on the four-wire serial bus. This four-wire bus consists of a conventional serial port's bus of clock, frame, and data (TCLK, TFRM, and TDAT) wires plus an additional wire (TADD) that carries the device addressing information. Note that the TDAT and TADD signals are bidirectional signals and are often driven by different devices on the bus during different time slots within a given frame of operation.

Figure 9–31. TDM 4-Wire Bus



The TADD line, which is driven by a particular device for a particular time slot, determines which device(s) in the TDM configuration should execute a valid TDM receive during that time slot. This is similar to a valid serial port read operation, as described in Section 9.2, *Serial Port Interface*, on page 9-3 except that some corresponding TDM registers are named differently. The TDM receive register is TRCV, and the TDM receive shift register is TRSR. Data is transmitted on the bidirectional TDAT line.

Note that in Figure 9–31(b) the device TDX and TDR pins are tied together externally to form the TDAT line. Also, note that only one device can drive the data and address line (TDAT and TADD) in a particular slot. All other devices' TDAT and TADD outputs should be in the high-impedance state during that slot, which is accomplished through proper programming of the TDM port control registers (this is described in detail later in this section). Meanwhile, in that particular slot, all the devices (including the one driving that slot) sample the TDAT and TADD lines to determine if the current transmission represents valid data to be read by any one of the devices on the bus (this is also discussed in detail later in this section). When a device recognizes an address to which it is supposed to respond, a valid TDM read then occurs, the value is transferred from TRSR to TRCV. A receive interrupt (TRINT) is generated, which indicates that TRCV has valid receive data and can be read.

All TDM port operations are synchronized by the TCLK and TFRM signals. Each of them are generated by only one device (typically the same device), referred to as the TCLK and TFRM source(s). The word master is not used here because it implies that one device controls the other, which is not the case, and TCSR must be set to prevent slot contention. Consequently, the remaining devices in the TDM configuration use these signals as inputs. Figure 9–31(b) shows that TCLKX and TCLKR are externally tied together to form the TCLK line. Also, TFRM and TADD originate from the TFSX and TFSR pins respectively. This is done to make the TDM serial port also easy to use in standard mode.

TDM port operation is controlled by six memory-mapped registers. The layout of these registers is shown in Figure 9–32. The TRCV and TDXR registers have the same functions as the DRR and DXR registers respectively, described in Section 9.2, *Serial Port Interface*. The TSPC is identical to the SPC except that bit 0 serves as the TDM mode enable control bit in the TSPC. This bit configures the port in TDM mode (TDM = 1) or stand-alone mode (TDM = 0). In stand-alone mode, the port operates as a standard serial port as described in Section 9.2. Refer to subsection 9.4.6, *Examples of TDM Serial Port Interface Operation*, on page 9-63 for additional information about the function of the bits in these registers.

Figure 9–32. TDM Serial Port Registers Diagram

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
TRCV	Receive Data															
TDXR	Transmit Data															
TSPC	Free	Soft	X	X	XRDY	RRDY	IN1	IN0	$\overline{\text{RRST}}$	$\overline{\text{XRST}}$	TXM	MCM	X	0	0	TDM
TCSR	X	X	X	X	X	X	X	X	CH7	CH6	CH5	CH4	CH3	CH2	CH1	CH0
TRTA	TA7	TA6	TA5	TA4	TA3	TA2	TA1	TA0	RA7	RA6	RA5	RA4	RA3	RA2	RA1	RA0
TRAD	X	X	X2	X1	X0	S2	S1	S0	A7	A6	A5	A4	A3	A2	A1	A0

Note: X=Don't care.

When TDM mode is selected, the DLB and FO bits in the TSPC are hard-configured to 0, resulting in no access to the digital loopback mode and in a fixed word length of 16 bits (a different type of loopback is discussed in the example in subsection 9.4.6 on page 9-63). Also, the value of FSM does not affect the port when TDM = 1, and the states of the underflow and overrun flags are indeterminate (subsection 9.4.5, *TDM Serial Port Interface Exception Conditions*, on page 9-63 explains how exceptions are handled in TDM mode). If TDM = 1, changes made to the contents of the TSPC become effective upon completion of channel 7 of the current frame. Thus the TSPC value cannot be changed for the current frame; any changes made will take effect in the next frame.

The source device for the TCLK and TFRM timing signals is set by the MCM and TXM bits, respectively. The TCLK source device is identified by setting the MCM bit of its TSPC to 1. Typically, this device is the same one that supplies the TDM port clock signal TCLK. The TCLKX pin is configured as an input if MCM = 0 and an output if MCM = 1. In the latter case (internal '54x clock), the device whose MCM = 1 supplies the clock (TCLK frequency = one fourth of CLKOUT frequency) for all devices on the TDM bus. The clock can be supplied by an external source if MCM = 0 for all devices. TFRM can also be supplied externally if TXM = 0. An external TFRM, however, must meet TDM receive timing specifications with respect to TCLK for proper operation. No more than one device should have MCM or TXM set to 1 at any given time. The specification of which device is to supply clock and framing signals is typically made only once, during system initialization.

The TDM channel select register (TCSR) of a given device specifies in which time slot(s) that device is to transmit. A 1 in any one or more of bits 0–7 of the TCSR sets the transmitter active during the corresponding time slot. Again, a key system-level constraint is that no more than one device can transmit

during the same time slot; devices do ***not*** check for bus contention, and slots must be consistently assigned. As in TSPC operation, a write to TCSR during a particular frame is valid only during the next frame. However, a given device can transmit in more than one slot. This is discussed in more detail in subsection 9.4.4, *TDM Mode Transmit and Receive Operations*, on page 9-61 with an emphasis on the utilization of TRTA, TDXR, and TCSR in this respect.

The TDM receive/transmit address register (TRTA) of a given device specifies two key pieces of information. The lower half specifies the receive address of the device, while the upper half of TRTA specifies the transmit address. The receive address (RA7–RA0, refer to Figure 9–32) is the 8-bit value that a device compares to the 8-bit value it samples on the TADD line in a particular slot to determine whether it should execute a valid TDM receive. The receive address, therefore, establishes the slots in which that device may receive, dependent on the addresses present in those slots, as specified by the transmitting devices. This process occurs on each device during every slot.

The transmit address (TA7–TA0, refer to Figure 9–32) is the address that the device drives on the TADD line during a transmit operation on an assigned slot. The transmit address establishes which receiving devices may execute a valid TDM receive on the driven data.

Only one device at a time can drive a transmit address on TADD. Each processor bit-wise-logically-ANDs the value it samples on the TADD line with its receive address (RA7–RA0). If this operation results in a nonzero value, then a valid TDM receive is executed on the processor(s) whose receive addresses match the transmitted address. Thus, for one device to transmit to another, there must be at least one bit in the upper half of the transmitting device's TRTA (the transmit address) with a value of 1 that matches one bit with a value of 1 in the lower half of TRTA (the receive address) of the receiving device. This method of configuration of TRTA allows one device to transmit to one or more devices, and for any one device to receive from one or more than one transmitter. This can also allow the transmitting device to control which devices receive, without the receive address on any of the devices having to be changed.

The TDM receive address register (TRAD) contains various information regarding the status of the TADD line which can be polled to verify the previous values of this signal and to verify the relationship between instruction cycles and TDM port timing.

Bits 13–11 (X2–X0) contain the current slot number value, regardless of whether a valid data receive was executed in that slot or not. This value is latched at the beginning of the slot and retained only until the end of the slot.

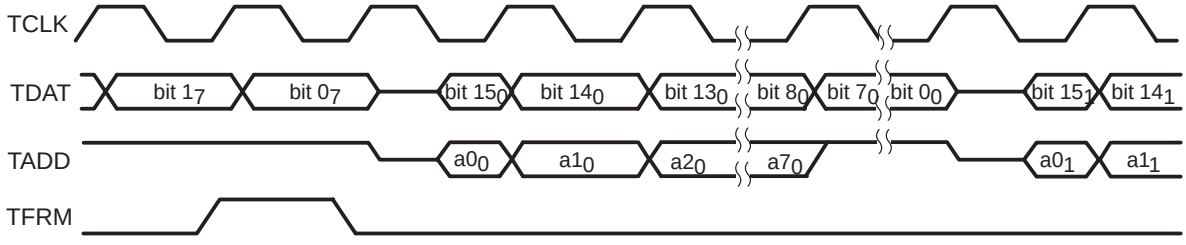
Bits 10–8 (S02–S0) hold the number of the last slot plus one (modulo eight) in which data was received (that is, if the last valid data read occurred in slot 5 in the previous frame, these bits would contain the number six). This value is latched during the TDM receive interrupt (TRINT) at the end of the slot in which the last valid data receive occurred, and maintained until the end of the next slot in which a valid receive occurs.

Bits 7–0 (A7–A0) hold the last address sampled on the TADD line, regardless of whether a valid data receive was executed or not. This value is latched halfway through each slot (so the value on the TADD may be shifted in) and maintained until halfway through the next slot, whether a valid receive is executed or not.

9.4.4 TDM Mode Transmit and Receive Operations

Figure 9–33 shows the timing for TDM port transfers. The TCLK and TFRM signals are generated by the timing source device. The TCLK frequency is one fourth the frequency of CLKOUT if generated by a ’54x device. The TFRM pulse occurs every 128 TCLK cycles and is timed to coincide with bit 0 of slot 7, which is the last bit of the previous frame. The relationship of TFRM and TCLK allows 16 data bits for each of eight time slots to be driven on the TDAT line, which also permits the processor to execute a maximum of 64 instructions during each slot, assuming that a ’54x internal clock is used. Beginning with slot 0 and with the MSB first, the transmitter drives 16 data bits for each slot, with each bit having a duration of one TCLK cycle, with the exception of the first bit of each slot, which lasts only half of one bit time. Note that data is both clocked onto the TDAT line by the transmitting device and sampled from the TDAT line by receiving devices on the rising edge of TCLK (see the data sheet for detailed TDM interface timings).

Figure 9–33. Serial Port Timing (TDM Mode)



Simultaneous with data transfer, the transmitting device also drives the TADD line with the transmit address for each slot. This information, unlike that on TDAT, is only one byte long and is transmitted with the LSB first for the first half of the slot. During the second half of the slot (that is, the last eight TCLK periods) the TADD line is driven high. The TDM receive logic samples the TADD line only for the first eight TCLK periods, ignoring it during the second half of the slot. Therefore, the transmitting device (if not a '54x) could drive TADD high or low during that time period. Note that, like TDAT, the first TADD bit transmitted lasts for only one half of one TCLK cycle.

If no device on the TDM bus is configured to transmit in a slot (that is, none of the devices has a 1 for the corresponding slot in their TCSR), that slot is considered empty. In an empty slot, both TADD and TDAT are high impedance. This condition has the potential for spurious receives, however, because TDAT and TADD are always sampled, and a device performs a valid TDM reception if its receive address matches the address on the TADD line. To avoid spurious reads, a 1-kilohm pull-down resistor *must* be tied to the TADD line. This causes the TADD line to read low on empty slots. Otherwise, any noise on the TADD line that happens to match a particular receive address would result in a spurious read. If power dissipation is a concern and the resistor is not desired, then an arbitrary processor with transmit address equal to 0h can drive empty slots by writing to TDXR in those slots. Slot manipulation is explained later in this subsection. The 1-kilohm resistor is not required on the TDAT line.

An empty TDM slot can result in the following cases: the first obvious case, as mentioned above, occurs when no device has its TCSR configured to transmit in that slot. A second more subtle case occurs when TDXR has not been loaded before a transmit slot in a particular frame. This may also happen when the TCSR contents are changed, since the actual TCSR contents are not updated until the next TFRM pulse occurs. Therefore, any subsequent change takes effect only in the next frame. The same is true for the receive address (the lower half of TRTA). The transmit address (upper half of TRTA), however, and TDXR, clearly, may be changed within the current frame for a particular slot, assuming that the slot has not yet been reached when the instruction to load the TRTA or TDXR is executed. Note that it is not necessary to load the transmit address each time TDXR is loaded; when a TDXR load occurs and a transmission begins, the current transmit address in TRTA is transmitted on TADD.

The current slot number may be obtained by reading the X2–X0 bits in TRAD. This affords the flexibility of reconfiguring the TDM port on a slot-by-slot basis, and even slot sharing if desired. The key to utilizing this capability is to understand the timing relationship between the instructions being executed and the frame/slots of the TDM port. If the TDM port is to be manipulated on a slot-by-

slot basis, changes must be made to appropriate registers quickly enough for the desired effect to take place at the desired time. It is also important to take into account that the TCSR and the receive address (lower half of TRTA) take effect only at the start of a new frame, while the transmit address (upper half of TRTA) and TDXR (transmit data) can take effect at the start of a new slot, as mentioned previously.

Note that if the transmit address is being changed on the fly, care should be exercised not to corrupt the receive address, since both addresses are located in the TRTA, thus maintaining the convention of allowing the transmitting device to specify which devices can receive.

9.4.5 TDM Serial Port Interface Exception Conditions

Because of the nature of the TDM architecture, with the ability for one processor to transmit in multiple slots, the concepts of overrun and underflow become indeterminate. Therefore, the overrun and underflow flags are not active in TDM mode.

In the receiver, if TRCV has not been read and a valid receive operation is initiated (because of the value on TADD and the device's receive address), the present value of TRCV is overwritten; the receiver is ***not*** halted. On the other hand, if TDXR has not been updated before a transmission, the TADD or TDAT lines are not driven, and these pins remain in the high-impedance state. This mode of operation prevents spurious transmits from occurring.

If a TFRM pulse occurs at an improper time during a frame, the TDM port is not able to continue functioning properly, since slot and bit numbers become ambiguous when this occurs. Therefore, only one TFRM should occur every 128 TCLK cycles. Unlike the serial port, the TDM port cannot be reinitialized with a frame sync pulse during transmission. To correct an improperly timed TFRM pulse, the TDM port must be reset.

9.4.6 Examples of TDM Serial Port Interface Operation

The following is an example of TDM serial port operation, showing the contents of some of the key device registers involved, and explaining the effect of this configuration on port operation. In this example, eight devices are connected to the TDM serial port as shown in Figure 9–34.

Table 9–14 shows the TADD value during each of the eight channels given the transmitter and receiver designations shown. This example shows the configuration for eight devices to communicate with each other. In this example, device 0 broadcasts to all other device addresses during slot 0. In subsequent time slots, devices 1–7 each communicate to one other processor.

Figure 9–34. TDM Example Configuration Diagram

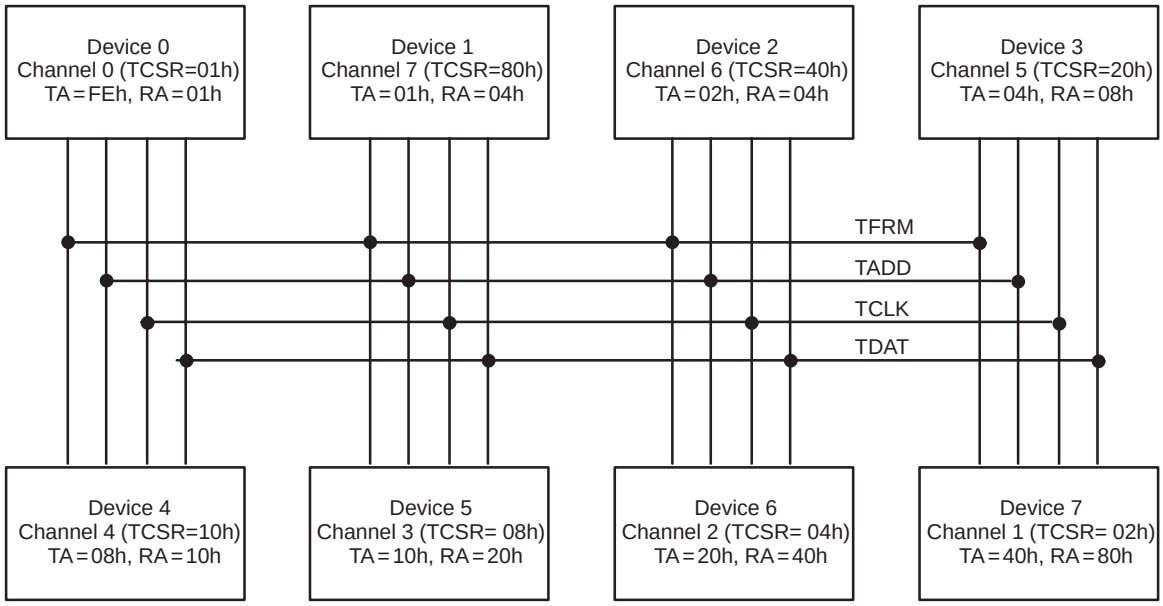


Table 9–14. Interprocessor Communications Scenario

Channel	TADD Data	Transmitter Device	Receiver Device(s)
0	FEh	0	1–7
1	40h	7	6
2	20h	6	5
3	10h	5	4
4	08h	4	3
5	04h	3	2
6	02h	2	1
7	01h	1	0

Table 9–15 shows the TDM serial port register contents of each device that results in the scenario given in Table 9–14. Device 0 provides the clock and frame control signals for all channels and devices. The TCSR and TRTA contents specify which device is to transmit on a given channel and which devices are to receive.

Table 9–15. TDM Register Contents

Device	TSPC	TRTA	TCSR
0	xxF9h	FE01h	xx01h
1	xxC9h	0102h	xx80h
2	xxC9h	0204h	xx40h
3	xxC9h	0408h	xx20h
4	xxC9h	0810h	xx10h
5	xxC9h	1020h	xx08h
6	xxC9h	2040h	xx04h
7	xxC9h	4080h	xx02h

In this example, the transmit address of a given device (the upper byte of TRTA) matches the receive address (the lower byte of TRTA) of the receiving device. Note, however, that it is not necessary for the transmit and receive addresses to match exactly; the matching operation implemented in the receiver is a bitwise AND operation. Thus, it is only necessary that one bit in the field matches for a receive to occur. The advantage of this scheme is that a transmitting device can select the device or devices to receive its transmitted data by simply changing its transmit address (as long as each devices' receive address is unique, the receive address of the receiving device does not need to be changed). In the example, device 0 can transmit to any combination of the other devices by merely writing to the upper byte of TRTA. Therefore, if a transmitting device changed its TRTA to 8001h on the fly, it would transmit only to device 7.

A device may also transmit to itself, because both the transmit and receive operations are executed on the rising edge of TCLK (see the '54x data sheet for TDM interface timings). To enable this type of loopback, it is necessary to use the standard TDM port interface connections as shown in Figure 9–31. Then, if device 0 has a TRTA of 0101h, it would transmit only to itself.

As an illustration of the proper operation of a TDM serial port, Example 9–5 through Example 9–8 define a sequence of actions. This illustration is based on the use of interrupts to handle the normal I/O between the serial port and CPU. The '542 peripheral configuration has been used as a reference for these examples.

In Example 9–5 the procedure for a one-way transmit of a sequence of values from device 0 to device 1 is shown. Device 0 transmits in slot 0 and has a transmit address of 01h. Example 9–7 shows the procedure for device 1. It has a receive address of 01h.

Example 9–5. TDM Serial Port Transmit Initialization Routine

	Action	Description
1)	Reset and initialize the TDM serial port by writing 0039h to TSPC.	This places both the transmit and receive portions of the TDM serial port in reset and sets up the serial port to operate with internally generated TFRM and TCLK signals in TDM mode.
2)	Clear any pending TDM serial port transmit interrupts by writing 0080h to IFR.	Eliminate any interrupts that may have occurred before initialization.
3)	Enable the TDM serial port interrupts by ORing 0080h with IMR.	Enable transmit interrupts.
4)	Enable interrupts globally (if necessary) by clearing the INTM bit in ST1.	Interrupts must be globally enabled for the CPU to respond.
5)	Write 0001h to TCSR.	This selects time slot 0 as the transmission time slot for this device.
6)	Write 0100h to TRTA.	This sets up this device to transmit data to the device receiving at address 01h. It also sets up this device to ignore all received data.
7)	Start the serial port by writing 0049h to TSPC.	This takes the transmit portion of the serial port out of reset and starts operations with the conditions defined in steps 1, 5 and 6.
8)	Perform a handshake to verify that the receiving device is ready to receive data.	For a single device pair, this could make use of $\overline{\text{BIO}}$ and XF. For several devices this might mean that the device generating TFRM and TCLK broadcasts a command to all other devices until each one returns an acknowledge.
9)	Write the first data value to TDXR (if not already done in step 8).	This initiates serial port transmit operations since TADD and TDAT are not driven if new data is not written to TDXR.

Example 9–6. TDM Serial Port Transmit Interrupt Service Routine

	Action	Description
1)	Save any context that may be modified on the stack.	The operating context of the interrupted code must be maintained.
2)	Write TDXR with a new value from a predetermined location in memory.	Write the new transmit data for the ISR.
3)	Restore the context that was saved in step 1.	The operating context of the interrupted code must be maintained.
4)	Return from the ISR with an RETE to reenable interrupts.	Interrupts must be reenabled for the CPU to respond to the next interrupt.

Example 9–7. TDM Serial Port Receive Initialization Routine

	Action	Description
1)	Reset and initialize the TDM serial port by writing 0009h to TSPC.	This places both the transmit and receive portions of the TDM serial port in reset and sets up the serial port to operate with externally generated TFRM and TCLK signals in TDM mode.
2)	Clear any pending TDM serial port receive interrupts by writing 0040h to IFR.	Eliminate any interrupts that may have occurred before initialization.
3)	Enable the TDM serial port interrupts by ORing 0040h with IMR.	Enable receive interrupts.
4)	Enable interrupts globally (if necessary) by clearing the INTM bit in ST1.	Interrupts must be globally enabled for the CPU to respond.
5)	Write 0000h to TCSR.	This sets up this device to not transmit in any time slot.
6)	Write 0001h to TRTA.	This sets up this device to not address any device. It also sets up this device to receive data sent to address 01h.
7)	Perform a handshake to notify the transmitting device that it is okay to send data.	For a single device pair, this could make use of $\overline{\text{BIO}}$ and XF. For several devices, this might mean that the device waits for a broadcast command and then returns an acknowledge.

Example 9–8. TDM Serial Port Receive Interrupt Service Routine

	Action	Description
1)	Save any context that may be modified on the stack.	The operating context of the interrupted code must be maintained.
2)	Read TDRR and write the value to a predetermined location in memory.	Read the new received data for the ISR.
3)	Restore the context that was saved in step 1.	The operating context of the interrupted code must be maintained.
4)	Return from the ISR with an RETE to reenable interrupts.	Interrupts must be reenabled for the CPU to respond to the next interrupt.

Chapter 10

External Bus Operation

This chapter describes external bus operation and control for memory and I/O accesses. Some of the external bus operation and control features of the ’54x include software wait states, bank-switching logic, and hold logic. The ’54x supports a wide range of system interfacing requirements.

Topic **Page**

10.1 External Bus Interface

The ’54x external interface consists of data buses, address buses, and a set of control signals for accessing off-chip memory and I/O ports. Table 10–1 lists key signals for the external interface.

Table 10–1. Key External Interface Signals

Signal Name	’541 ’542, ’543, ’545, ’546	’548, ’549	Description
A0–A15	15–0	22–0	Address bus
D0–D15	15–0	15–0	Data bus
$\overline{\text{MSTRB}}$	✓	✓	External memory access strobe
$\overline{\text{PS}}$	✓	✓	Program space select
$\overline{\text{DS}}$	✓	✓	Data space select
$\overline{\text{IOSTRB}}$	✓	✓	I/O access strobe
$\overline{\text{IS}}$	✓	✓	I/O space select
R/$\overline{\text{W}}$	✓	✓	Read/write signal
READY	✓	✓	Data ready to complete cycle
$\overline{\text{HOLD}}$	✓	✓	Request for control of memory interface
$\overline{\text{HOLDA}}$	✓	✓	Acknowledge $\overline{\text{HOLD}}$ request
$\overline{\text{MSC}}$	✓	✓	Microstate complete
$\overline{\text{IAQ}}$	✓	✓	Instruction acquisition
$\overline{\text{IACK}}$	✓	✓	Interrupt acknowledge

The parallel interface consists of two mutually-exclusive interfaces controlled by the $\overline{\text{MSTRB}}$ and $\overline{\text{IOSTRB}}$ signals. $\overline{\text{MSTRB}}$ is activated for memory accesses (program or data), and $\overline{\text{IOSTRB}}$ is used to access I/O ports. The R/$\overline{\text{W}}$ signal controls the direction of the accesses.

The external ready input signal (READY) and the software-generated wait states allow the processor to interface with memory and I/O devices of varying speeds. When communicating with slower devices, the CPU waits until the other device completes its function and sends the READY signal to continue execution.

In some cases, wait states are needed only when transitions are made between two external memory devices. The programmable bank-switching logic provides automatic insertion of a wait state in these situations.

The hold mode allows an external device to take control of the ’54x external buses to access the resources in the ’54x external program, data, and I/O memory spaces. Two hold mode types, normal mode and concurrent DMA mode, are available.

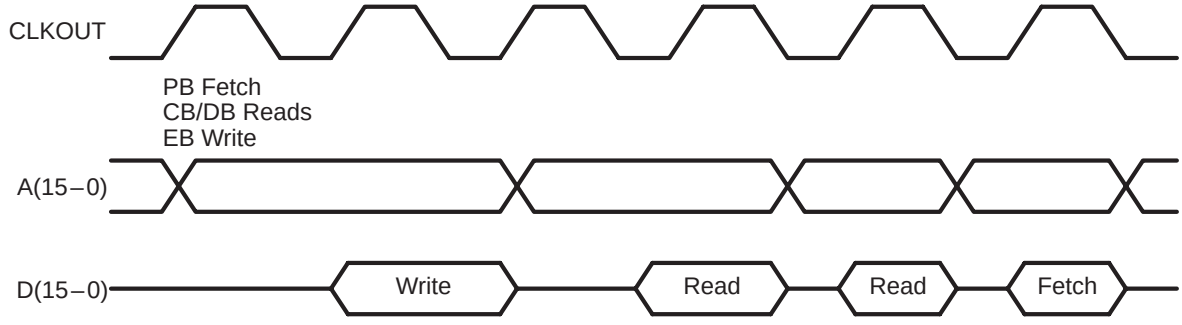
When the CPU addresses internal memory, the data bus is placed in the high-impedance state. However, the address bus and the memory-select signals (program select ($\overline{PS}$), data select ($\overline{DS}$), and I/O select ($\overline{IS}$)) maintain the previous state. The $\overline{MSTRB}$, $\overline{IOSTRB}$, R/$\overline{W}$, $\overline{IAQ}$, and $\overline{MSC}$ signals remain inactive. If the address visibility mode (AVIS) bit, located in the PMST, is set to 1, the internal program address is placed on the address bus with an active $\overline{IAQ}$.

10.2 External Bus Priority

The ’54x CPU has one program bus (PB), three data buses (CB, DB, and EB), and four address buses (PAB, CAB, DAB, and EAB). The CPU can access its buses simultaneously because of its pipelined structure; however, the external interface can support only one access per cycle. A pipeline conflict occurs if, in a single cycle, the CPU accesses external memory twice to fetch an instruction, a data-memory operand, or an external I/O device. This pipeline conflict is automatically resolved by a predetermined priority, defined by the stage of the pipeline.

Figure 10–1 shows multiple CPU accesses to fetch an instruction, and to write and read data operands over the external interface in one cycle. Data accesses have a higher priority than program-memory fetches: the program-memory fetch cannot begin until all CPU data accesses are completed.

Figure 10–1. External Bus Interface Priority



Pipeline conflicts occur when the program and data are in external memory and a single-operand write instruction is followed by a dual-operand read or a 32-bit operand read. The following sequence of instructions shows the pipeline conflict discussed.

```
ST     T,*AR6        ;Smem write operation
LD     *AR4+,A       ;Xmem and Ymem read operation
||MAC  *AR5+, B
```

Chapter 7, *Pipeline,* describes pipeline operation and conflicts in detail.

10.3 External Bus Control

Two units in the ’54x control the external bus: the wait-state generator and the bank-switching logic. These units are controlled by two registers: the software wait-state register (SWWSR) and the bank-switching control register (BSCR).

10.3.1 Wait-State Generator

The software-programmable wait-state generator can extend external bus cycles by up to seven machine cycles (14 machine cycles on ’549 devices), providing a convenient means to interface the ’54x to slower external devices. Devices that require more than seven wait states can be interfaced using the hardware READY line. When all external accesses are configured for zero wait states, the internal clocks to the wait-state generator are shut off; shutting off these paths from the internal clocks allows the device to run with lower power consumption.

The software-programmable wait-state generator is controlled by the 16-bit software wait-state register (SWWSR), which is memory-mapped to address 0028h in data space.

The program and data spaces each consist of two 32K-word blocks; the I/O space consists of one 64K-word block. Each of these blocks has a corresponding 3-bit field in the SWWSR. These fields are shown in Figure 10–2 and described in Table 10–2. The ’548 and ’549 are described in Table 10–3.

The value of a 3-bit field in SWWSR specifies the number of wait states to be inserted for each access in the corresponding space and address range. The minimum value, which adds no wait states, is 0 (000b). A value of 7 (111b) provides the maximum number of wait states.

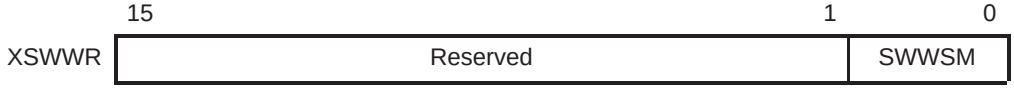
Figure 10–2. Software Wait-State Register (SWWSR) Diagram

15	14–12	11–9	8–6	5–3	2–0
Reserved/XPA†	I/O	Data	Data	Program	Program
R	R/W	R/W	R/W	R/W	R/W

† XPA bit on ’548 and ’549 only

The ’549 has an extra bit (software wait-state multiplier, SWWSM) that resides in XSWWR, which is memory mapped to address 002Bh in data space.

Figure 10–3. Extended Software Wait-State Register (XSWWR) Diagram



When SWWSM is set to 1, the wait states are multiplied by two extending the number of wait states from 7 to 14.

Table 10–2. Software Wait-State Register (SWWSR) Bit Summary

Bit	Name	Reset Value	Function
15	Reserved	0	Reserved. In the ’548 and ’549, this bit changes the operation of the program fields (see Table 10–3).
14–12	I/O	1	I/O space. The field value (0–7) corresponds to the number of wait states for I/O space 0000–FFFFh.
11–9	Data	1	Data space. The field value (0–7) corresponds to the number of wait states for data space 8000–FFFFh.
8–6	Data	1	Data space. The field value (0–7) corresponds to the number of wait states for data space 0000–7FFFh.
5–3	Program	1	Program space. The field value (0–7) corresponds to the number of wait states for program space 8000–FFFFh.
2–0	Program	1	Program space. The field value (0–7) corresponds to the number of wait states for program space 0000–7FFFh.

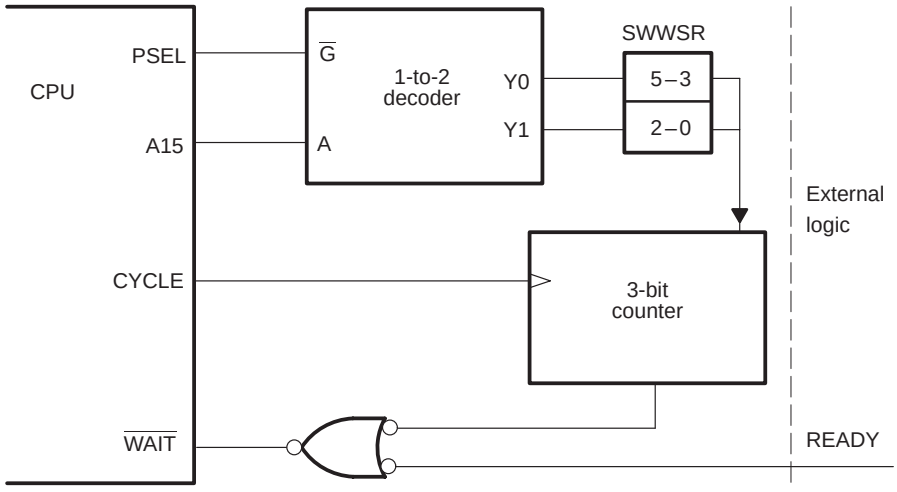
Table 10–3. TMS320C548/549 Software Wait-State Register (SWWSR) Bit Summary

Bit	Name	Reset Value	Function
15	XPA	0	Extended program address control. Selects the address ranges selected by the program fields.
14–12	I/O	1	I/O space. The field value (0–7) corresponds to the number of wait states for I/O space 0000–FFFFh.
11–9	Data	1	Data space. The field value (0–7) corresponds to the number of wait states for data space 8000–FFFFh.
8–6	Data	1	Data space. The field value (0–7) corresponds to the number of wait states for data space 0000–7FFFh.

Bit	Name	Reset Value	Function
5–3	Program	1	Program space. The field value (0–7) corresponds to the number of wait states for: ❑ XPA = 0: xx8000 – xxFFFFh ❑ XPA = 1: 400000h–7FFFFF
2–0	Program	1	Program space. The field value (0–7) corresponds to the number of wait states for: ❑ XPA = 0: xx0000–xx7FFFh ❑ XPA = 1: 000000–3FFFFFh

Figure 10–4 is a block diagram of the wait-state generator logic for external program space. When an external program access is decoded, the appropriate field of the SWWSR is loaded into the counter. If the field is not 000, a not-ready signal is sent to the CPU and the wait-state counter is started. The not-ready condition is maintained until the counter decrements to 0 and the external READY line is set high. The external READY and the wait-state READY are ORed together to generate the CPU $\overline{\text{WAIT}}$ signal. The READY line is machine-sampled at the falling edge of CLKOUT. The processor detects READY only if a minimum of two software wait states are programmed. The external READY line is not sampled until the last wait-state cycle.

Figure 10–4. Software Wait-State Generator Block Diagram



At reset, all fields in the SWWSR are set to 111b (SWWSR = 7FFFh), the maximum number of wait states for external accesses. This feature ensures

that the CPU can communicate with slow external memories during processor initialization.

10.3.2 Bank-Switching Logic

Programmable bank-switching logic allows the ’54x to switch between external memory banks without requiring external wait states for memories that need several cycles to turn off. The bank-switching logic automatically inserts one cycle when accesses cross memory-bank boundaries inside program or data space.

Bank switching is defined by the bank-switching control register (BSCR), which is memory-mapped at address 0029h. Figure 10–5 shows the BSCR and its fields are described in Table 10–4.

Figure 10–5. Bank-Switching Control Register (BSCR) Diagram

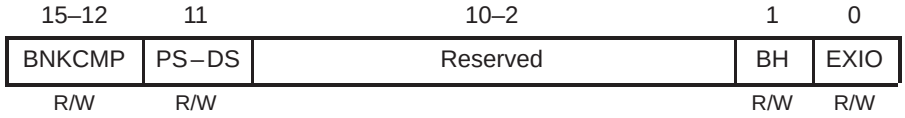


Table 10–4. Bank-Switching Control Register (BSCR) Bit Summary

Bit	Name	Reset Value	Function
15–12	BNKCMP	–	Bank compare. Determines the external memory-bank size. BNKCMP is used to mask the four MSBs of an address. For example, if BNKCMP = 1111b, the four MSBs (bits 12–15) are compared, resulting in a bank size of 4K words. Bank sizes from 4K words to 64K words are allowed. Table 10–5 on page 10-10 shows the relationship between BNKCMP and the address range.
11	PS–DS	–	Program read–data read access. Inserts an extra cycle between consecutive accesses of program read and data read, or data read and program read.
			PS–DS = 0 No extra cycles are inserted by this feature.
			PS–DS = 1 One extra cycle is inserted between consecutive accesses of program read and data read, or data read and program read.
10–2	Reserved	–	These bits are reserved.
1	BH	0	Bus holder. Controls the bus holder:
			BH = 0 The bus holder is disabled.
			BH = 1 The bus holder is enabled. The data bus, D(15–0), is held in the previous logic level.
0	EXIO	0	External bus interface off. The EXIO bit controls the external-bus-off function.

Bit	Name	Reset Value	Function	
			EXIO = 0	The external-bus-off function is disabled.
			EXIO = 1	The external-bus-off function is enabled. The address bus, data bus, and control signals become inactive after completing the current bus cycle. Table 10–6 on page 10-10 lists the state of the signals when the external bus interface is disabled. The DROM, MP/$\overline{\text{MC}}$, and OVLY bits in PMST and the HM bit in ST1 cannot be modified.

Table 10–5 summarizes the relationship between BNKCMP, the address bits to be compared, and the bank size. BNKCMP values not listed in the table are not allowed. Table 10–6 lists the state of the ports when the external bus interface is disabled (EXIO = 1).

Table 10–5. Relationship Between BNKCMP and Bank Size

BNKCMP					
Bit 15	**Bit 14**	**Bit 13**	**Bit 12**	**MSBs to Compare**	**Bank Size (16-Bit Words)**
0	0	0	0	None	64K
1	0	0	0	15	32K
1	1	0	0	15–14	16K
1	1	1	0	15–13	8K
1	1	1	1	15–12	4K

Table 10–6. State of Signals When External Bus Interface is Disabled (EXIO = 1)

Signal	State	Signal	State
A(15–0)	Previous state	R/$\overline{W}$	High level
D(15–0)	High impedance	$\overline{MSC}$	High level
$\overline{PS}$, $\overline{DS}$, $\overline{IS}$	High level	$\overline{IAQ}$	High level
$\overline{MSTRB}$, $\overline{IOSTRB}$	High level		

The EXIO and BH bits control the use of the external address and data buses. These bits should be set to 0 for normal operation. To reduce power dissipation, especially if external memory is never or only infrequently accessed, EXIO and BH can be set to 1.

When the EXIO bit in BSCR is set to 1, the CPU cannot modify the the HM bit in ST1 and cannot modify the memory map by changing the value of the DROM, MP/$\overline{MC}$, and OVLY bits in PMST.

The ’54x has an internal register that contains the MSBs (as defined by the BNKCMP field) of the last address used for a read or write operation in program or data space. If the MSBs of the address used for the current read do not match those contained in this internal register, the $\overline{\text{MSTRB}}$ (memory strobe) signal is not asserted for one CLKOUT cycle. During this extra cycle, the address bus switches to the new address. The contents of the internal register are replaced with the MSBs for the read of the current address. If the MSBs of the address used for the current read match the bits in the register, a normal read cycle occurs.

If repeated reads are performed from the same memory bank, no extra cycles are inserted. When a read is performed from a different memory bank, memory conflicts are avoided by inserting an extra cycle. An extra cycle is inserted only if a read memory access is followed by another read memory access. This feature can be disabled by clearing BNKCMP to 0.

The ’54x bank-switching mechanism automatically inserts one extra cycle in the following cases:

- A program-memory read followed by another program-memory or data-memory read from a different memory bank.
- A program-memory read followed by a data-memory read when the PS–DS bit is set to 1.
- A program-memory read followed by another program-memory read from a different page (with the ’548 and ’549).
- A data-memory read followed by another program-memory or data-memory read from a different memory bank.
- A data-memory read followed by a program-memory read when the PS–DS bit is set to 1.

Figure 10–6 illustrates the addition of an inactive cycle when memory banks are switched.

Figure 10–6. Bank Switching Between Memory Reads

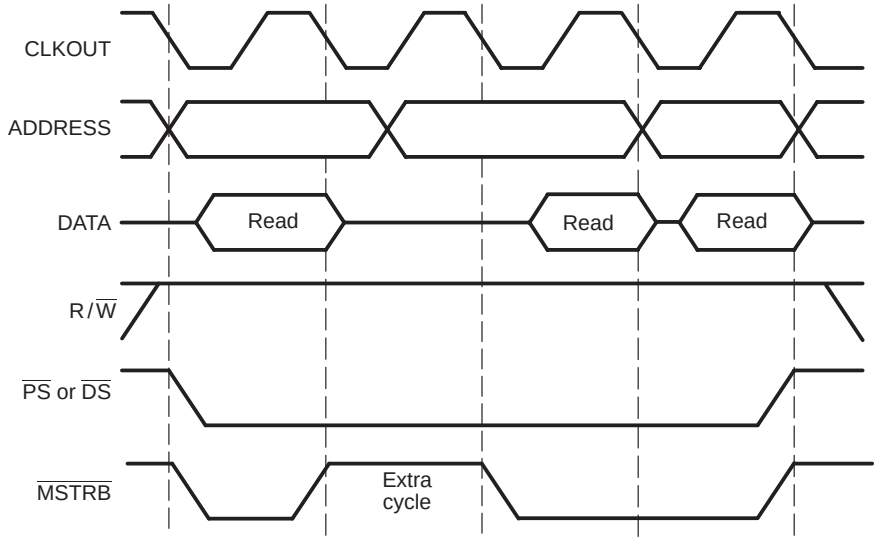
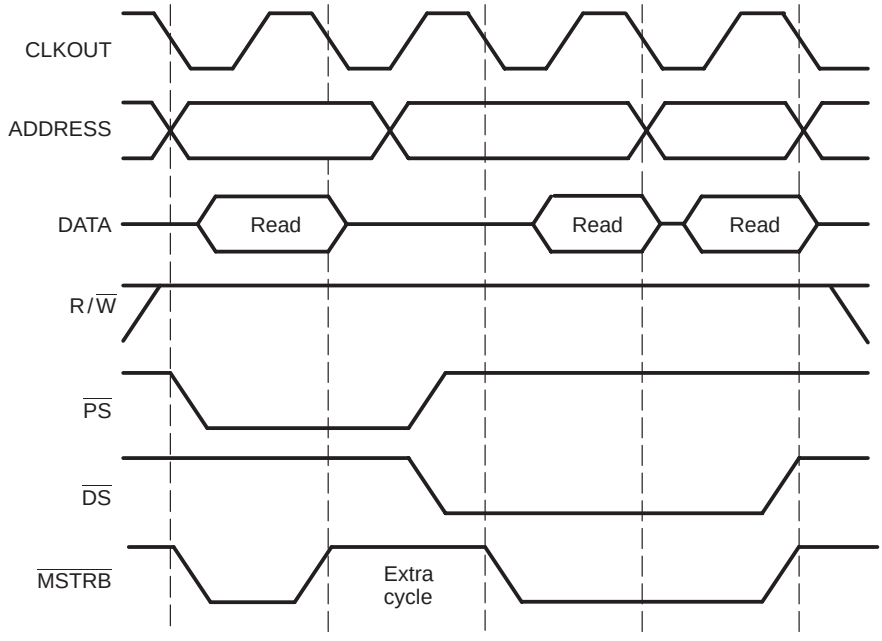


Figure 10–7 illustrates the insertion of the extra cycle between a consecutive program read and a data read.

Figure 10–7. Bank Switching Between Program Space and Data Space



10.4 External Bus Interface Timing

All external bus accesses complete in an integral number of CLKOUT cycles. One CLKOUT cycle is defined as the time from one falling edge to the next falling edge of CLKOUT. Some external bus accesses with no wait states, for example, memory writes, or I/O writes and I/O reads, take two cycles. Memory reads take one cycle; however, if a memory read follows a memory write, or vice versa, the memory read takes an additional half-cycle. The following subsections discuss zero wait-state accesses, unless otherwise specified.

10.4.1 Memory Access Timing

The $\overline{\text{MSTRB}}$ signal is low for the active portion of reads and writes; an active portion of a memory access lasts (at least) one CLKOUT cycle. In addition, a transition CLKOUT cycle occurs before and after the active portion for writes. During this transition cycle:

- $\overline{\text{MSTRB}}$ is high.
- R/$\overline{\text{W}}$ changes on CLKOUT's rising edge when required.
- The address changes on CLKOUT's rising edge in the following cases. In all other instances, the address changes on the CLKOUT falling edge.
  - The previous CLKOUT cycle was the active portion of a memory write.
  - A memory read is followed by a memory write.
  - A memory read is followed by an I/O write.
  - A memory read is followed by an I/O read.
- $\overline{\text{PS}}$, $\overline{\text{DS}}$, or $\overline{\text{IS}}$ changes, if necessary, when the address changes.

Figure 10–8 shows a read-read-write sequence with $\overline{\text{MSTRB}}$ active and no wait states. The data is read as late in the cycle as possible to allow for maximum access time from a valid address. Although the external writes take two cycles, internally they require only one cycle if no accesses to the external interface are in progress. This helps maintain processing throughput at the maximum level possible.

The timing diagram illustrates these concepts:

- Back-to-back reads from the same bank are single-cycle accesses.
- $\overline{\text{MSTRB}}$ stays low during back-to-back reads.
- $\overline{\text{MSTRB}}$ goes high for one cycle during read-to-write transitions to frame the address and R/$\overline{\text{W}}$ signal changes.

Figure 10–8. Memory Interface Operation for Read-Read-Write

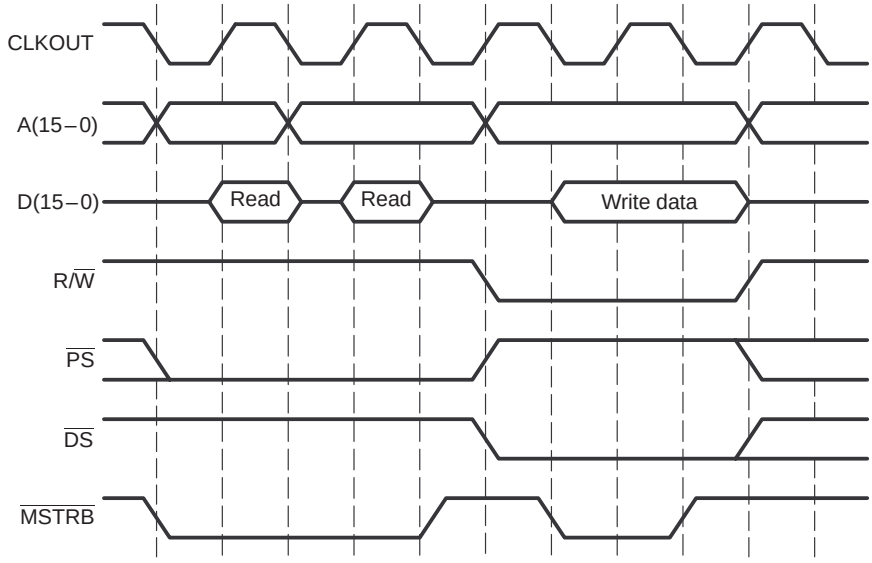
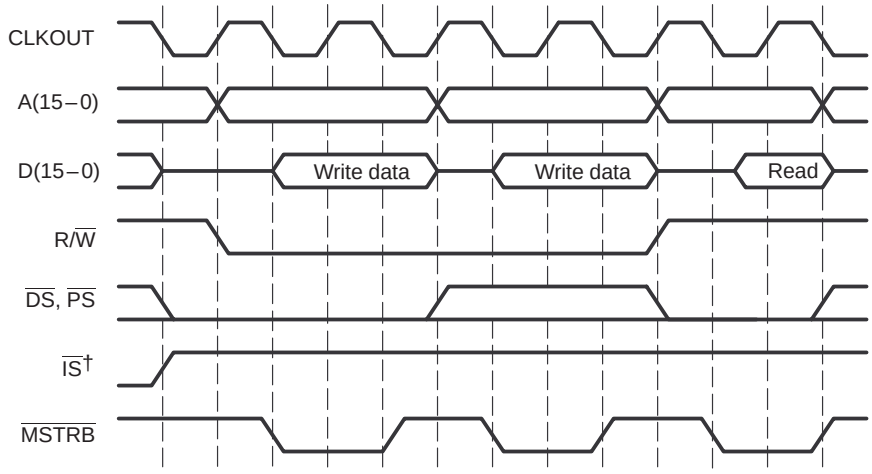


Figure 10–9 shows a write-write-read sequence with $\overline{\text{MSTRB}}$ active and no wait states. The address and data written are held valid approximately one half-cycle after $\overline{\text{MSTRB}}$ changes.

The timing diagram illustrates these concepts:

- $\overline{\text{MSTRB}}$ goes high at the end of every write cycle to disable the memory while the address and/or R/$\overline{\text{W}}$ signal changes.
- Each write takes two cycles.
- A read following a write takes two cycles.

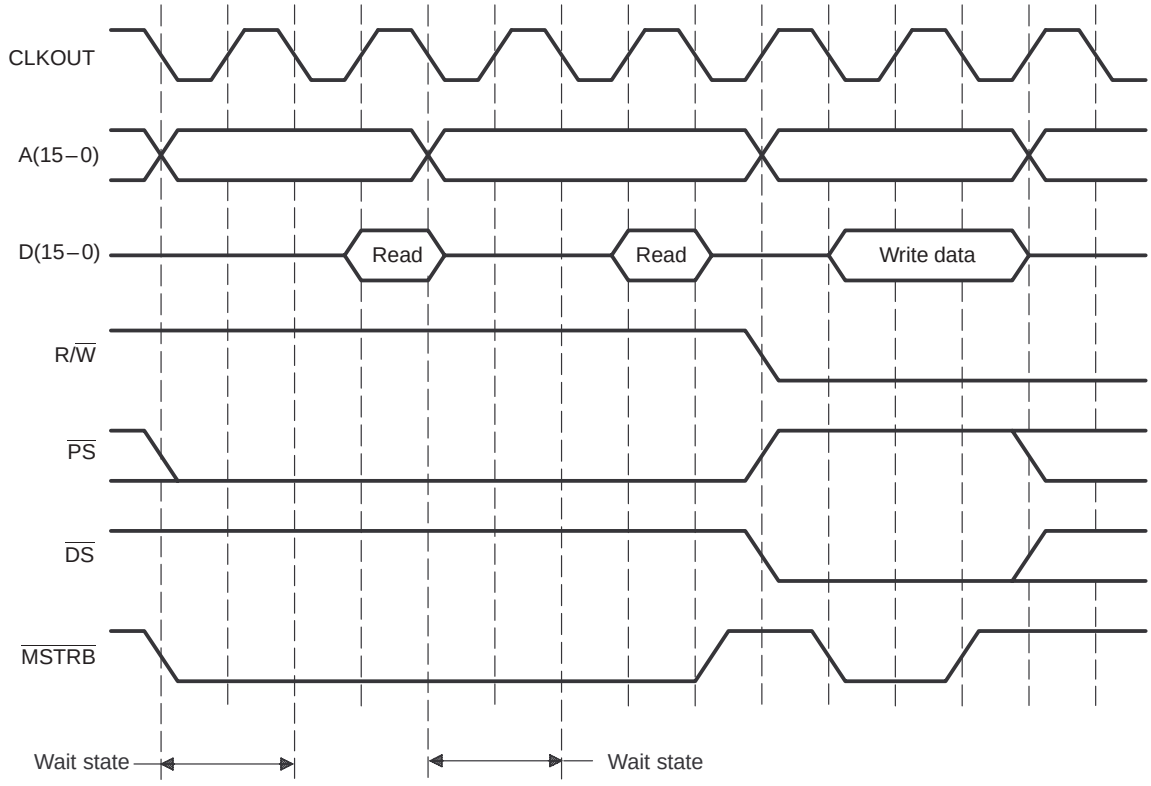
Figure 10–9. Memory Interface Operation for Write-Write-Read



† Assuming that an I/O write preceded the first memory write

Figure 10–10 shows a read-read-write sequence using $\overline{\text{MSTRB}}$ active and one wait state. Because the reads are normally one cycle, they are extended by one additional cycle for the wait state. However, the write, which is already two cycles, is not extended for the first software wait state.

Figure 10–10. Memory Interface Operation for Read-Read-Write (Program-Space Wait States)



10.4.2 I/O Access Timing

In I/O accesses, the active portion of reads and writes lasts two cycles (with no wait states). Otherwise, the timing for these accesses is the same as for memory accesses. During these cycles, the address changes on the falling edge of CLKOUT, except for a memory access to I/O access sequence. $\overline{\text{IOSTRB}}$ is low from one rising edge to the next rising edge of the CLKOUT cycle.

Figure 10–11 shows a read-write-read sequence for $\overline{\text{IOSTRB}}$ with no wait states. For $\overline{\text{IOSTRB}}$ accesses, reads and writes require a minimum of two cycles. Some off-chip peripherals can change their status bits during reads or writes; therefore, it is important that addresses remain valid when communicating with those peripherals. For reads and writes with $\overline{\text{IOSTRB}}$ active, $\overline{\text{IOSTRB}}$ is completely framed by the address to meet this requirement.

The timing diagram illustrates these concepts:

- Each I/O access takes two cycles.
- $\overline{\text{IOSTRB}}$ goes high at the end of each access to frame address and R/$\overline{\text{W}}$ signal changes.

Figure 10–11. Parallel I/O Interface Operation for Read–Write–Read

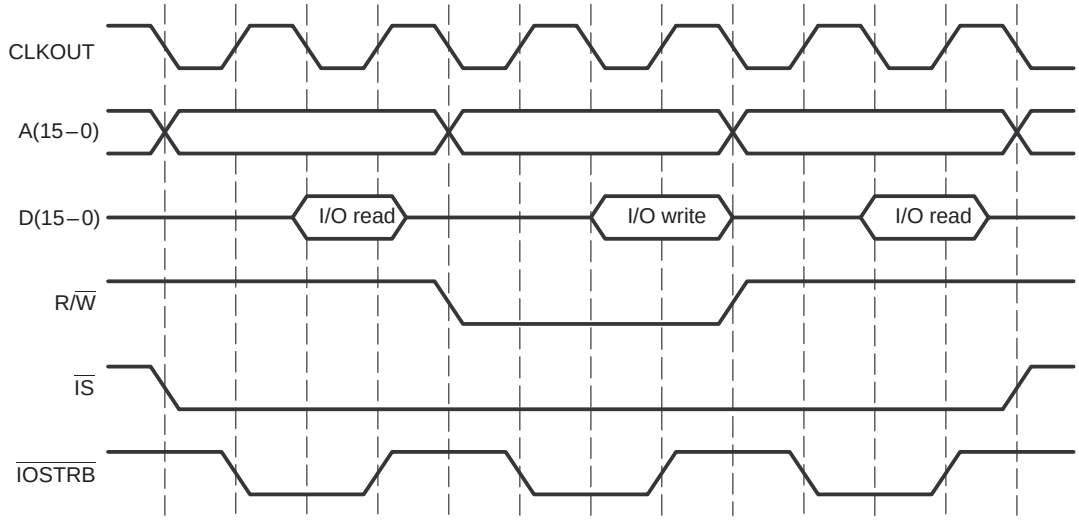
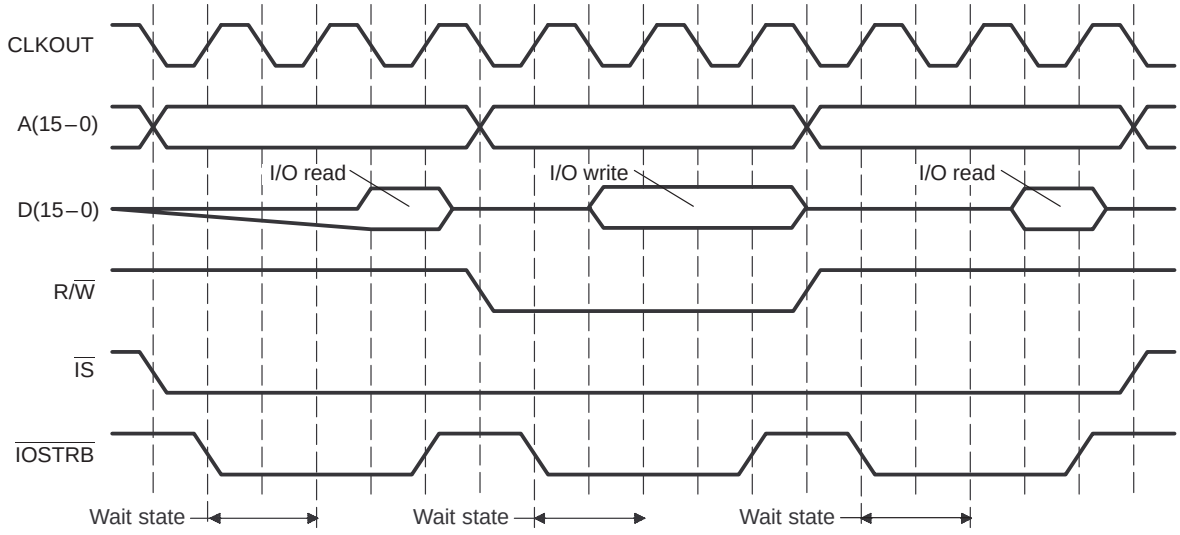


Figure 10–12 shows the same I/O space access with one wait state accesses. Each read and write access is extended by an additional cycle.

Figure 10–12. Parallel I/O Operation for Read-Write-Read (I/O-Space Wait States)



10.4.3 Memory and I/O Access Timing

Figure 10–13 through Figure 10–20 show the various transitions between memory reads and writes, and I/O reads and writes over the external interface bus.

The timing diagrams illustrate these concepts:

- ❑ I/O reads and writes take at least three cycles when they follow a memory read or write.
- ❑ Memory reads take two cycles when they follow an I/O read or write.

Figure 10–13. Memory Read and I/O Write

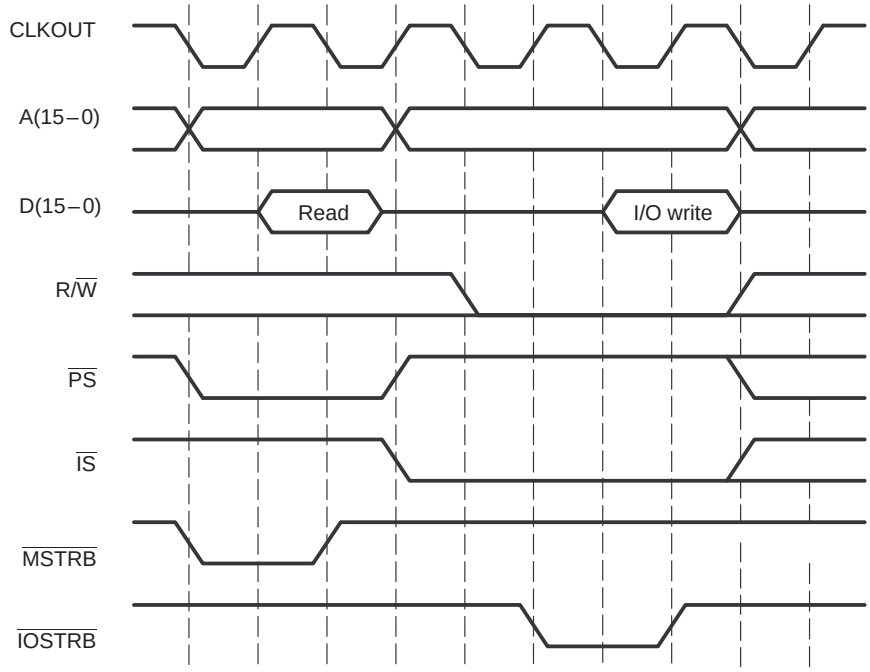


Figure 10–14. Memory Read and I/O Read

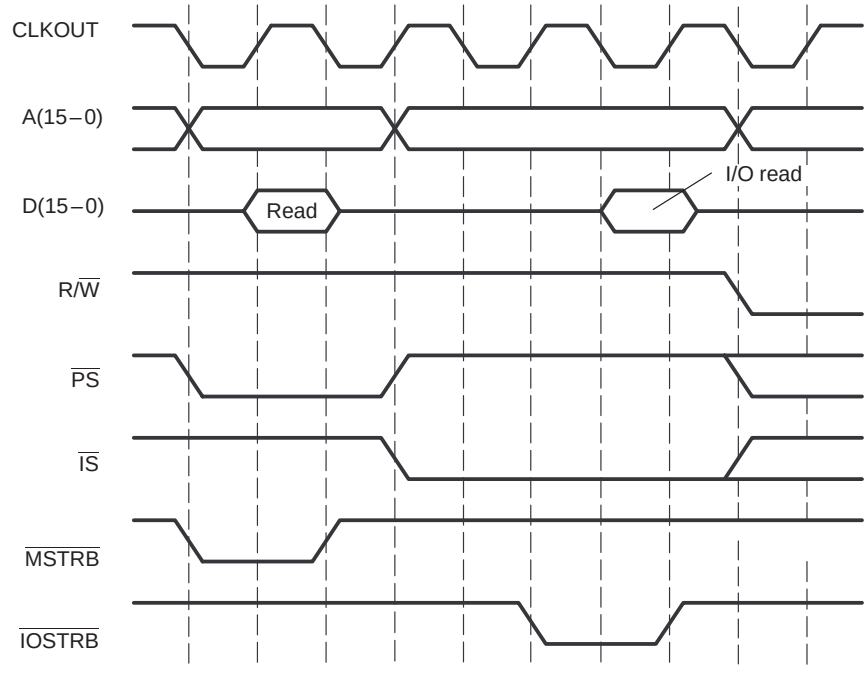


Figure 10–15. Memory Write and I/O Write

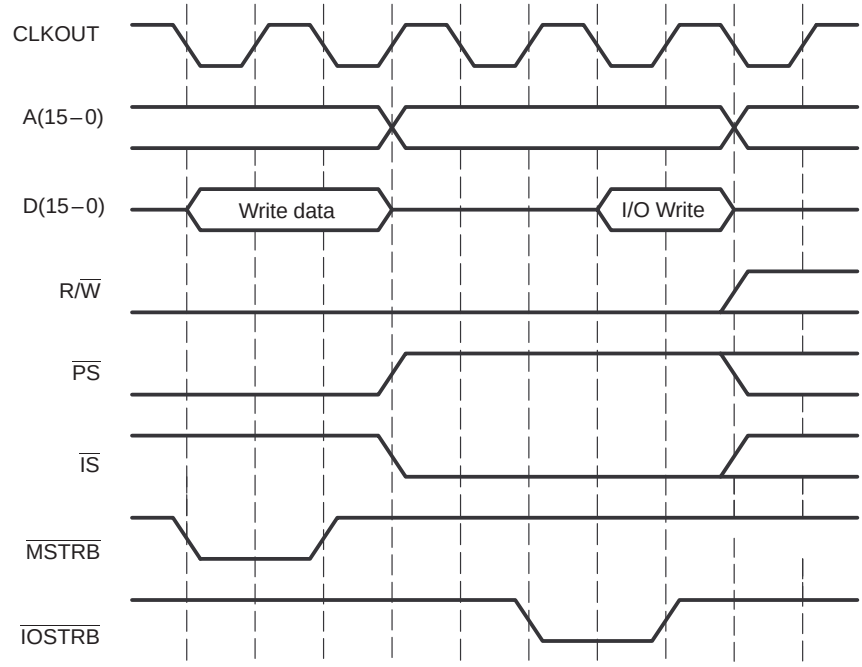


Figure 10–16. Memory Write and I/O Read

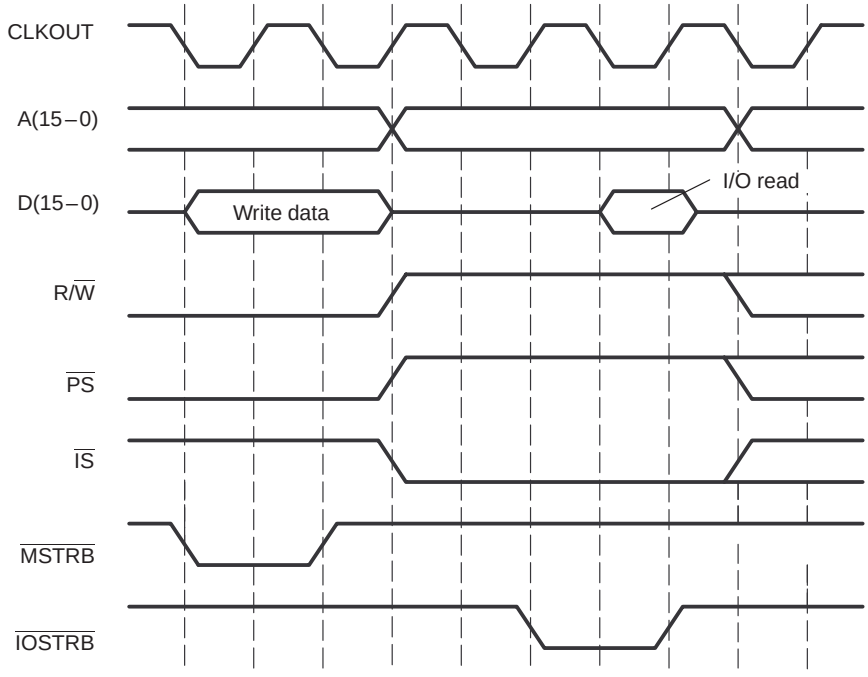


Figure 10–17. I/O Write and Memory Write

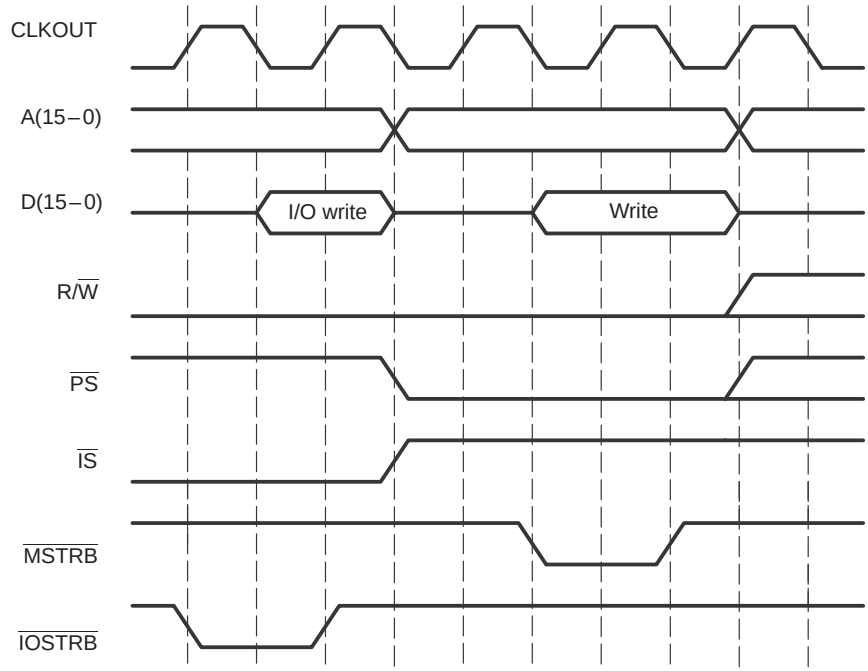


Figure 10–18. I/O Write and Memory Read

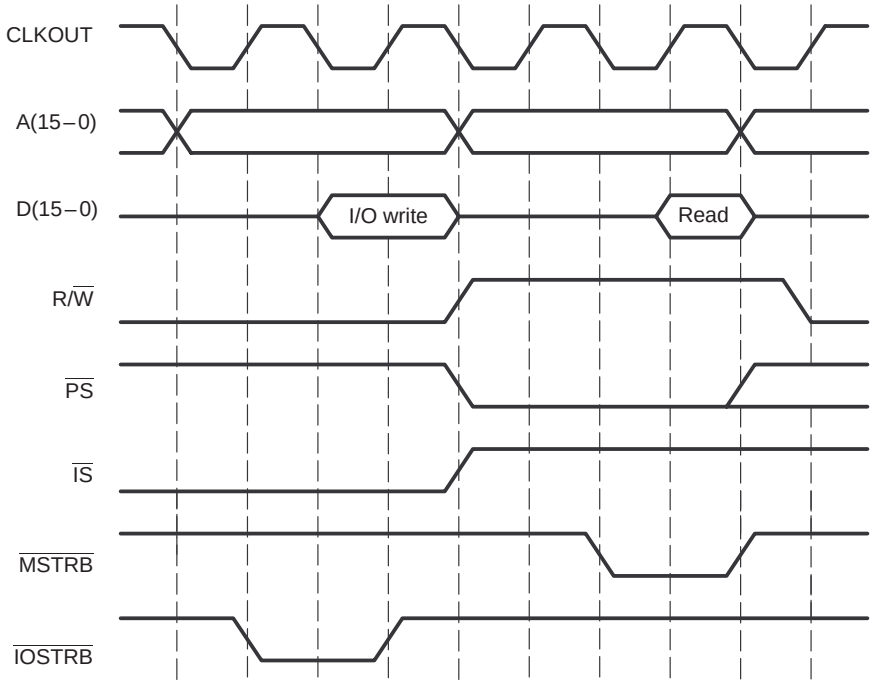


Figure 10–19. I/O Read and Memory Write

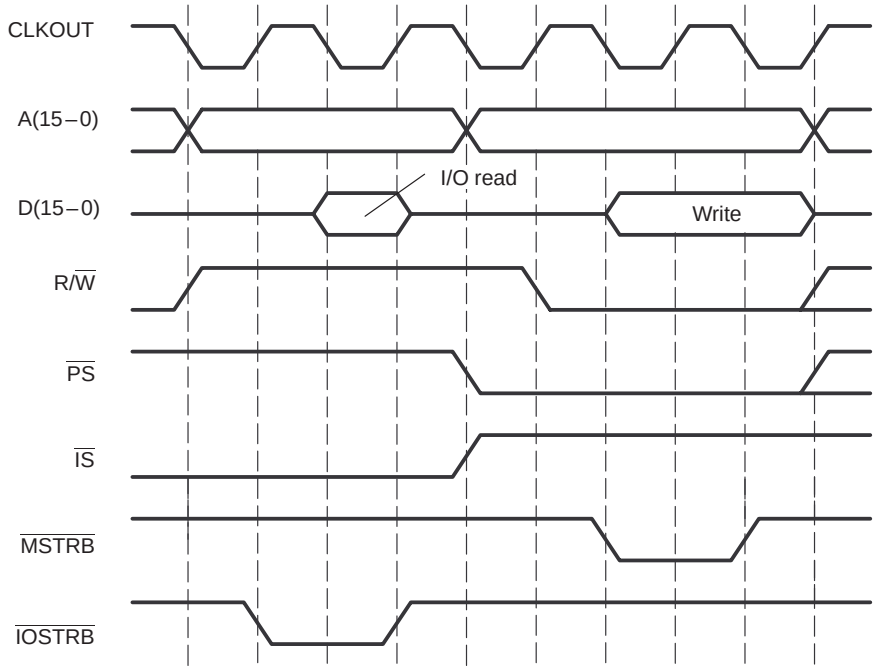
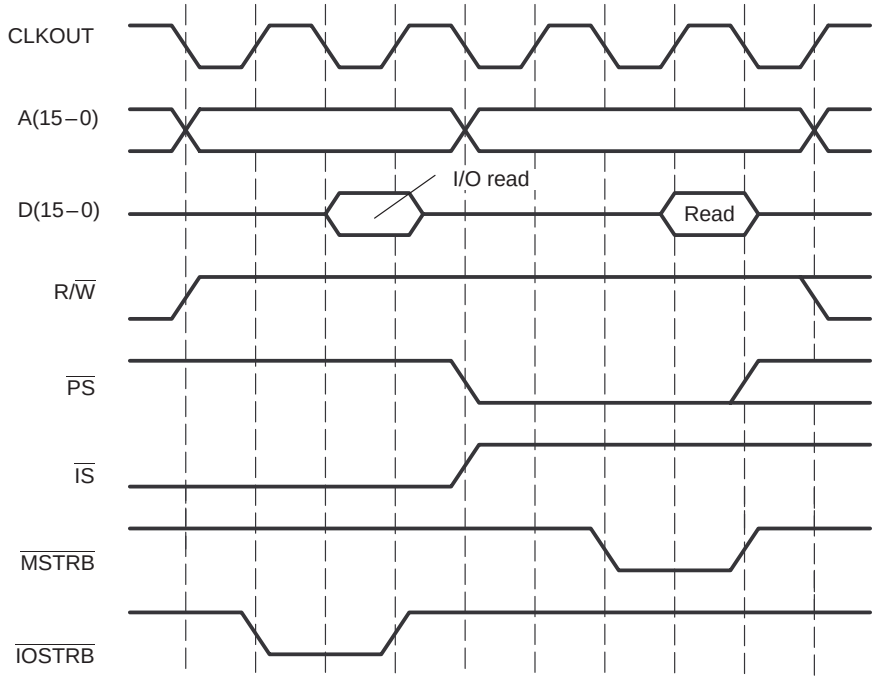


Figure 10–20. I/O Read and Memory Read



10.5 Start-Up Access Sequences

The ’54x transitions between active and inactive states when entering or leaving one of four modes: IDLE1, IDLE2, reset, or IDLE3.

Entering or leaving the first two modes, IDLE1 or IDLE2, requires no special consideration because the clocks to both the CPU and the on-chip peripherals remain active. However, special considerations are necessary when entering or leaving the other two modes:

- **Reset.** Hardware initialization takes place.
- **IDLE3.** The device makes a transition from a state where neither the CPU nor the on-chip peripherals are being clocked to an active state.

10.5.1 Reset

Figure 10–21 shows the reset sequence of the external bus. For proper reset operation, the $\overline{RS}$ signal must be active for at least two CLKOUT cycles. However, power-up and IDLE3 power-down mode require the reset signal to be active for more than two CLKOUT cycles. See Section 6.11, *Power-Down Modes*, on page 6-45 for more detailed information.

When the ’54x acknowledges a reset, the CPU terminates program execution and forces the program counter to FF80h. The address bus is driven with FF80h while $\overline{RS}$ is low.

The device enters its reset state, in reference to the external bus, according to three steps:

1) Four cycles after $\overline{RS}$ is asserted low, $\overline{PS}$, $\overline{MSTRB}$, and $\overline{IAQ}$ are driven high.
2) Five cycles after $\overline{RS}$ is asserted low, R/$\overline{W}$ is driven high, the data bus (if driven) goes into the high-impedance state, and the address bus is driven with FF80h.
3) The device enters its reset state.

When reset becomes inactive, program execution starts from the program memory location FF80h. The instruction acquisition signal ($\overline{IAQ}$) and the interrupt acknowledge signal ($\overline{IACK}$) become active, as shown in Figure 10–21, regardless of the state of the MP/$\overline{MC}$ signal.

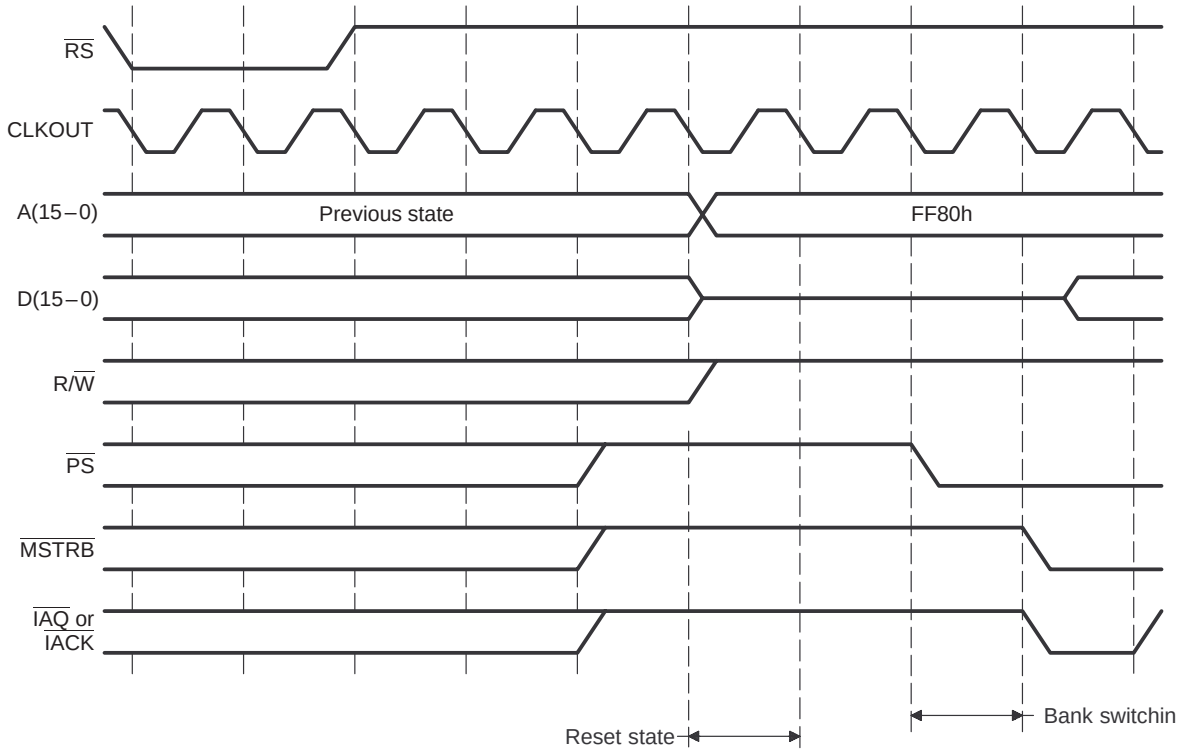
The device enters its active state, in reference to the external bus, according to three steps:

1) Five cycles after $\overline{RS}$ is asserted high, $\overline{PS}$ is driven low.

2) Six cycles after $\overline{RS}$ is asserted high, $\overline{MSTRB}$ and $\overline{IACK}$ are driven low.

3) One half-cycle later, the device is ready to read data and the device moves into its active state.

Figure 10–21. External Bus Reset Sequence



Notes: 1) $\overline{RS}$ is an asynchronous input and can be asserted at any point during a clock cycle. If the specified timings are met, the sequence shown occurs; otherwise, an additional delay of one clock cycle can occur.

2) During reset, the data bus, is placed in high impedance and the control signals are deasserted.

3) The reset vector is fetched with seven wait states.

4) The bank-switching cycle is inserted in the first access after reset.

10.5.2 IDLE3

The execution of the IDLE 3 instruction initiates the IDLE3 power-down mode. In this power-down mode, the PLL is halted completely to reduce power consumption. In the IDLE mode, the input clock can be kept running without additional power consumption, because a transfer gate inside the ’54x isolates the clock from the internal logic. The PLL must be restarted and locked before the ’54x can resume processing when it exits IDLE3. This power-down mode is terminated by activating the external interrupt pins, $\overline{\text{INT}}$n, $\overline{\text{NMI}}$ and $\overline{\text{RS}}$, in a particular sequence.

Table 10–7 shows the wake-up time of IDLE3 with the $\overline{\text{INT}}$n and $\overline{\text{NMI}}$ signals. These times are defined for the hardware-configurable PLL. The times for the software programmable PLL are given in section 8.4.2. When an interrupt pin goes low, an internal counter counts the input clock cycles. The initial value loaded in the counter depends on the PLL multiplication factor to ensure the counter down-time is greater than 50 μs for a 40 MIPS DSP.

Table 10–7. Counter Down-Time With PLL Multiplication Factors at 40 MHz Operation

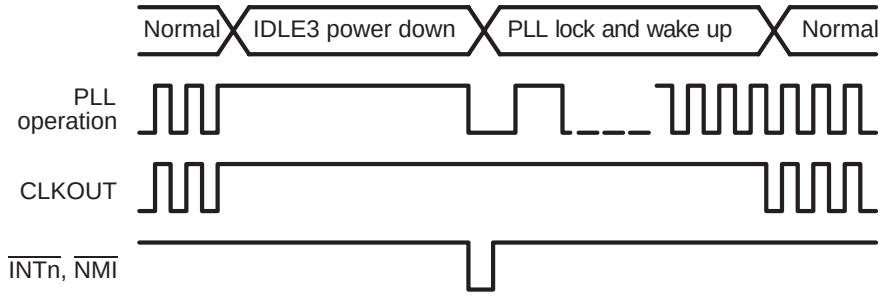
Counter Start Value	PLL Multiplication Factor	Equivalent CLKOUT Cycles (N)	Counter Down-Time at 40 MHz (μs)
2048	1	2048	51.2
2048	1.5	3072	76.8
1024	2	2048	51.2
1024	2.5	2560	64
1024	3	3072	76.8
512	4	2048	51.2
512	4.5	2304	57.6
512	5	2560	64

The counter down-times in Table 10–7 are valid when the input clock frequency is such that the CLKOUT frequency is 40 MHz when the PLL is locked. After the counter counts down to 0, the output from the locked PLL is fed to the internal logic.

A low pulse (minimum duration of 10 ns) of an external interrupt causes the ’54x to wake up from IDLE3 (see Figure 10–22). The locked PLL clock is fed into the CPU after *n* cycles. An additional three cycles are needed before the ’54x comes out of IDLE3. However, the ’54x does not need an extra two cycles for interrupt synchronization because the interrupt pulse initializes the interrupt synchronization, which is used to detect the interrupt immediately after the ’54x wake-up.

When reset is used to wake up from IDLE3, the counter is not used; the output from the PLL is immediately fed to the internal logic and the CLKOUT pin is asserted. The lock-up time is 50 μs for the PLL and CLKOUT to be stable. Therefore, it is necessary to keep the reset line low during this 50-μs lock-up time so that the ’54x does not start processing using an unstable clock.

Figure 10–22. IDLE3 Wake-Up Sequence



10.6 Hold Mode

The ’54x supports direct-memory accesses (DMAs) to its off-chip program, data, and I/O spaces. Two signals, $\overline{\text{HOLD}}$ and $\overline{\text{HOLDA}}$, allow an external device to take control of the processor’s buses. The processor acknowledges receiving a $\overline{\text{HOLD}}$ signal from an external device by bringing $\overline{\text{HOLDA}}$ low. The ’54x enters the hold mode and places its external address buses, data buses, and control signals into high impedance.

The hold mode (HM) status bit, located in ST1, determines the following operating modes for the hold function:

- ❑ Normal mode suspends program execution during a low $\overline{\text{HOLD}}$ signal.
- ❑ Concurrent DMA mode allows program execution to continue operating from internal memory (ROM or RAM).

When HM = 1, the ’54x operates in the normal mode. When HM = 0, the ’54x operates in the concurrent DMA mode. In this mode, the ’54x enters the hold state only if program execution is from external memory or if an external-memory operand is being accessed. However, if program execution is from internal memory and no external memory operands are accessed, the ’54x enters the hold state externally but program execution continues internally. Thus, a program can continue executing while an external DMA operation is performed. This makes the system operation more efficient.

Program execution ceases until $\overline{\text{HOLD}}$ is removed if the ’54x is in a hold state with HM = 0, and an internally executing program requires an external access, or a branch to an external address. Also, if a repeat instruction that requires the use of the external bus is executing with HM = 0 when a hold occurs, the hold state is entered after the current bus cycle. If a hold occurs when a repeat instruction is executing with HM = 1, the ’54x halts the execution after the current bus cycle, for either internal or external accesses. Upon reset, HM is cleared to 0. HM is set and reset by the SSBX and RSBX instructions, respectively.

$\overline{\text{HOLD}}$ is not treated as an interrupt. The hold is accepted while executing the IDLE1 instruction regardless of the HM values. The hold is not accepted while executing the IDLE2 or IDLE3 instructions regardless of the HM value. If $\overline{\text{HOLD}}$ is received, the CPU continues to execute the IDLE instruction even though the external buses and the control signals are placed in high impedance.

Figure 10–23 shows the timing for $\overline{\text{HOLD}}$ and $\overline{\text{HOLDA}}$. If $\overline{\text{HOLD}}$ meets the setup time before CLKOUT is low, a minimum of three machine cycles are needed before the buses and control signals go into high impedance. The $\overline{\text{HOLD}}$ is an external asynchronous input which is not latched. The external device must keep $\overline{\text{HOLD}}$ low. The external device can determine that the hold state has been entered when it receives a $\overline{\text{HOLDA}}$ signal from the ’54x.

If the ’54x is in the middle of a multicycle instruction, it finishes the instruction before entering the hold state. After the instruction is completed, the buses are placed into high impedance. This also applies to instructions that become multicycle because wait states are added.

After $\overline{\text{HOLD}}$ is deasserted, program execution resumes at the same instruction from which it was halted. $\overline{\text{HOLDA}}$ is removed synchronously with $\overline{\text{HOLD}}$, as shown in Figure 10–23. If the setup time is met, the processor requires two machine cycles (HM = 0) or three machine cycles (HM = 1) before the buses and control signals become valid.

10.6.1 Interrupts During Hold

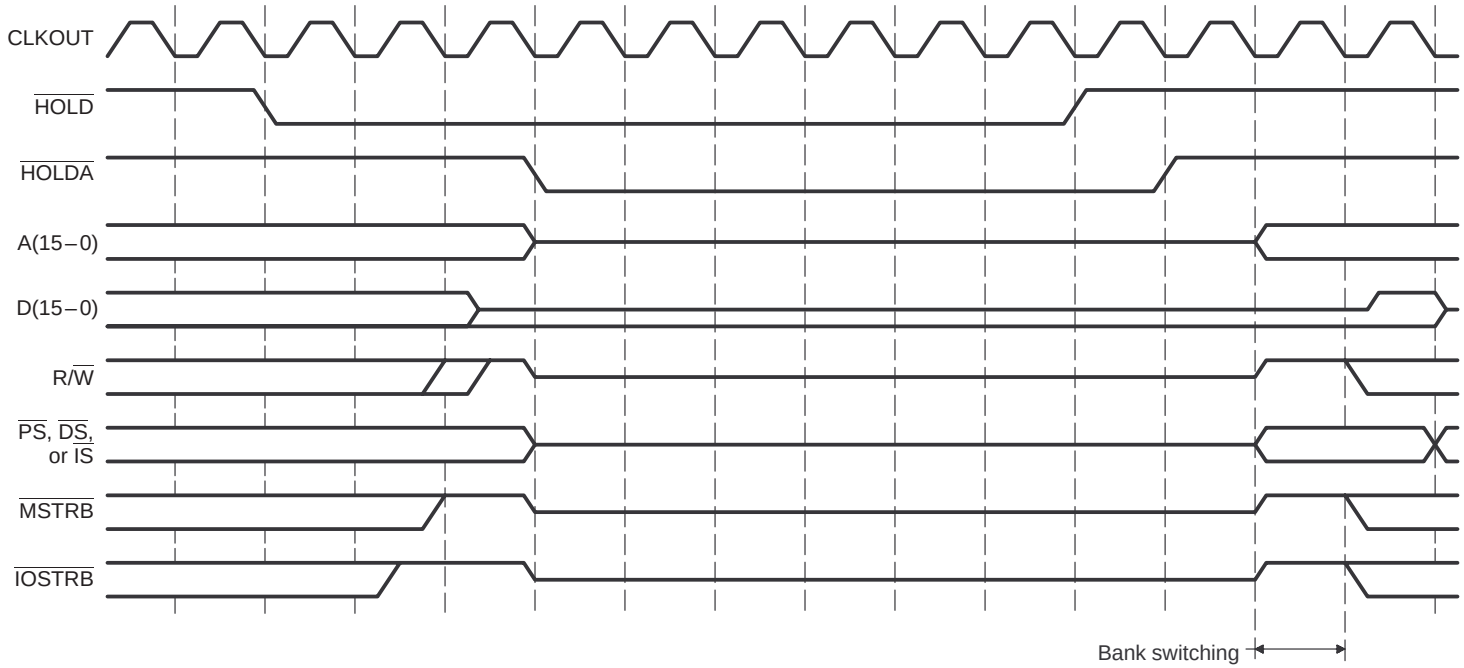
All interrupts are disabled while $\overline{\text{HOLD}}$ is active with HM = 1. If an interrupt is received during this period, the interrupt is latched and remains pending; therefore, $\overline{\text{HOLD}}$ does not affect any interrupt flags or registers. When HM = 0, interrupts function normally.

10.6.2 Hold and Reset

If $\overline{\text{HOLD}}$ is asserted while $\overline{\text{RS}}$ is active, normal reset operation occurs internally, but all buses and control lines remain or become high impedance and $\overline{\text{HOLDA}}$ is asserted, as shown in Figure 10–24 (a) and (b). However, if $\overline{\text{RS}}$ is asserted while $\overline{\text{HOLD}}$ and $\overline{\text{HOLDA}}$ are active, the CPU is reset and the data bus, address bus, and control signals remain in high impedance, as shown in Figure 10–24 (c) and (d).

If $\overline{\text{HOLD}}$ is deasserted while $\overline{\text{RS}}$ is active, $\overline{\text{HOLDA}}$ is deasserted normally in response, and the address, data, and control lines are driven according to the active reset state, as shown in Figure 10–24 (a) and (d). However, if $\overline{\text{RS}}$ is deasserted while $\overline{\text{HOLD}}$ and $\overline{\text{HOLDA}}$ are active, the operation of the device depends on the state of the MP/$\overline{\text{MC}}$ pin. If MP/$\overline{\text{MC}}$ is high, the CPU begins fetching the reset vector when the hold mode is exited. If MP/$\overline{\text{MC}}$ is low, the CPU fetches the reset vector internally and continues processing, unless it requires an external access before the hold mode is exited. Figure 10–24 (b) and (c) show examples of the case in which $\overline{\text{RS}}$ is deasserted while $\overline{\text{HOLD}}$ and $\overline{\text{HOLDA}}$ are active.

Figure 10–23. $\overline{HOLD}$ and $\overline{HOLDA}$ Minimum Timing for HM = 0



Notes: 1) The timing shows the hold mode when HM = 0. When HM = 1, another cycle is required before $\overline{HOLDA}$ becomes inactive.
2) The first cycle after releasing the hold mode is a cycle of bank switching.

Figure 10–24. $\overline{HOLD}$ and $\overline{RS}$ Interaction

(a)

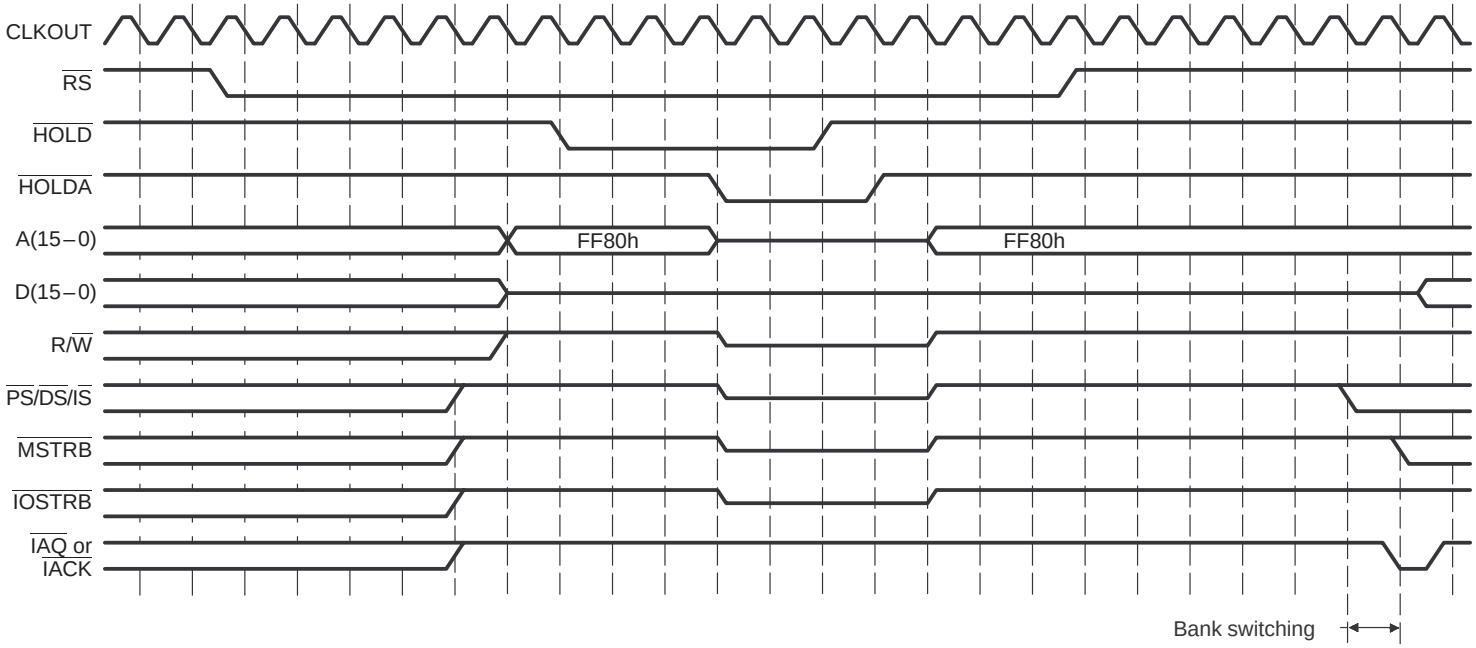


Figure 10–24. $\overline{HOLD}$ and $\overline{RS}$ Interaction (Continued)

(b)

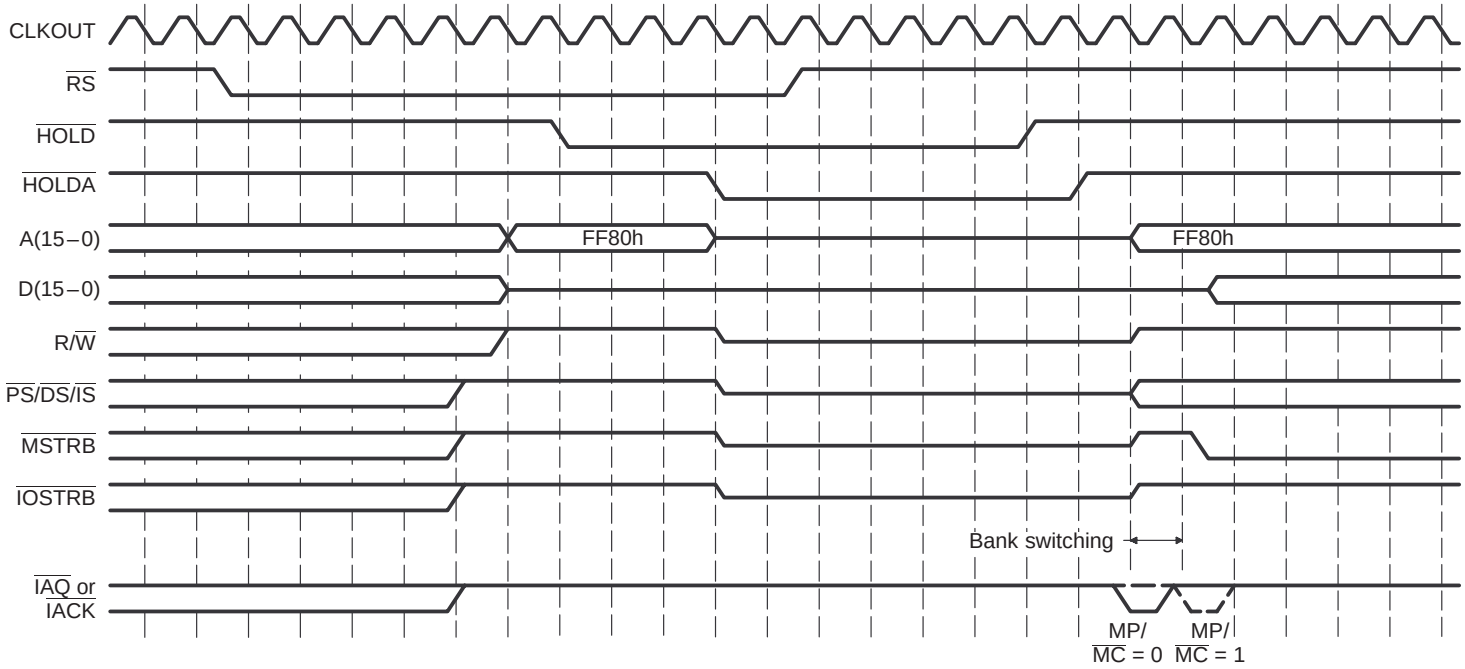


Figure 10–24. $\overline{HOLD}$ and $\overline{RS}$ Interaction (Continued)

(c)

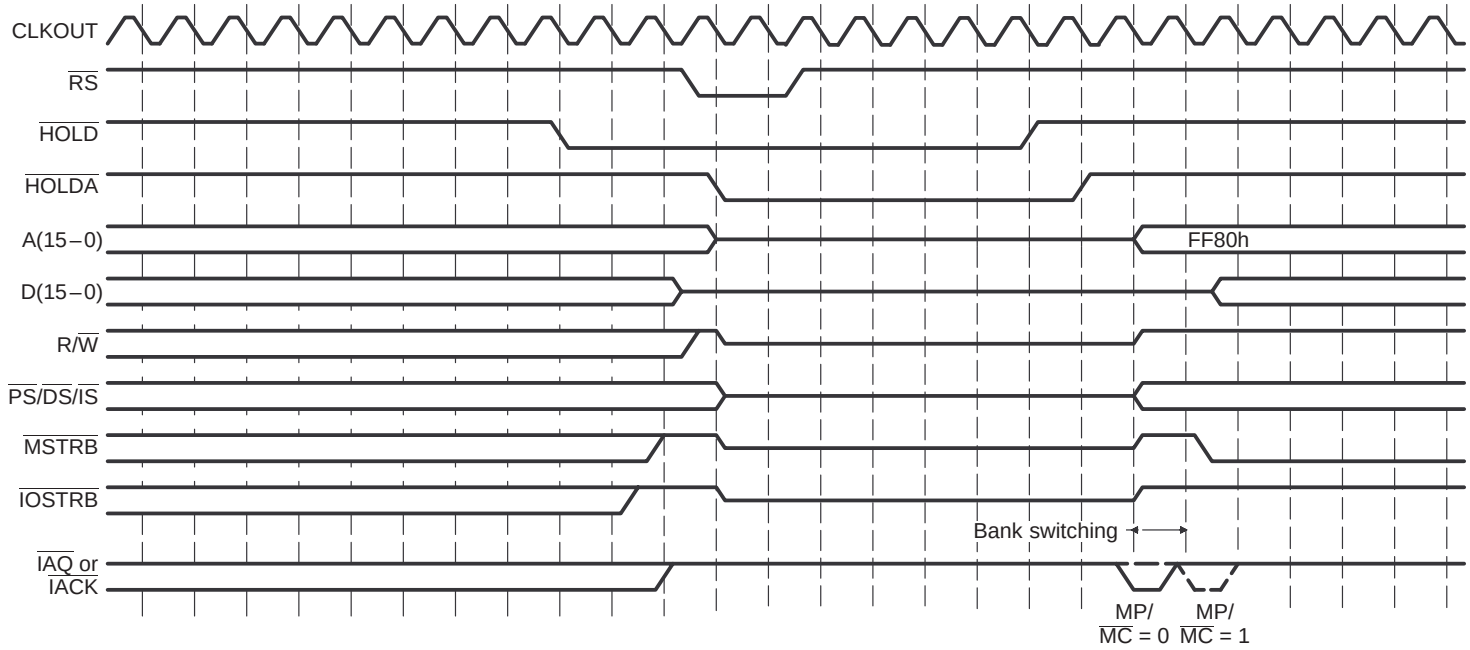
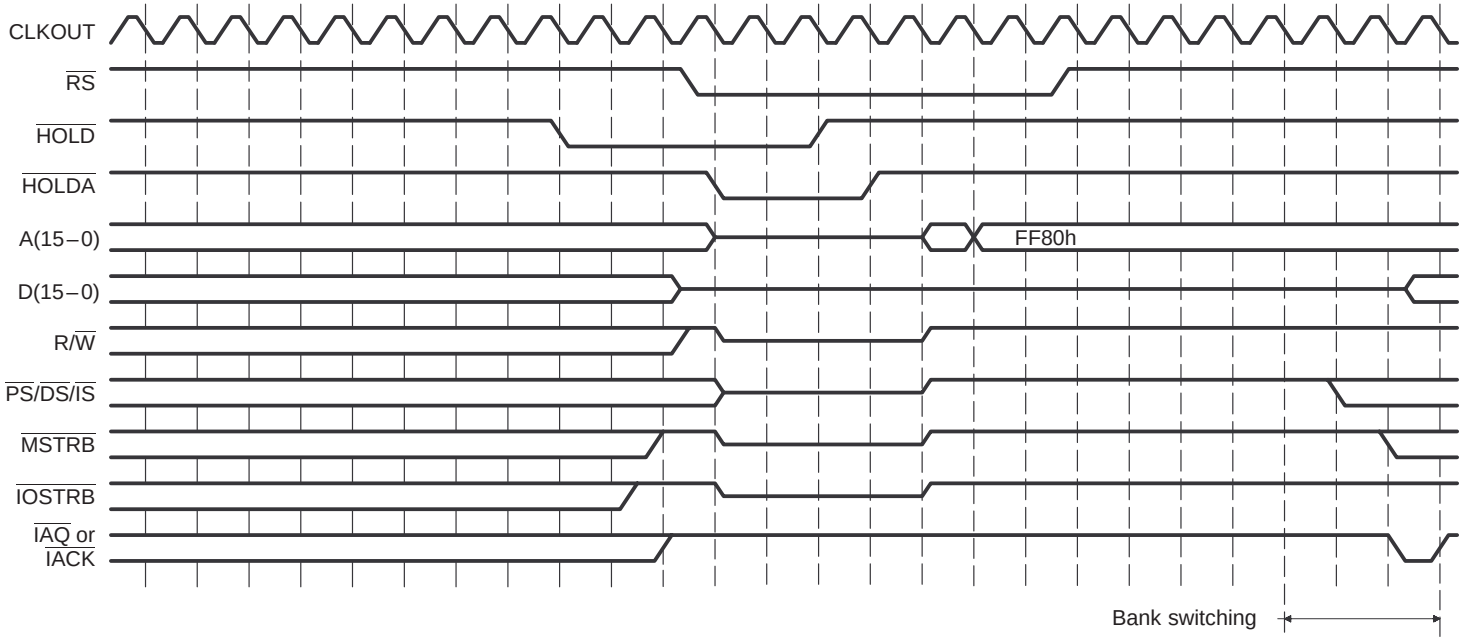


Figure 10–24. $\overline{HOLD}$ and $\overline{RS}$ Interaction (Continued)

(d)



Appendix A

CPU and Peripheral Registers

This appendix shows the bit fields of the ’54x CPU and peripheral registers. The following table defines terms used in identifying these register fields.

Term	Definition	Term	Definition
ARP	Auxiliary register pointer	OVA	Overflow flag A
ASM	Accumulator shift mode	OVB	Overflow flag B
AVIS	Address visibility mode	OVLY	RAM overlay
BH	Bus holder	OVM	Overflow mode
BMINT1,BMINT0	Buffer misalignment interrupt	PCM	Pulse code modulation mode
BNKCMP	Bank compare	PLLCOUNT	PLL counter value
BOB	Byte ordering bit	PLLDIV	PLL divider
BRAF	Block-repeat active flag	PLLMUL	PLL multiplier
BRE	Autobuffering receive enable	PLLNDIV	PLL clock generator select
BRINT1, BRINT0	BSP receive interrupt	PLLON/OFF	PLL on/off
BXE	Autobuffering transmit enable	PLLSTATUS	PLL status
BXINT1, BXINT0	BSP transmit interrupt	PSC	Timer prescaler counter
C	Carry	PS–DS	Program read–data read access
CH0–CH7	TDM channel 0–7	RA0–RA7	TDM receive address
CLKDV	Internal transmit clock division factor	Res, Resvd	Reserved
CLKOFF	CLOCKOUT off	RH	Receive buffer half received
CLKP	Clock polarity	RINT1, RINT0	Serial port receive interrupt
CMPT	Compatibility mode	RRDY	Receive ready
CPL	Compiler mode	$\overline{\text{RRST}}$	Receive reset
C16	Dual 16-bit/Double-precision arithmetic mode	RSRFULL	Receive shift register full

Term	Definition	Term	Definition
DLB	Digital loopback mode	SMOD	Shared-access mode
DP	Data page pointer	SMUL	Saturation on multiplication
DROM	Data ROM	SST	Saturation on store
DSPINT	DSP interrupt	SXM	Sign-extension mode
EXIO	External-bus-interface off	TA0–TA7	TDM transmit address
FE	Format extension	TC	Test/control flag
FIG	Frame ignore	TDDR	Timer divide-down ratio
FO	Format	TDM	TDM mode
FRCT	Fractional mode	TINT	Internal timer interrupt
FSM	Frame sync mode	TRB	Timer reload
FSP	Frame sync polarity	TRINT	TDM receive interrupt
HALTR	Autobuffering receive halt	TSS	Timer stop status
HALTX	Autobuffering transmit halt	TXINT	TDM transmit interrupt
HINT	’54x-to-Host interrupt	TXM	Transmit mode
HM	Hold mode	XF	External flag (XF) status
HPINT	HPI interrupt	XH	Transmit buffer half transmitted
INTM	Interrupt mode	XINT1, XINT0	Serial port transmit interrupt
INT0–INT3	External user interrupt #0–3	XPA	Extended program address control
IPTR	Interrupt vector pointer	XRDY	Transmit ready
MCM	Clock mode	$\overline{\text{XRST}}$	Transmitter reset
MP/$\overline{\text{MC}}$	Microprocessor/microcontroller	$\overline{\text{XSREMPTY}}$	Transmit shift register empty

Figure A–1. Bank-Switching Control Register (BSCR) Diagram

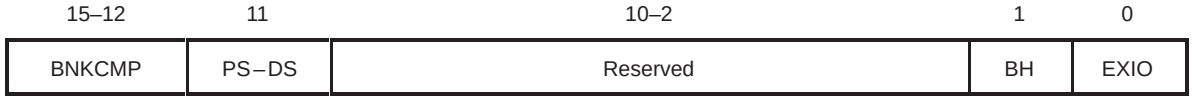


Figure A–2. BSP Control Extension Register (BSPCE) Diagram

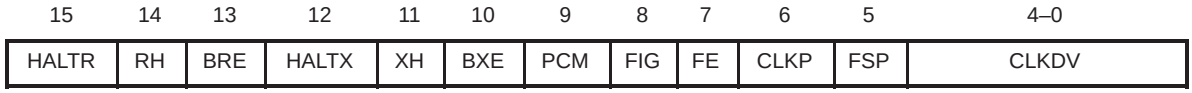


Figure A–3. Clock Mode Register (CLKMD) Diagram (LP devices only)

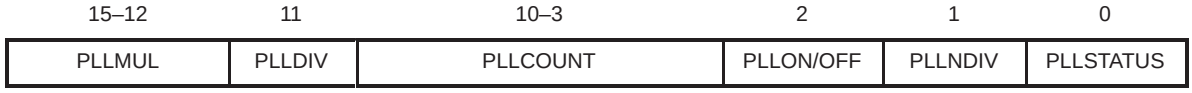
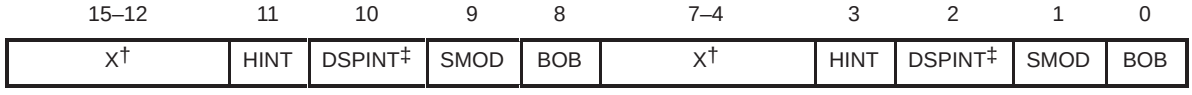


Figure A–4. HPI Control Register (HPIC) Diagram



† Unknown value is read, any value can be written.
‡ 0 is read.

Figure A–5. Interrupt Flag Register (IFR) Diagram

(a) ’541 IFR

15–12	11	10	9	8	7	6	5	4	3	2	1	0
Resvd	Resvd	Resvd	Resvd	INT3	XINT1	RINT1	XINT0	RINT0	TINT	INT2	INT1	INT0

(b) ’542 IFR

15–12	11	10	9	8	7	6	5	4	3	2	1	0
Resvd	Resvd	Resvd	HPINT	INT3	TXINT	TRINT	BXINT0	BRINT0	TINT	INT2	INT1	INT0

(c) ’543 IFR

15–12	11	10	9	8	7	6	5	4	3	2	1	0
Resvd	Resvd	Resvd	Resvd	INT3	TXINT	TRINT	BXINT0	BRINT0	TINT	INT2	INT1	INT0

(d) ’545 IFR

15–12	11	10	9	8	7	6	5	4	3	2	1	0
Resvd	Resvd	Resvd	HPINT	INT3	XINT1	RINT1	BXINT0	BRINT0	TINT	INT2	INT1	INT0

(e) ’546 IFR

15–12	11	10	9	8	7	6	5	4	3	2	1	0
Resvd	Resvd	Resvd	Resvd	INT3	XINT1	RINT1	BXINT0	BRINT0	TINT	INT2	INT1	INT0

(f) ’548 IFR

15–12	11	10	9	8	7	6	5	4	3	2	1	0
Resvd	BXINT1	BRINT1	HPINT	INT3	TXINT	TRINT	BXINT0	BRINT0	TINT	INT2	INT1	INT0

(g) ’549 IFR

15–14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Resvd	BMINT1	BMINT0	BXINT1	BRINT1	HPINT	INT3	TXINT	TRINT	BXINT0	BRINT0	TINT	INT2	INT1	INT0

Figure A–6. Interrupt Mask Register (IMR) Diagram

(a) ’541 IMR

15–12	11	10	9	8	7	6	5	4	3	2	1	0
Resvd	Resvd	Resvd	Resvd	INT3	XINT1	RINT1	XINT0	RINT0	TINT	INT2	INT1	INT0

(b) ’542 IMR

15–12	11	10	9	8	7	6	5	4	3	2	1	0
Resvd	Resvd	Resvd	HPINT	INT3	TXINT	TRINT	BXINT0	BRINT0	TINT	INT2	INT1	INT0

(c) ’543 IMR

15–12	11	10	9	8	7	6	5	4	3	2	1	0
Resvd	Resvd	Resvd	Resvd	INT3	TXINT	TRINT	BXINT0	BRINT0	TINT	INT2	INT1	INT0

(d) ’545 IMR

15–12	11	10	9	8	7	6	5	4	3	2	1	0
Resvd	Resvd	Resvd	HPINT	INT3	XINT1	RINT1	BXINT0	BRINT0	TINT	INT2	INT1	INT0

(e) ’546 IMR

15–12	11	10	9	8	7	6	5	4	3	2	1	0
Resvd	Resvd	Resvd	Resvd	INT3	XINT1	RINT1	BXINT0	BRINT0	TINT	INT2	INT1	INT0

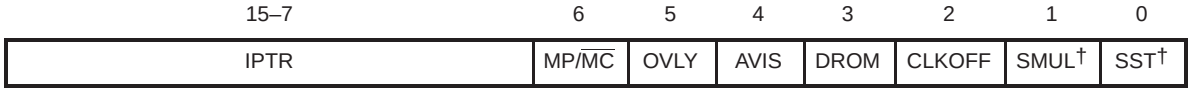
(f) ’548 IMR

15–12	11	10	9	8	7	6	5	4	3	2	1	0
Resvd	BXINT1	BRINT1	HPINT	INT3	TXINT	TRINT	BXINT0	BRINT0	TINT	INT2	INT1	INT0

(g) ’549 IMR

15–14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Resvd	BMINT1	BMINT0	BXINT1	BRINT1	HPINT	INT3	TXINT	TRINT	BXINT0	BRINT0	TINT	INT2	INT1	INT0

Figure A–7. Processor Mode Status Register (PMST) Diagram



† Only on the LP device; reserved bits on all other devices

Figure A–8. Serial Port Control Register (SPC) Diagram

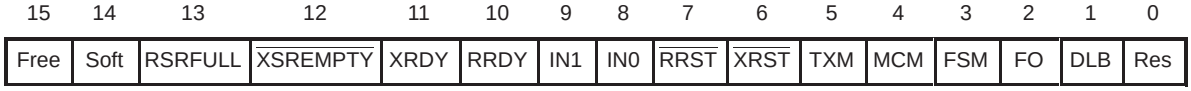
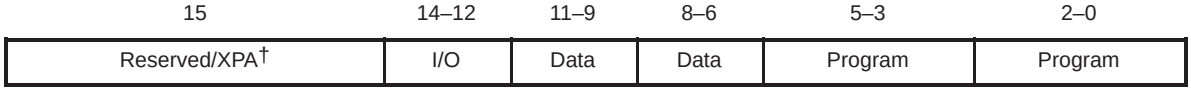


Figure A–9. Software Wait-State Register (SWWSR) Diagram



† XPA bit on ’548 and ’549 only

Figure A–10. Extended Software Wait-State Register (XSWWR) Diagram

Figure A–11. Status Register 0 (ST0) Diagram

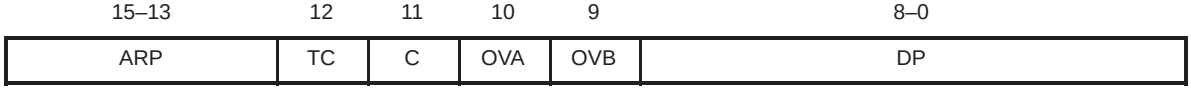


Figure A–12. Status Register 1 (ST1) Diagram

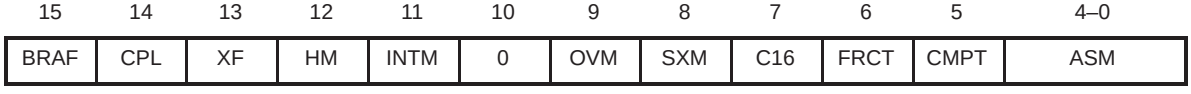
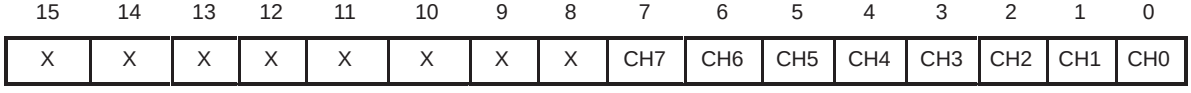


Figure A–13. TDM Channel Select Register (TCSR) Diagram



Note: X=Don’t care.

Figure A–14. TDM Receive Address Register (TRAD) Diagram

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
X	X	X2	X1	X0	S2	S1	S0	A7	A6	A5	A4	A3	A2	A1	A0

Note: X=Don't care.

Figure A–15. TDM Receive/Transmit Address Register (TRTA) Diagram

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
TA7	TA6	TA5	TA4	TA3	TA2	TA1	TA0	RA7	RA6	RA5	RA4	RA3	RA2	RA1	RA0

Figure A–16. TDM Serial Port Control Register (TSPC) Diagram

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Free	Soft	X	X	XRDY	RRDY	IN1	IN0	$\overline{\text{RRST}}$	$\overline{\text{XRST}}$	TXM	MCM	X	0	0	TDM

Note: X=Don't care.

Figure A–17. Timer Control Register (TCR) Diagram

15–12	11	10	9–6	5	4	3–0
Reserved	Soft	Free	PSC	TRB	TSS	TDDR

Appendix B

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 book, refers to TI scan-based emulation, which is based on the IEEE 1149.1 standard.

For more information concerning the IEEE 1149.1 standard, contact IEEE Customer Service:

Address: IEEE Customer Service
445 Hoes Lane, PO Box 1331
Piscataway, NJ 08855-1331

Phone: (800) 678–IEEE in the US and Canada
(908) 981–1393 outside the US and Canada

FAX: (908) 981–9667 Telex: 833233

Topic **Page**

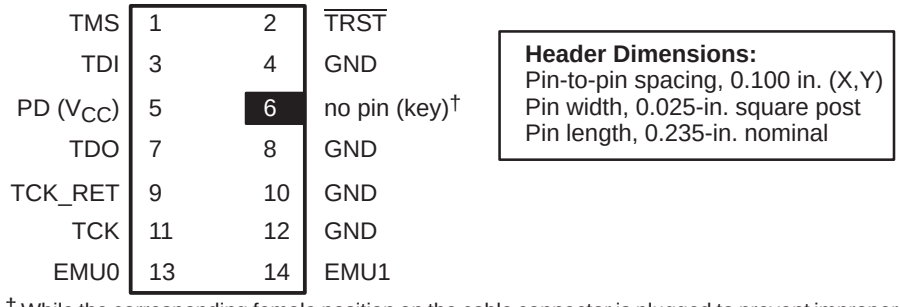
B.1 Designing Your Target System's Emulator Connector (14-Pin Header)

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 B–1. Table B–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 B–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 B–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. May be a buffered or unbuffered version of TCK.	I	O
TDI	Test data input	O	I
TDO	Test data output	I	O
TMS	Test mode select	O	I
$\overline{\text{TRST}}$‡	Test reset	O	I

† I = input; O = output

‡ Do not use pullup resistors on $\overline{\text{TRST}}$: it has an internal pulldown device. In a low-noise environment, $\overline{\text{TRST}}$ can be left floating. In a high-noise environment, an additional pulldown resistor may be needed. (The size of this resistor should be based on electrical current considerations.)

B.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.

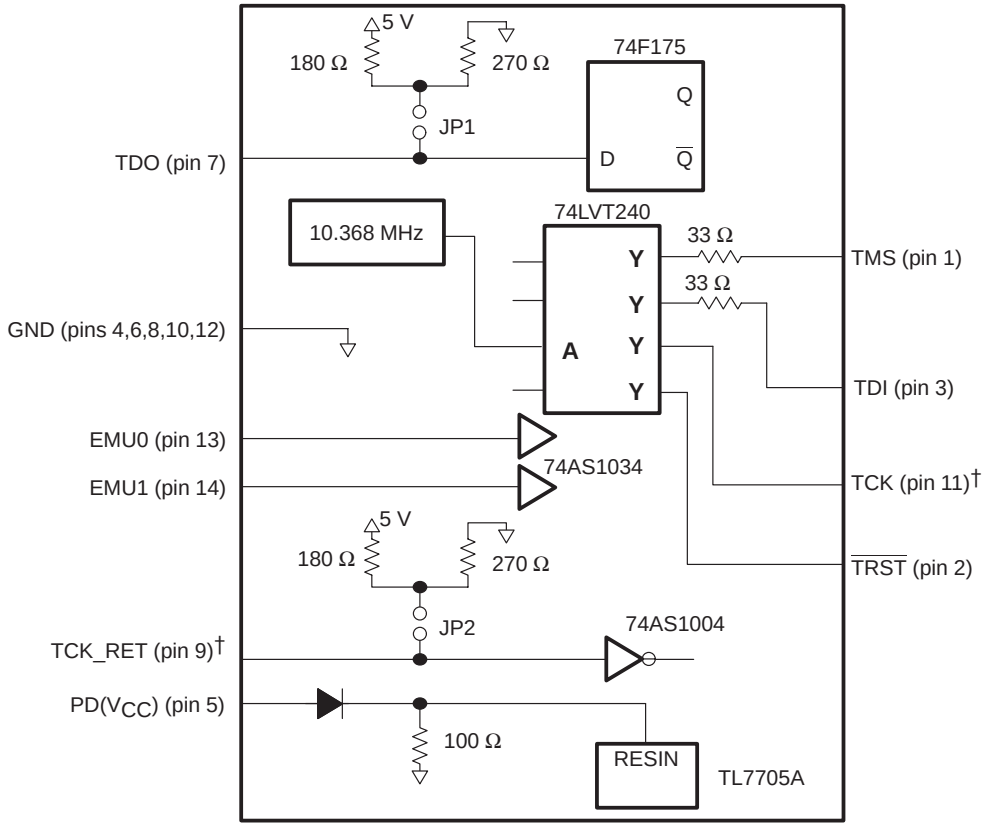
The IEEE 1149.1 specification does not provide rules for bus master (emulator) devices. Instead, it states that the device expects a bus master to provide bus slave compatible timings. The XDS510 provides timings that meet the bus slave rules.

B.3 Emulator Cable Pod

Figure B–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 B–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.

B.4 Emulator Cable Pod Signal Timing

Figure B–3 shows the signal timings for the emulator cable pod. Table B–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 B–3. Emulator Cable Pod Timings

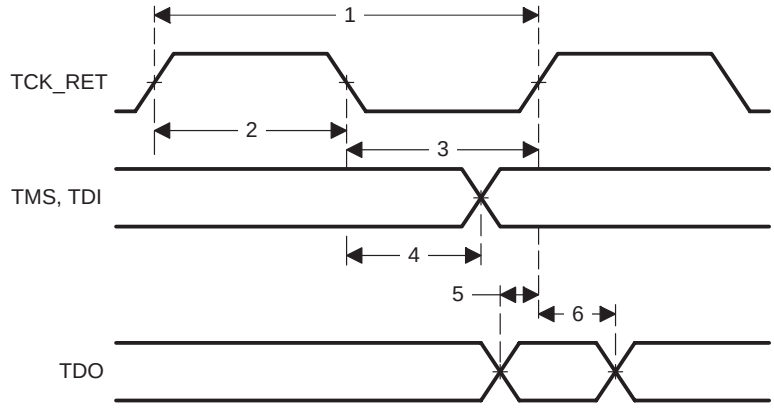


Table B–2. Emulator Cable Pod Timing Parameters

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

B.5 Emulation Timing Calculations

Example B–1 and Example B–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 B–2 on page B-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 B–1. Key Timing for a Single-Processor System Without Buffers

$$t_{pd\,(TCK\_RET\text{-}TMS/TDI)} = \frac{\left[t_{d\,(TMSmax)} + t_{su\,(TTMS)}\right]}{t_{TCKfactor}}$$

$$= \frac{(20\text{ ns} + 10\text{ ns})}{0.4}$$

$$= 75\text{ ns, or } 13.3\text{ MHz}$$

$$t_{pd\,(TCK\_RET–TDO)} = \frac{\left[t_{d\,(TTDO)} + t_{su\,(TDOmin)}\right]}{t_{TCKfactor}}$$

$$= \frac{(15\text{ ns} + 3\text{ ns})}{0.4}$$

$$= 45\text{ ns, or } 22.2\text{ MHz}$$

In this case, because the TCK_RET-to-TMS/TDI path requires more time to complete, it is the limiting factor.

Example B–2. Key Timing for a Single- or Multiple-Processor System With Buffered Input and Output

$$t_{pd\,(TCK\_RET\text{-}TMS/TDI)} = \frac{\left[t_{d\,(TMSmax)} + t_{su\,(TTMS)} + t_{bufskew}\right]}{t_{TCKfactor}}$$

$$= \frac{(20\text{ ns} + 10\text{ ns} + 1.35\text{ ns})}{0.4}$$

$$= 78.4\text{ ns, or } 12.7\text{ MHz}$$

$$t_{pd\,(TCK\_RET–TDO)} = \frac{\left[t_{d\,(TTDO)} + t_{su\,(TDOmin)} + t_{d\,(bufmax)}\right]}{t_{TCKfactor}}$$

$$= \frac{(15\text{ ns} + 3\text{ ns} + 10\text{ ns})}{0.4}$$

$$= 70\text{ ns, or } 14.3\text{ MHz}$$

In 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:

$$
\begin{aligned}
t_r &= 5(R_{pullup} \times N_{devices} \times C_{load\_per\_device}) \\
&= 5(4.7\ k\Omega \times 16 \times 15\ pF) \\
&= 5(4.7 \times 10^3\ \Omega \times 16 \times 15 = no\ ^{-12}\ F) \\
&= 5(1128 \times 10^{-9}\ ) \\
&= 5.64\ \mu s
\end{aligned}
$$

B.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.

B.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 B–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 B–4. Emulator Connections Without Signal Buffering

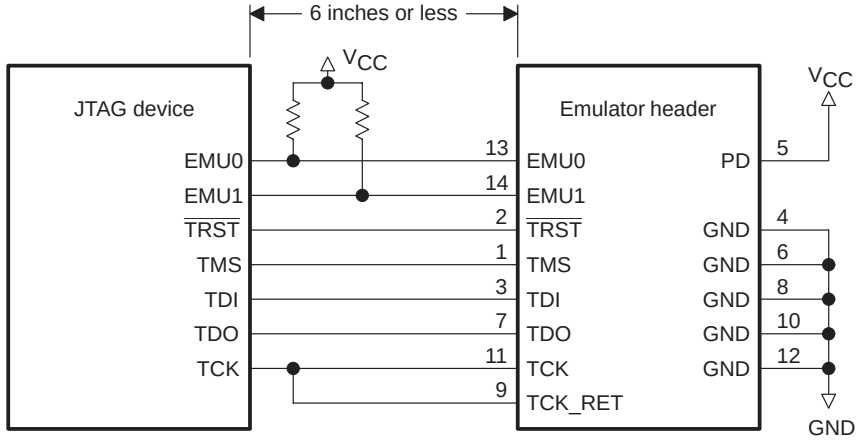
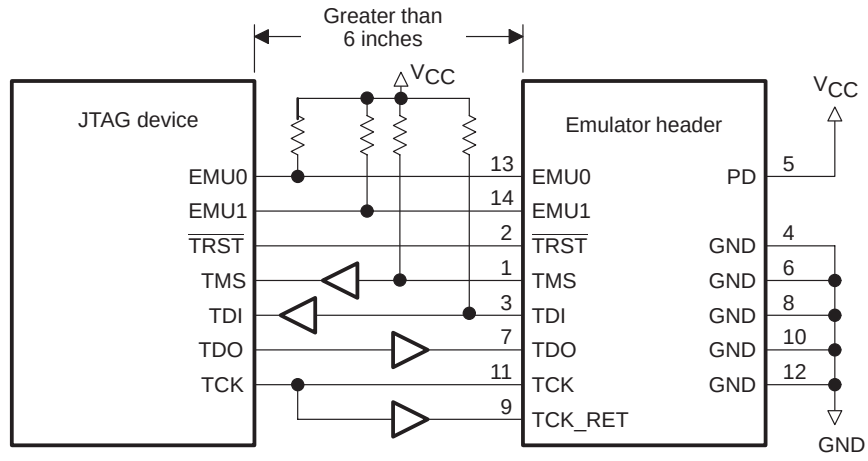


Figure B–5 shows the connections necessary for buffered transmission signals. The distance between the emulation header and the processor is greater than 6 inches. Emulation signals TMS, TDI, TDO, and TCK_RET are buffered through the same device package.

Figure B–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.

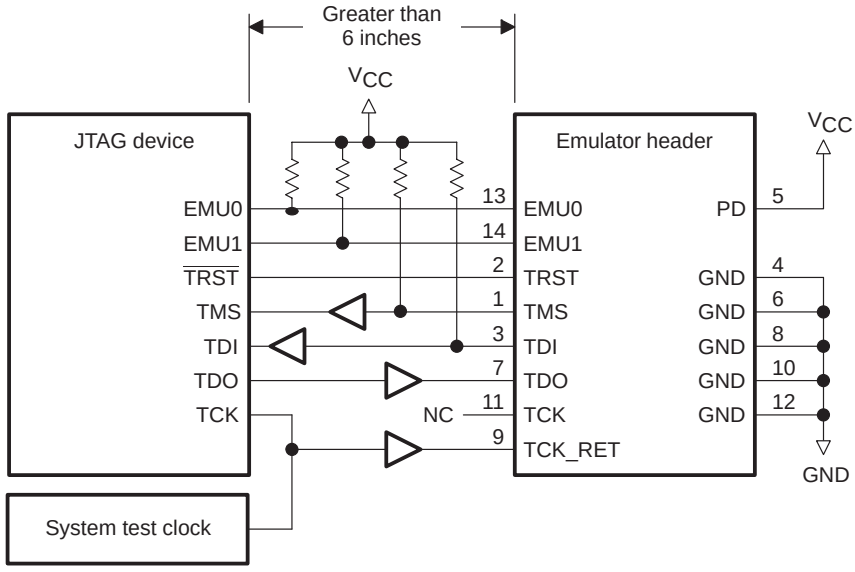
To have high-quality signals (especially the processor TCK and the emulator TCK_RET signals), you may have to employ special care when routing the printed wiring board trace. You also may have to use termination resistors to match the trace impedance. The emulator pod provides optional internal parallel 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.

B.6.2 Using a Target-System Clock

Figure B–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 B–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:

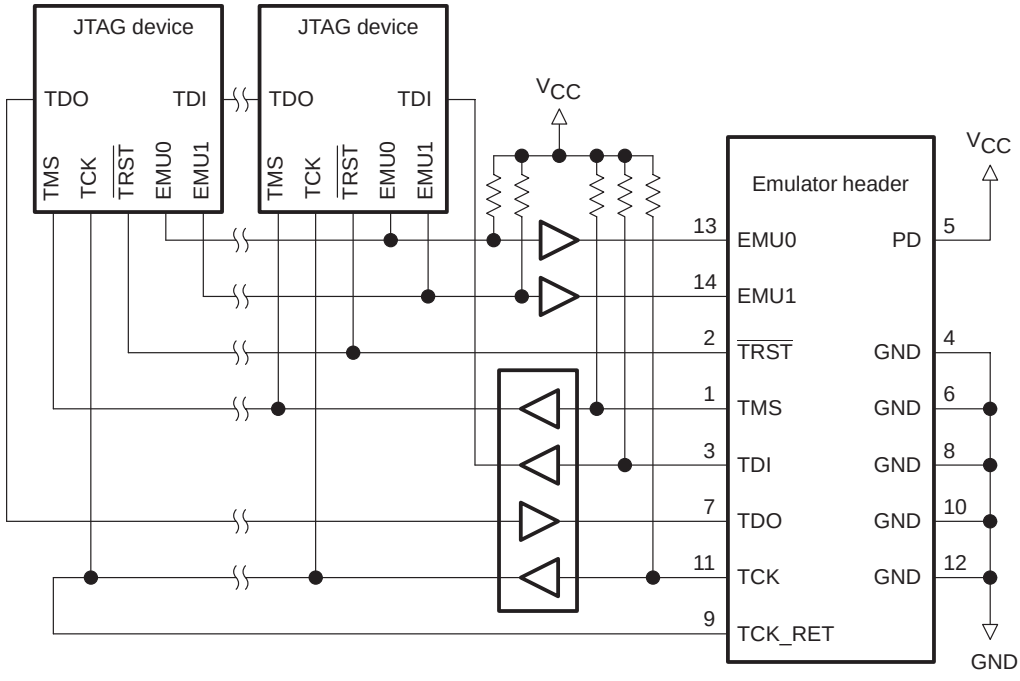
- The emulator provides only a single 10.368-MHz test clock. If you allow the target system to generate your test clock, you can set the frequency to match your system requirements.
- In some cases, you may have other devices in your system that require a test clock when the emulator is not connected. The system test clock also serves this purpose.

B.6.3 Configuring Multiple Processors

Figure B–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.

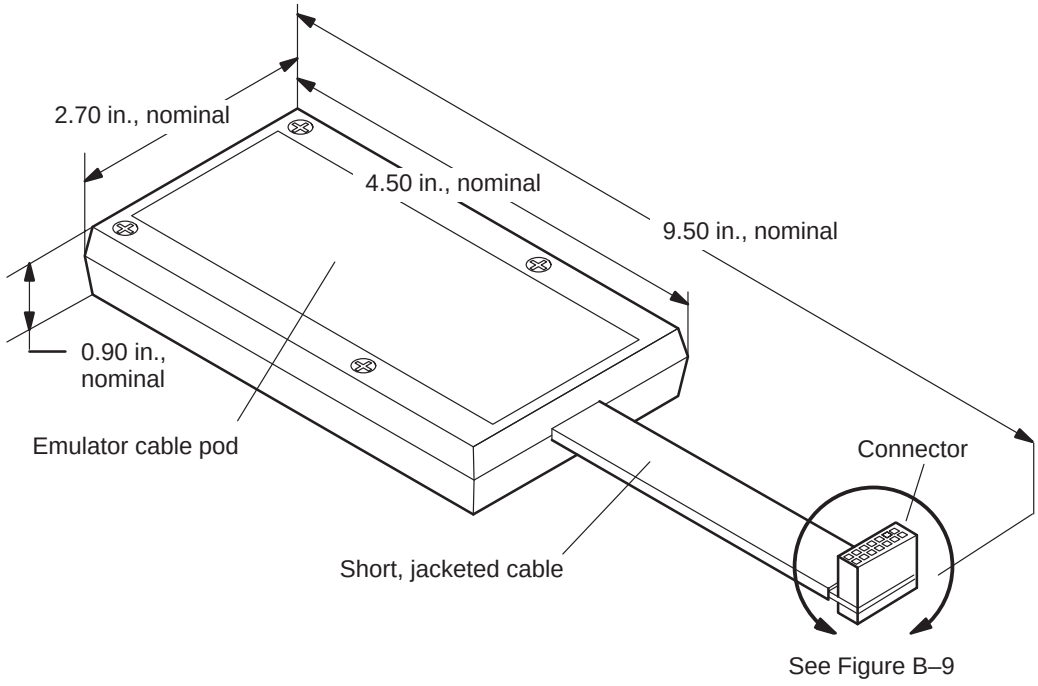
Figure B–7. Multiprocessor Connections



B.7 Physical Dimensions for the 14-Pin Emulator Connector

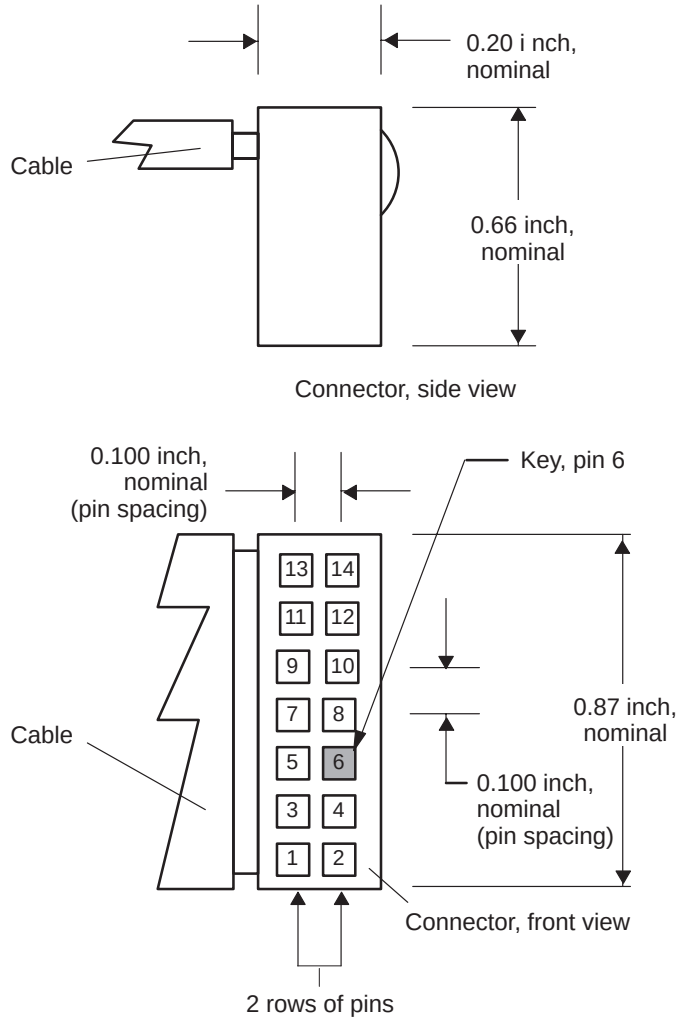
The JTAG emulator target cable consists of a 3-foot section of jacketed cable that connects to the emulator, an active cable pod, and a short section of jacketed cable that connects to the target system. The overall cable length is approximately 3 feet 10 inches. Figure B–8 and Figure B–9 (page B-15) 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 B–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 B–9. 14-Pin Connector Dimensions



B.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.

B.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.

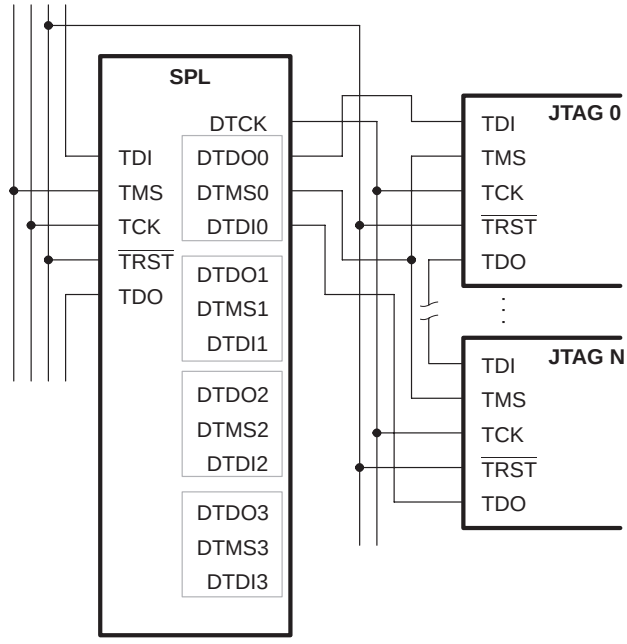
A system of multiple, secondary JTAG scan paths has better fault tolerance and isolation than a single scan path. Since an SPL has the capability of adding all secondary scan paths to the main scan path simultaneously, it can support global emulation operations, such as starting or stopping a selected 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 B–10 shows how to connect a secondary scan path to an SPL.

Figure B–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.

B.8.2 Emulation Timing Calculations for a Scan Path Linker (SPL)

Example B–3 and Example B–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 B–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 B–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.

B.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:

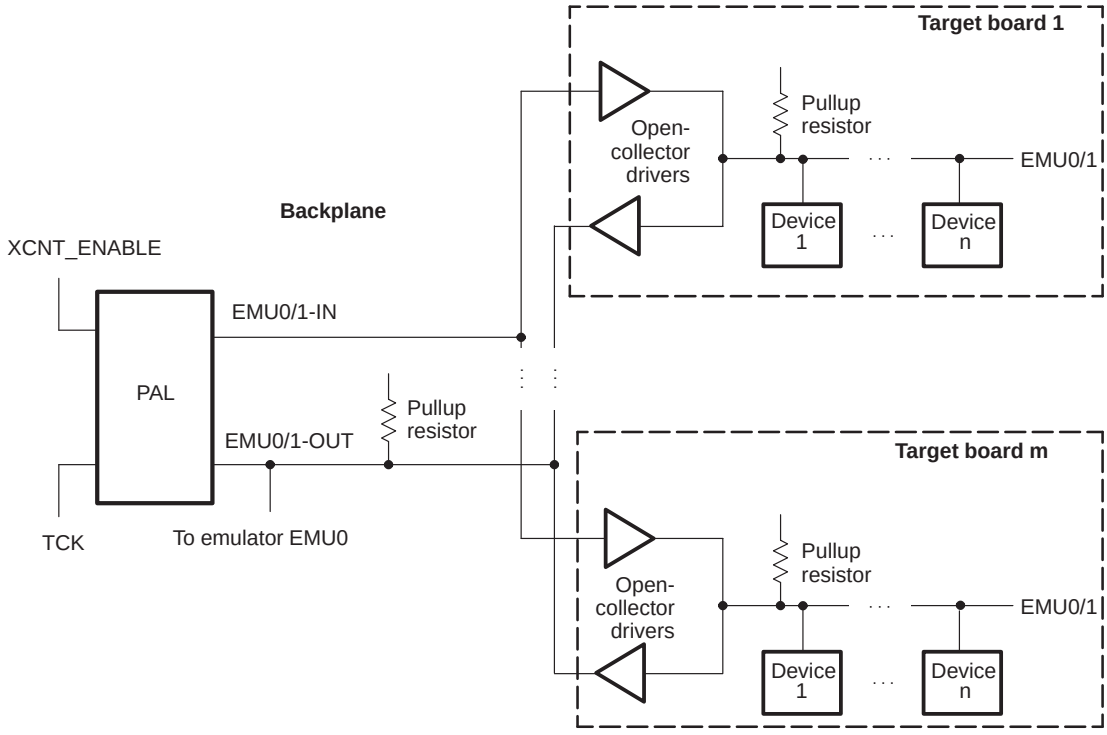
- Signal Event. The EMU0/1 pins can be configured via software to signal internal events. In this mode, driving one of these pins low can cause devices to signal such events. To enable this operation, the EMU0/1 pins function as open-collector sources. External devices such as logic analyzers can also 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 B–11 shows an example configuration that allows any processor in the system to stop any other processor in the system. Do not tie the EMU0/1 pins of more than 16 processors together in a single group without using buffers. Buffers provide the crisp signals that are required during a RUNB (run benchmark) debugger command or when the external analysis counter feature is used.

Figure B–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 during the RUNB command or when the external counter selected from the debugger analysis menu is used.

These seven important points apply to the circuitry shown in Figure B–11 and the timing shown in Figure B–12:

- ❏ Open-collector drivers isolate each board. The EMU0/1 pins are tied together on each board.
- ❏ At the board edge, the EMU0/1 signals are split to provide both input and output connections. This is required to prevent the open-collector drivers from acting as latches that can be set only once.
- ❏ The EMU0/1 signals are bused down the backplane. Pullup resistors must be installed as required.

- The bused EMU0/1 signals go into a programmable logic array device PAL® whose function is to generate a low pulse on the EMU0/1-IN signal when a low level is detected on the EMU0/1-OUT signal. This pulse must be longer than one TCK period to affect the devices but less than 10 μs to avoid possible conflicts or retriggering once the emulation software clears the device's pins.
- During a RUNB debugger command or other external analysis count, the EMU0/1 pins on the target device become totem-pole outputs. The EMU1 pin is a ripple carry-out of the internal counter. EMU0 becomes a *processor-halted* signal. During a RUNB or other external analysis count, the EMU0/1-IN signal to all boards must remain in the high (disabled) state. You must provide some type of external input (XCNT_ENABLE) to the PAL® to disable the PAL® from driving EMU0/1-IN to a low state.
- If you use sources other than TI processors (such as logic analyzers) to drive EMU0/1, their signal lines must be isolated by open-collector drivers and be inactive during RUNB and other external analysis counts.
- You must connect the EMU0/1-OUT signals to the emulation header or directly to a test bus controller.

Figure B–12. Suggested Timings for the EMU0 and EMU1 Signals

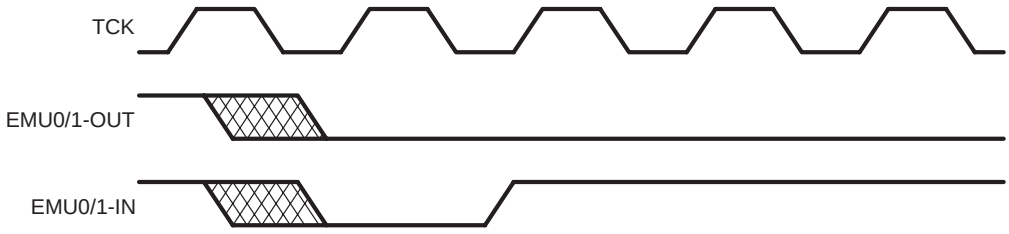
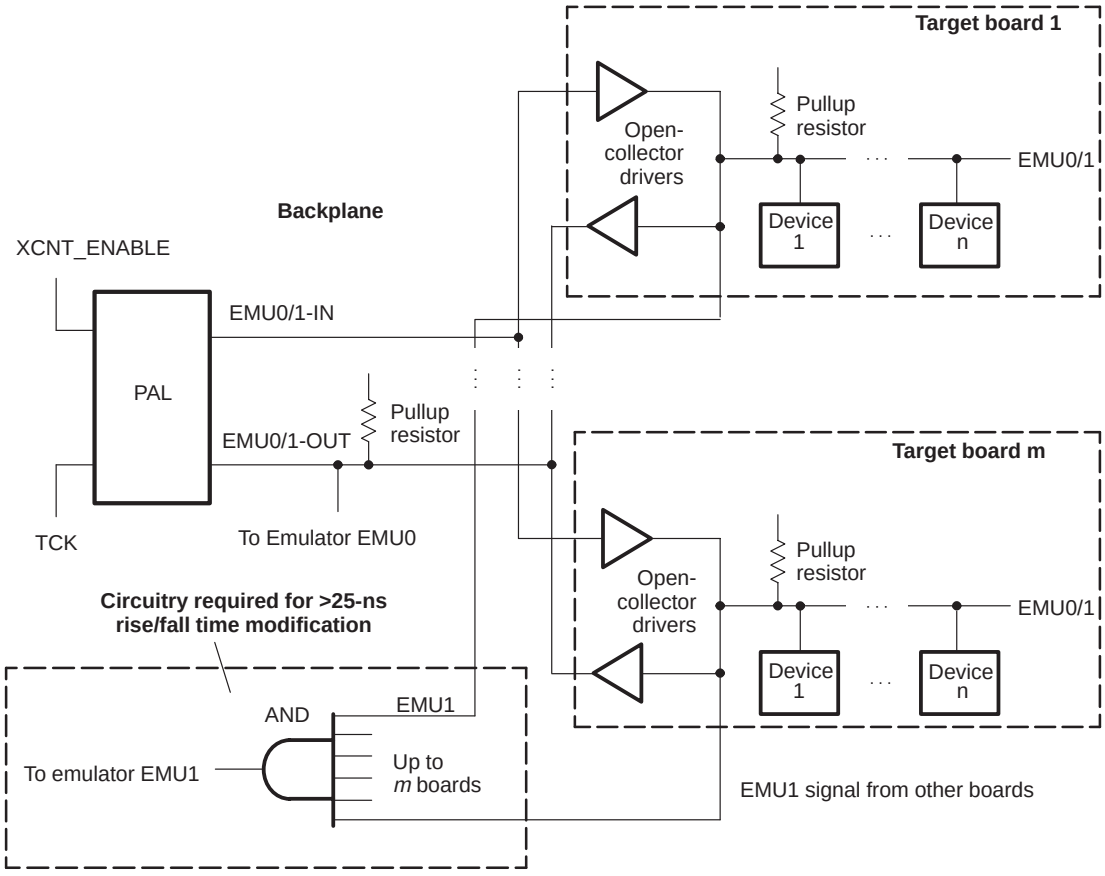


Figure B–13. EMU0/1 Configuration With Additional AND Gate to Meet Timing Requirements of Greater Than 25 ns

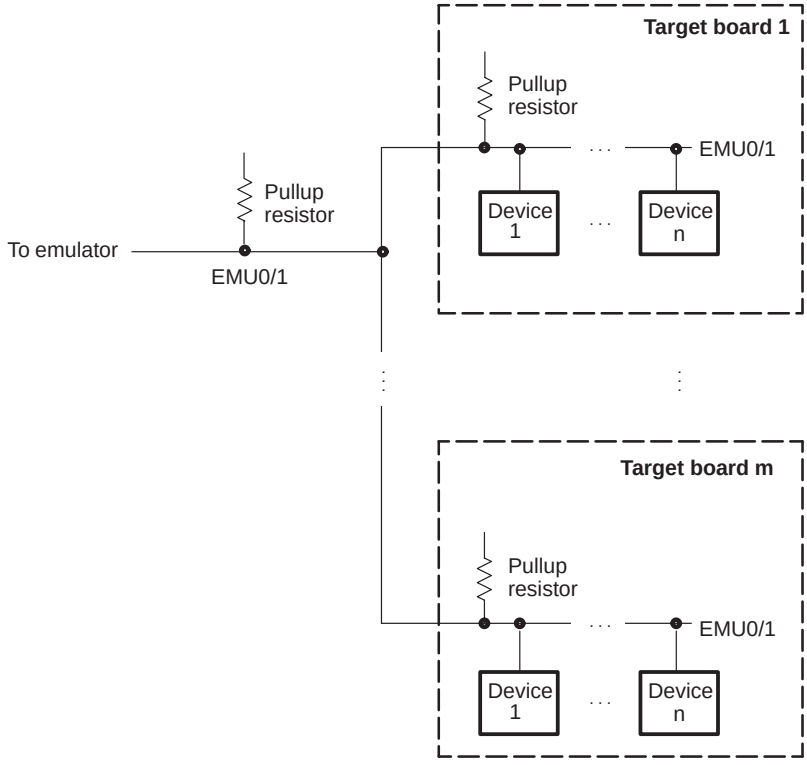


Notes: 1) The low time on EMU0/1-IN should be at least one TCK cycle and less than 10 μs. Software will set the EMU0/1-OUT port to a high state.

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 during the RUNB command or when the external counter selected from the debugger analysis menu is used.

You do not need to have devices on one target board stop devices on another target board using the EMU0/1 signals (see the circuit in Figure B–14). In this configuration, the global-stop capability is lost. It is important not to overload EMU0/1 with more than 16 devices.

Figure B–14. EMU0/1 Configuration Without Global Stop

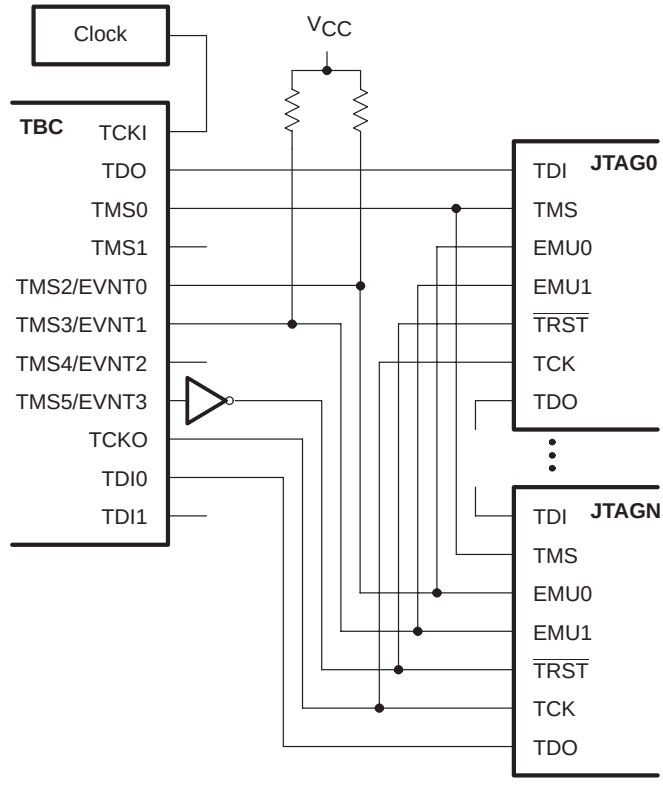


Note: The open-collector driver and pullup resistor on EMU1 must be able to provide rise/fall times of less than 25 ns. Rise times of more than 25 ns can cause the emulator to detect false edges during the RUNB command or when the external counter selected from the debugger analysis menu is used. If this condition cannot be met, then the EMU0/1 signals from the individual boards must be ANDed together (as shown in Figure B–14) to produce an EMU0/1 signal for the emulator.

B.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 B–15 shows the scan path connections of *n* devices to the TBC.

Figure B–15. TBC Emulation Connections for n JTAG Scan Paths



In the system design shown in Figure B–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 C

Development Support and Part Order Information

This appendix provides development support information, device part numbers, and support tool ordering information for the ’54x.

Each ’54x support product is described in the *TMS320 DSP Development Support Reference Guide*. In addition, more than 100 third-party developers offer products that support the TI TMS320 family. For more information, refer to the *TMS320 Third-Party Support Reference Guide*.

For information on pricing and availability, contact the nearest TI Field Sales Office or authorized distributor. See the list at the back of this book.

Topic **Page**

C.1 Development Support

This section describes the development support provided by Texas Instruments.

C.1.1 Development Tools

TI offers an extensive line of development tools for the ’54x generation of DSPs, including tools to evaluate the performance of the processors, generate code, develop algorithm implementations, and fully integrate and debug software and hardware modules.

Code Generation Tools

- The optimizing ANSI C compiler translates ANSI C language directly into highly optimized assembly code. You can then assemble and link this code with the TI assembler/linker, which is shipped with the compiler. This product is currently available for PCs (DOS, DOS extended memory, OS/2), HP workstations, and SPARC workstations. See the *TMS320C54x Optimizing C Compiler User’s Guide* for detailed information about this tool.
- The assembler/linker converts source mnemonics to executable object code. This product is currently available for PCs (DOS, DOS extended memory, OS/2). The ’54x assembler for HP and SPARC workstations is available only as part of the optimizing ’54x compiler. See the *TMS320C54x Assembly Language Tools User’s Guide* for detailed information about available assembly-language tools.

System Integration and Debug Tools

- The simulator simulates (via software) the operation of the ’54x and can be used in C and assembly software development. This product is currently available for PCs (DOS, Windows), HP workstations, and SPARC workstations. See the *TMS320C54x C Source Debugger User’s Guide* for detailed information about the debugger.
- The XDS510 emulator performs full-speed in-circuit emulation with the ’54x, providing access to all registers as well as to internal and external memory of the device. It can be used in C and assembly software development and has the capability to debug multiple processors. This product is currently available for PCs (DOS, Windows, OS/2), HP workstations, and SPARC workstations. This product includes the emulator board (emulator box, power supply, and SCSI connector cables in the HP and SPARC versions), the ’54x C source debugger and the JTAG cable.

Because the ’C2xx, ’C3x, ’C4x, and ’C5x XDS510 emulators also come with the same emulator board (or box) as the ’54x, you can buy the ’54x C Source Debugger Software as a separate product called the ’54x C Source Debugger Conversion Software. This enables you to debug ’54x applications with a previously purchased emulator board. The emulator cable that comes with the ’C3x XDS510 emulator cannot be used with the ’54x. You need the JTAG emulation conversion cable (see Section C.2) instead. The emulator cable that comes with the ’C5x XDS510 emulator can be used with the ’54x without any restriction. See the *TMS320C54x C Source Debugger User’s Guide*) for detailed information about the ’54x emulator.

- The TMS320C54x evaluation module (EVM) is a PC/AT plug-in card that lets you evaluate certain characteristics of the ’54x digital signal processor to see if it meets your application requirements. The ’54x EVM carries a ’541 DSP on board to allow full-speed verification of ’54x code. The EVM has 5K bytes of on-chip program/data RAM, 28K bytes of on-chip ROM, two serial ports, a timer, access to 64K bytes each of external program and data RAM, and an external analog interface for evaluation of the ’54x family of devices for applications. See the *TMS320C54x Evaluation Module Technical Reference* for detailed information about the ’54x EVM.

C.1.2 Third-Party Support

The TMS320 family is supported by products and services from more than 100 independent third-party vendors and consultants. These support products take various forms (both as software and hardware), from cross-assemblers, simulators, and DSP utility packages to logic analyzers and emulators. The expertise of those involved in support services ranges from speech encoding and vector quantization to software/hardware design and system analysis.

To ask about third-party services, products, applications, and algorithm development packages, contact the third party directly. Refer to the *TMS320 Third-Party Support Reference Guide* for addresses and phone numbers.

C.1.3 Technical Training Organization (TTO) TMS320 Workshops

'54x DSP Design Workshop. This workshop is tailored for hardware and software design engineers and decision-makers who will be designing and utilizing the '54x generation of DSP devices. Hands-on exercises throughout the course give participants a rapid start in developing '54x design skills. Microprocessor/assembly language experience is required. Experience with digital design techniques and C language programming experience is desirable.

These topics are covered in the '54x workshop:

- '54x architecture/instruction set
- Use of the PC-based software simulator
- Use of the '54x assembler/linker
- C programming environment
- System architecture considerations
- Memory and I/O interfacing
- Development support

For registration information, pricing, or to enroll, call (800)336–5236, ext. 3904.

C.1.4 Assistance

For assistance to TMS320 questions on device problems, development tools, documentation, software upgrades, and new products, you can contact TI. See *If You Need Assistance* in *Preface* for information.

C.2 Part Order Information

This section describes the part numbers of ’54x devices, development support hardware, and software tools.

C.2.1 Device and Development Support Tool Nomenclature Prefixes

To designate the stages in the product development cycle, TI assigns prefixes to the part numbers of all TMS320 devices and support tools. Each TMS320 device has one of three prefix designators: TMX, TMP, or TMS. Each support tool has one of two possible prefix designators: TMDX or TMDS. These prefixes represent evolutionary stages of product development from engineering prototypes (TMX/TMDX) through fully qualified production devices and tools (TMS/TMDS). This development flow is defined below.

Device Development Evolutionary Flow:

TMX The part is an experimental device that is not necessarily representative of the final device’s electrical specifications.

TMP The part is a device from a final silicon die that conforms to the device’s electrical specifications but has not completed quality and reliability verification.

TMS The part is a fully qualified production device.

Support Tool Development Evolutionary Flow:

TMDX The development-support product that has not yet completed Texas Instruments internal qualification testing.

TMDS The development-support product is a fully qualified development support product.

TMX and TMP devices and TMDX development support tools are shipped with the following disclaimer:

“Developmental product is intended for internal evaluation purposes.”

TMS devices and TMDS development support tools have been fully characterized, and the quality and reliability of the device has been fully demonstrated. Texas Instruments standard warranty applies to these products.

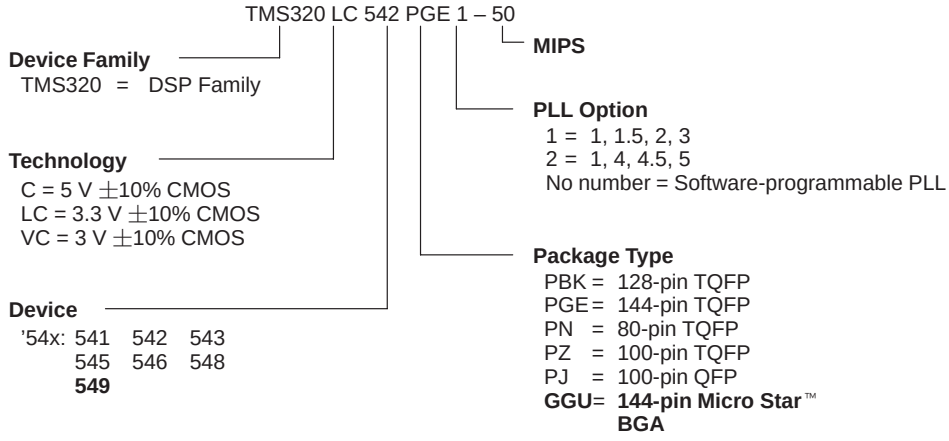
Note:

It is expected that prototype devices (TMX or TMP) have a greater failure rate than standard production devices. Texas Instruments recommends that these devices *not* be used in any production system, because their expected end-use failure rate is still undefined. Only qualified production devices should be used.

C.2.2 Device Nomenclature

TI device nomenclature includes the device family name and a suffix. Figure C–1 provides a legend for reading the complete device name for any ’54x device family member.

Figure C–1. TMS320C54x Device Nomenclature



C.2.3 Development Support Tools

Table C–1 lists the development support tools available for the ’54x, the platform on which they run, and their part numbers.

Table C–1. Development Support Tools Part Numbers

Development Tool	Platform	Part Number
Assembler/Linker	PC (DOS™)	TMDS324L850-02
C Compiler/Assembler/Linker	PC (DOS™, Windows™, OS/2™)	TMDS324L855-02
C Compiler/Assembler/Linker	HP (HP-UX™) / SPARC™ (Sun OS™)	TMDS324L555-08
C Source Debugger Conversion Software	PC (DOS™, Windows™, OS/2™) (XDS510™)	TMDS32401L0
C Source Debugger Conversion Software	HP (HP-UX™) / SPARC™ (Sun OS™) (XDS510WS™)	TMDS32406L0
Evaluation Module (EVM)	PC (DOS™, Windows™, OS/2™)	TMDX3260051
Simulator (C language)	PC (DOS™, Windows™)	TMDS324L851-02
Simulator (C language)	HP (HP-UX™) / SPARC™ (Sun OS™)	TMDS324L551-09
XDS510 Emulator†	PC (DOS™, Windows™, OS/2™)	TMDS00510
XDS510WS Emulator‡	HP (HP-UX™) / SPARC™ (Sun OS™) (SCSI)	TMDS00510WS
3 V/5 V PC/SPARC JTAG Emulation Cable	XDS510™ / XDS510WS™	TMDS3080002

† Includes XDS510 board and JTAG cable; TMDS32401L0 C-source debugger conversion software not included
‡ Includes XDS510WS box, SCSI cable, power supply, and JTAG cable; TMDS32406L0 C-source debugger conversion software not included

Appendix D

Submitting ROM Codes to TI

The size of a printed circuit board is a consideration in many DSP applications. To make full use of the board space, Texas Instruments offers this ROM code option that reduces the chip count and provides a single-chip solution. This option allows you to use a code-customized processor for a specific application while taking advantage of:

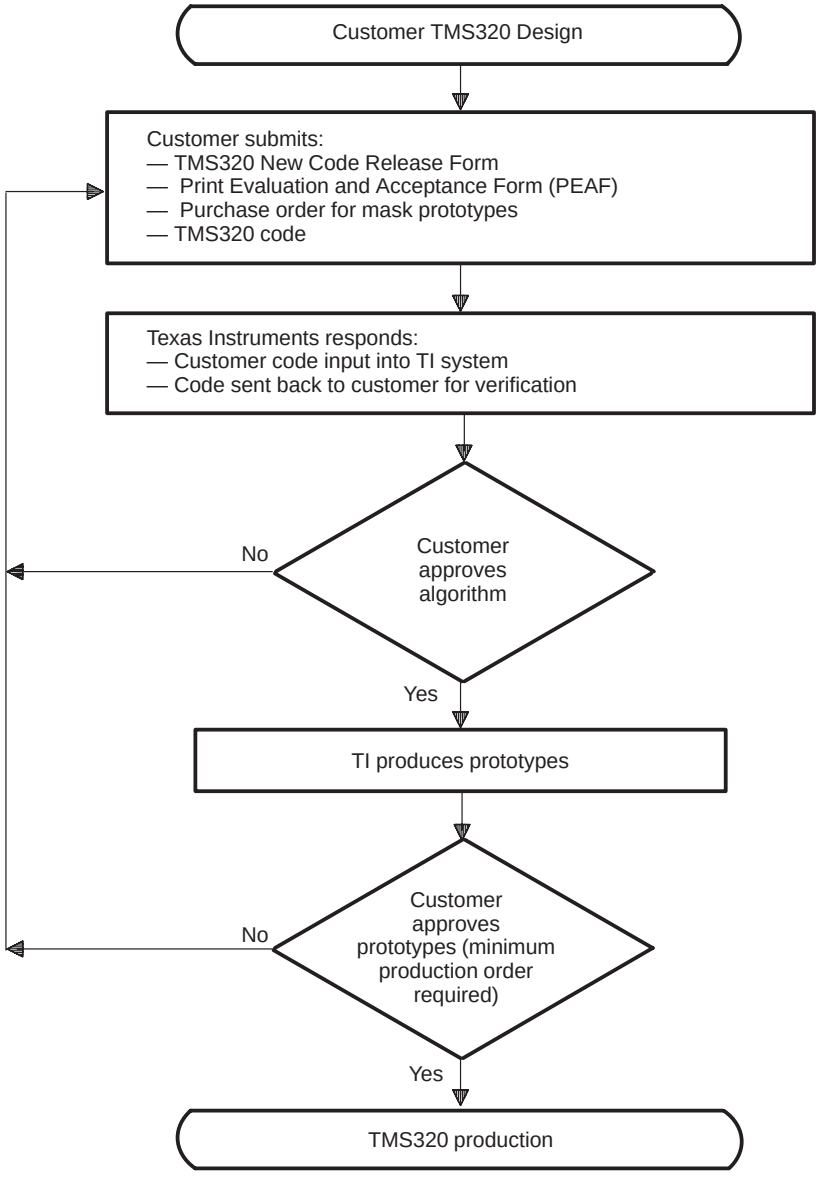
- Greater memory expansion
- Lower system cost
- Less hardware and wiring
- Smaller PCB

If a routine or algorithm is used often, it can be programmed into the on-chip ROM of a TMS320 DSP. TMS320 programs can also be expanded by using external memory; this reduces chip count and allows for a more flexible program memory. Multiple functions are easily implemented by a single device, thus enhancing system capabilities.

TMS320 development tools are used to develop, test, refine, and finalize the algorithms. The microprocessor/microcomputer (MP/$\overline{\text{MC}}$) mode is available on all ROM-coded TMS320 DSP devices when accesses to either on-chip or off-chip memory are required. The microprocessor mode is used to develop, test, and refine a system application. In this mode of operation, the TMS320 acts as a standard microprocessor by using external program memory. When the algorithm has been finalized, the code can be submitted to Texas Instruments for masking into the on-chip program ROM. At that time, the TMS320 becomes a microcomputer that executes customized programs from the on-chip ROM. Should the code need changing or upgrading, the TMS320 can once again be used in the microprocessor mode. This shortens the field-upgrade time and avoids the possibility of inventory obsolescence.

Figure D–1 illustrates the procedural flow for developing and ordering TMS320 masked parts. When ordering, there is a one-time, nonrefundable charge for mask tooling. A minimum production order per year is required for any masked-ROM device. ROM codes will be deleted from the TI system one year after the final delivery.

Figure D–1. TMS320 ROM Code Submittal Flowchart



The TMS320 ROM code may be submitted in one of the following forms:

- ❑ 3-1/2-inch floppy: COFF format from macro-assembler/linker (preferred)
- ❑ 5-1/4-inch floppy: COFF format from macro-assembler/linker
- ❑ Modem (BBS): COFF format from macro-assembler/linker
- ❑ EPROM (others): TMS27C64
- ❑ PROM: TBP28S166, TBP28S86

When code is submitted to TI for masking, the code is reformatted to accommodate the TI mask-generation system. System-level verification by the customer is therefore necessary to ensure the reformatting remains transparent and does not affect the execution of the algorithm. The formatting changes involve the removal of address-relocation information (the code address begins at the base address of the ROM in the TMS320 device and progresses without gaps to the last address of the ROM) and the addition of data in the reserved locations of the ROM for device ROM test. Because these changes have been made, a checksum comparison is not a valid means of verification.

With each masked-device order, the customer must sign a disclaimer that states:

> The units to be shipped against this order were assembled, for expediency purposes, on a prototype (that is, nonproduction qualified) manufacturing line, the reliability of which is not fully characterized. Therefore, the anticipated inherent reliability of these prototype units cannot be expressly defined.

and a release that states:

> Any masked ROM device may be resymbolized as TI standard product and resold as though it were an unprogrammed version of the device, at the convenience of Texas Instruments.

The use of the ROM-protect feature does not hold for this release statement. Additional risk and charges are involved when the ROM-protect feature is selected. Contact the nearest TI Field Sales Office for more information on procedures, leadtimes, and cost associated with the ROM-protect feature.

Appendix E

Glossary

A

A: See *accumulator A*.

ABU: See *autobuffering unit*.

ABUC: *ABU control register*. A register that controls the operation of the autobuffering unit.

accumulator: A register that stores the results of an operation and provides an input for subsequent arithmetic logic unit (ALU) operations.

accumulator A: 40-bit register that stores the result of an operation and provides an input for subsequent arithmetic logic unit (ALU) operations.

accumulator B: 40-bit registers that stores the result of an operation and provides an input for subsequent arithmetic logic unit (ALU) operations.

accumulator shift mode field (ASM): A 5-bit field in status register 1 (ST1) that specifies a shift value (from –16 to 15) used to shift an accumulator value when executing certain instructions, such as instructions with parallel loads and stores.

adder: A unit that adds or subtracts two numbers.

address: The location of a word in memory.

address bus: A group of connections used to route addresses. The ’54x has four 16-bit address busses: CAB, DAB, EAB, and PAB.

addressing mode: The method by which an instruction calculates the location of an object in memory.

address visibility mode (AVIS): A bit in processor mode status register (PMST) that determines whether or not the internal program address appears on the device’s external address bus pins.

AG: *accumulator guard bits*. An 8-bit register that contains bits 39–32 (the guard bits) of accumulator A.

AH: *accumulator A high word*. Bits 31–16 of accumulator A.

AL: *accumulator A low word*. Bits15–0 of accumulator A.

ALU: *arithmetic logic unit*. The part of the CPU that performs arithmetic and logic operations.

analog-to-digital (A/D) converter: Circuitry that translates an analog signal to a digital signal.

AR0–AR7: *auxiliary registers 0–7*. Eight 16-bit registers that can be accessed by the CPU and modified by the auxiliary register arithmetic units (ARAUs) and are used primarily for data memory addressing.

ARAU: See *auxiliary register arithmetic unit*.

ARP: See *auxiliary register pointer*.

ARR, ARR0, ARR1: *ABU address receive register* A 16-bit register that specifies the destination address at which the autobuffering unit begins storing received data.

ASM: See *accumulator shift mode field*.

autobuffering receiver enable (BRE): A bit in the BSP control extension register (BSPCE) that enables/disables the autobuffering receiver.

autobuffering receiver halt (HALTR): A bit in the BSP control extension register (BSPCE) that enables/disables the autobuffer receiver when the current half of the buffer is received.

autobuffering transmitter enable (BXE): A bit in the BSP control extension register (BSPCE) that enables/disables the autobuffering transmitter.

autobuffering transmitter halt (HALTX): A bit in the BSP control extension extension (BSPCE) that enables/disables the autobuffer transmitter when the current half of the buffer has been transmitted.

autobuffering unit: An extension to the synchronous serial port that reads and writes data to the synchronous serial port independent of the CPU.

auxiliary register arithmetic unit: An unsigned, 16-bit arithmetic logic unit (ALU) used to calculate indirect addresses using auxiliary registers.

auxiliary register file: The area in data memory containing the eight 16-bit auxiliary registers. See also *auxiliary registers*.

auxiliary register pointer (ARP): A 3-bit field in status register 0 (ST0) used as a pointer to the currently-selected auxiliary register, when the device is operating in ’C5x/’C2xx compatibility mode.

auxiliary registers: Eight 16-bit registers (AR7 – AR0) that are used as pointers to an address within data space. These registers are operated on by the auxiliary register arithmetic units (ARAUs) and are selected by the auxiliary register pointer (ARP). See also *auxiliary register arithmetic unit*.

AVIS: See *address visibility mode bit*.

AXR, AXR0, AXR1: *ABU address transmit register*. A 16-bit register that specifies the source address from which the autobuffering unit begins transmitting data.

B

B: See *accumulator B*.

bank-switching control register (BSCR): A 16-bit register that defines the external memory bank size and enables or disables automatic insertion of extra cycles when accesses cross memory bank boundaries.

barrel shifter: A unit that rotates bits in a word.

BDRR, BDRR0, BDRR1: *BSP data receive register*. Two 16-bit registers used to receive data through the buffered serial ports. BDRR0 corresponds to buffered serial port 0 and BDRR1 corresponds to buffered serial port 1.

BDXR, BDXR0, BDXR1: *BSP data transmit register*. Two 16-bit registers used to transmit data through the buffered serial ports. BDXR0 corresponds to buffered serial port 0 and BDXR1 corresponds to buffered serial port 1.

BG: *accumulator B guard bits*. An 8-bit register that contains bits 39–32 (the guard bits) of accumulator B.

BH: *accumulator B high word*. Bits 31–16 of accumulator B.

BK: See *circular buffer size register*.

BKR, BKR0, BKR1: *ABU receive buffer size register*. A 16-bit register that sets the size of the receive buffer for the autobuffering unit.

BKX, BKX0, BKX1: *ABU transmit buffer size register*. A 16-bit register that sets the size of the transmit buffer for the autobuffering unit.

BL: *accumulator B low word*. Bits 15–0 of accumulator B.

block-repeat active flag (BRAF): A bit in status register 1 (ST1) that indicates whether or not a block repeat is currently active.

block-repeat counter (BRC): A 16-bit register that specifies the number of times a block of code is to be repeated when a block repeat is performed.

block-repeat end address register (REA): A 16-bit memory-mapped register containing the end address of a code segment being repeated.

block-repeat start address register (RSA): A 16-bit memory-mapped register containing the start address of a code segment being repeated.

BMINT: See *buffer misalignment interrupt*.

boot: The process of loading a program into program memory.

boot loader: A built-in segment of code that transfers code from an external source to program memory at power-up.

BRAF: See *block-repeat active flag*.

BRC: See *block-repeat counter*.

BRE: See *autobuffering receiver enable*.

BRINT, BRINT0, BRINT1: See *BSP receive interrupt*.

BRSR: *BSP data receive shift register.* A 16-bit register that holds serial data received from the BDR pin. See also *BDRR*.

BSCR: See *bank-switching control register*.

BSP: *buffered serial port*. An enhanced synchronous serial port that includes an autobuffering unit (ABU) that reduces CPU overhead in performing serial operations.

BSP receive interrupt (BRINT, BRINT0, BRINT1): A bit in the interrupt flag register (IFR) that indicates the BSP data receive shift register (BRSR) contents have been copied to the BSP data receive register (BDRR). BRINT0 corresponds to buffered serial port 0 and BRINT1 corresponds to buffered serial port 1.

BSP transmit interrupt (BXINT, BXINT0, BXINT1): A bit in the interrupt flag register (IFR) that indicates the the BSP data transmit register (BDXR) contents has been copied to the BSP data transmit shift register (BXSR). BXINT0 corresponds to buffered serial port 0 and BXINT1 corresponds to buffered serial port 1.

BSPC, BSPC0, BSPC1: Buffered serial port control registers 0 and 1. A 16-bit register that contains status and control bits for the buffered serial port. BSPC0 corresponds to buffered serial port 0 and BSPC1 corresponds to buffered serial port 1.

BSPCE, BSPCE0, BSPCE1: *BSP control extension register.* A 16-bit register that contains status and control bits for the buffered serial port (BSP) interface. The 10 LSBs of the BSPCE are dedicated to serial port interface control, whereas the 6 MSBs are used for autobuffering unit (ABU) control.

buffer misalignment interrupt (BMINT): A '549 feature that detects potential error conditions and indicates lost words on a serial port interface.

burst mode: A synchronous serial port mode in which a single word is transmitted following a frame synchronization pulse (FSX and FSR).

butterfly: A kernel function for computing an N-point fast Fourier transform (FFT), where N is a power of 2. The combinational pattern of inputs resembles butterfly wings.

BXE: See *autobuffering transmitter enable*.

BXSR: *BSP data transmit shift register.* A 16-bit register that holds serial data to be transmitted from the BDX pin. See also *BDXR*.

C

C: See *carry bit*.

C16: A bit in status register 1 (ST1) that determines whether the ALU operates in dual 16-bit mode or in double-precision mode.

CAB: *C address bus*. A bus that carries addresses needed for accessing data memory.

carry bit (C): A bit in status register 0 (ST0) used by the ALU in extended arithmetic operations and accumulator shifts and rotates. The carry bit can be tested by conditional instructions.

CB: *C bus*. A bus that carries operands that are read from data memory.

circular buffer size register (BK): A 16-bit register used by the auxiliary register arithmetic units (ARAUs) to specify the data-block size in circular addressing.

CLKDV: See *internal transmit clock division factor*.

CLKP: See *clock polarity*.

CLKOUT off (CLKOFF): A bit in processor mode status register (PMST) that enables/disables the CLKOUT output.

clock mode (MCM): A bit in the serial port control register (SPC), buffered serial port control register (BSPC), and TDM serial port control register (TSPC) that specifies the source of the clock for CLKX.

clock polarity (CLKP): A bit in the BSP control extension register (BSPCE) that indicates when the data is sampled by the receiver and sent by the transmitter.

CMPT: See *compatibility mode*.

code: A set of instructions written to perform a task; a computer program or part of a program.

cold boot: The process of loading a program into program memory at power-up.

compare, select, and store unit (CSSU): An application-specific hardware unit dedicated to add/compare/select operations of the Viterbi operator.

compatibility mode (CMPT): A bit in status register 1 (ST1) that determines whether or not the auxiliary register pointer (ARP) is used to select an auxiliary register in single indirect addressing mode.

compiler mode (CPL): A bit in status register 1 (ST1) that determines whether the CPU uses the data page pointer or the stack pointer to generate data memory addresses in direct addressing mode.

continuous mode: A synchronous serial port mode in which only one frame synchronization pulse (FSX and FSR) is necessary to transmit several packets at maximum frequency.

CPL: See *compiler mode*.

CSSU: *See compare, select, and store unit*.

D

DAB: *D address bus*. A bus that carries addresses needed for accessing data memory.

DAB address register (DAR): A register that holds the address to be put on the DAB to address data memory for reads via the DB.

DAGEN: See *data-address generation logic (DAGEN)*.

DAR: See *DAB address register*.

DARAM: *dual-access RAM*. Memory that can be accessed twice in the same clock cycle.

data address bus: A group of connections used to route data memory addresses. The ’54x has three 16-bit buses that carry data memory addresses: CAB, DAB, and EAB.

data-address generation logic (DAGEN): Logic circuitry that generates the addresses for data memory reads and writes. See also *program-address generation logic (PAGEN).*

data bus: A group of connections used to route data. The ’54x has three 16-bit data buses: CB, DB, and EB.

data memory: A memory region used for storing and manipulating data. Addresses 00h–1Fh of data memory contain CPU registers. Addresses 20h–5Fh of data memory contain peripheral registers.

data page pointer (DP): A 9-bit field in status register 0 (ST0) that specifies which of 512, 128 × 16 word pages is currently selected for direct address generation. DP provides the nine MSBs of the data-memory address; the dma provides the lower seven. See also *dma.*

data ROM (DROM): A bit in processor mode status register (PMST) that determines whether or not part of the on-chip ROM is mapped into data space.

DB: *D bus*. A bus that carries operands that are read from data memory.

digital loopback mode: A synchronous serial port test mode in which the DLB bit connects the receive pins to the transmit pins on the same device to test if the port is operating correctly.

digital loopback mode (DLB) bit: A bit in the serial port control register (SPC), buffered serial port control register (BSPC), and TDM serial port control register (TSPC) that puts the serial port in digital loopback mode.

digital-to-analog (D/A) converter: Circuitry that translates a digital signal to an analog signal.

direct data-memory address bus: A 16-bit bus that carries the direct address for data memory.

direct memory address (dma, DMA) : The seven LSBs of a direct-addressed instruction that are concatenated with the data page pointer (DP) to generate the entire data memory address. See also *data page pointer.*

dma: *See direct memory address*.

DP: *See data page pointer.*

DRB: *direct data-memory address bus*. A 16-bit bus that carries the direct address for data memory.

DROM: *See data ROM*.

DRR, DRR0, DRR1: *serial port data receive register*. Two 16-bit registers used to receive data through the synchronous serial ports. DRR0 corresponds to synchronous serial port 0 and DRR1 corresponds to synchronous serial port 1.

DSP interrupt (DSPINT): A bit in the HPI control register (HPIC) that enables/disables an interrupt from a host device to the ’54x.

DXR, DXR0, DXR1: *serial port data transmit register*. Two 16-bit registers used to transmit data through the synchronous serial ports. DXR0 corresponds to synchronous serial port 0 and DXR1 corresponds to synchronous serial port 1.

E

EAB: *E address bus*. A bus that carries addresses needed for accessing data memory.

EAB address register (EAR): A register that holds the address to be put on the EAB to address data memory for reads via the EB.

EB: *E bus*. A bus that carries data to be written to memory.

exponent encoder (EXP): A hardware device that computes the exponent value of the accumulator.

external interrupt: A hardware interrupt triggered by a pin ($\overline{\text{INT0}}$–$\overline{\text{INT3}}$).

F

fast Fourier transform (FFT): An efficient method of computing the discrete Fourier transform, which transforms functions between the time domain and frequency domain. The time-to-frequency domain is called the forward transform, and the frequency-to-time domain is called the inverse transformation. See also *butterfly*.

fast return register (RTN): A 16-bit register used to hold the return address for the fast return from interrupt (RETF[D]) instruction.

FE: See *format extension*.

FIG: See *frame ignore*.

format (FO): A bit in the serial port control register (SPC), buffered serial port control register (BSPC), and TDM serial port control register (TSPC) that specifies the word length of the serial port transmitter and receiver.

format extension (FE): A bit in the BSP control extension register (BSPCE) used in conjunction with the format bit (FO) to specify the word length of the BSP serial port transmitter and receiver.

frame ignore (FIG): A bit in the BSP control extension register (BSPCE) used only in transmit continuous mode with external frame and in receive continuous mode.

frame synchronization mode (FSM): A bit in the serial port control register (SPC), buffered serial port control register (BSPC), and TDM serial port control register (TSPC) that specifies whether frame synchronization pulses (FSX and FSR) are required for serial port operation.

frame synchronization polarity (FSP): A bit in the BSP control extension register (BSPCE) that determines the status of the frame synchronization (FSX and FSR) pulses.

fractional mode (FRCT): A bit in status register 1 (ST1) that determines whether or not the multiplier output is left-shifted by one bit.

Free bit: A bit in the serial port control register (SPC), buffered serial port control register (BSPC), timer control register (TCR), and TDM serial port control register (TSPC) used in conjunction with the Soft bit to determine the state of the serial port or timer clock when a breakpoint is encountered in the high-level language debugger. See also *Soft bit*.

FSM: See *frame synchronization mode*.

FSP: See *frame synchronization polarity*.

G

general-purpose input/output pins: Pins that can be used to supply input signals from an external device or output signals to an external device. These pins are not linked to specific uses; rather, they provide input or output signals for a variety of purposes. These pins include the general-purpose $\overline{\text{BIO}}$ input pin and XF output pin.

H

HALTR: See *autobuffering receiver halt*.

HALTX: See *autobuffering transmitter halt*.

hardware interrupt: An interrupt triggered through physical connections with on-chip peripherals or external devices.

HINT: *'54x-to-Host Processor Interrupt*. A bit in the HPI control register (HPIC) that enables/disables an interrupt from the '54x to a host device.

HM: *See hold mode.*

hold mode (HM): A bit in status register ST1 that determines whether the CPU enters the hold state in normal mode or concurrent mode.

host-only mode (HOM): The mode that allows the host to access HPI memory while the 54x is in IDLE2 (all internal clocks stopped) or in reset mode.

host port interface (HPI): An 8-bit parallel interface that the CPU uses to communicate with a host processor.

HPI address register (HPIA): A 16-bit register that stores the address of the host port interface (HPI) memory block. The HPIA can be preincremented or postincremented.

HPI control register (HPIC): A 16-bit register that contains status and control bits for the host port interface (HPI).

I

IFR: See *interrupt flag register*.

IMR: See *interrupt mask register*.

IN0: *input 0 bit*. A bit in the serial port control register (SPC), buffered serial port control register (BSPC), and TDM serial port control register (TSPC) that allows the CLKR pin to be used as an input. IN0 reflects the current level of the CLKR pin of the device.

IN1: *input 1 bit*. A bit in the serial port control register (SPC), buffered serial port control register (BSPC), and TDM serial port control register (TSPC) that allows the CLKX pin to be used as an input. IN1 reflects the current level of the CLKX pin of the device.

internal transmit clock division factor (CLKDV): A 5-bit field in the BSP control extension register (BSPCE) that determines the internal transmit clock duty cycle.

interrupt: A condition caused by internal hardware, an event external to the CPU, or by a previously executed instruction that forces the current program to be suspended and causes the processor to execute an interrupt service routine corresponding to the interrupt.

interrupt flag register (IFR): A 16-bit memory-mapped register that flags pending interrupts.

interrupt mask register (IMR): A 16-bit memory-mapped register that masks external and internal interrupts.

interrupt mode (INTM): A bit in status register 1 (ST1) that globally masks or enables all interrupts.

interrupt service routine (ISR): A module of code that is executed in response to a hardware or software interrupt.

IPTR: *interrupt vector pointer* A 9-bit field in the processor mode status register (PMST) that points to the 128-word page where interrupt vectors reside.

IR: *instruction register.* A 16-bit register used to hold a fetched instruction.

L

latency: The delay between when a condition occurs and when the device reacts to the condition. Also, in a pipeline, the necessary delay between the execution of two instructions to ensure that the values used by the second instruction are correct.

LSB: *least significant bit*. The lowest order bit in a word.

M

maskable interrupts: A hardware interrupt that can be enabled or disabled through software.

MCM: See *clock mode*.

memory map: A map of the addressable memory space accessed by the ’54x processor partitioned according to functionality (memory, registers, etc.).

memory-mapped register (MMR): The ’54x processor registers mapped into page 0 of the data memory space.

microcomputer mode: A mode in which the on-chip ROM is enabled and addressable for program accesses.

microprocessor/microcomputer (MP/$\overline{\text{MC}}$): A bit in the processor mode status register (PMST) that indicates whether the processor is operating in microprocessor or microcomputer mode. See also *microcomputer mode; microprocessor mode.*

microprocessor mode: A mode in which the on-chip ROM is disabled for program accesses.

micro stack: A stack that provides temporary storage for the address of the next instruction to be fetched when the program address generation logic is used to generate sequential addresses in data space.

MSB: *most significant bit.* The highest order bit in a word.

multiplier: A 17-bit $\times$ 17-bit multiplier that generates a 32-bit product. The multiplier executes multiple operations in a single cycle and operates using either signed or unsigned 2s-complement arithmetic.

N

nested interrupt: A higher-priority interrupt that must be serviced before completion of the current interrupt service routine (ISR). An executing ISR can set the interrupt mask register (IMR) bits to prevent being suspended by another interrupt.

nonmaskable interrupt: An interrupt that can be neither masked by the interrupt mask register (IMR) nor disabled by the INTM bit of status register 1 (ST1).

O

OVA: *overflow flag A.* A bit in status register0 (ST0) that indicates the overflow condition of accumulator A.

OVB: *overflow flag B.* A bit in status register 0 (ST0) that indicates the overflow condition of accumulator B.

overflow: A condition in which the result of an arithmetic operation exceeds the capacity of the register used to hold that result.

overflow flag: A flag that indicates whether or not an arithmetic operation has exceeded the capacity of the corresponding register.

OVLY: See *RAM overlay*.

OVM: *overflow mode bit.* A bit in status register 1 (ST1) that specifies how the ALU handles an overflow after an operation.

P

PAB: See *program address bus.*

PAGEN: See *program-address generation logic (PAGEN)*.

PB: See *program data bus*.

PC: See *program counter.*

PCM: See *pulse coded modulation mode*.

pipeline: A method of executing instructions in an assembly-line fashion.

pmad: *program-memory address.* A 16-bit immediate program-memory address.

PMST: *processor mode status register*. A 16-bit status register that controls the memory configuration of the device. See also *ST0, ST1.*

pop: Action of removing a word from a stack.

PRD: *timer period register*. A 16-bit register that defines the period for the on-chip timer.

program address bus (PAB): A 16-bit bus that provides the address for program memory reads and writes.

program-address generation logic (PAGEN): Logic circuitry that generates the address for program-memory reads and writes, and the address for data memory in instructions that require two data operands. This circuitry can generate one address per machine. See also *data-address generation logic (DAGEN)*.

program address register (PAR): A register that holds the address to be put on the PAB to address memory for reads via the PB.

program controller: Logic circuitry that decodes instructions, manages the pipeline, stores status of operations, and decodes conditional operations.

program counter (PC): A 16-bit register that indicates the location of the next instruction to be executed.

program counter extension register (XPC): A register that contains the upper 7 bits of the current program memory address.

program data bus (PB): A bus that carries the instruction code and immediate operands from program memory.

program memory: A memory region used for storing and executing programs.

pulse coded modulation mode (PCM): A bit in the BSP control extension register (BSPCE) that enables/disables the BSP transmitter.

push: Action of placing a word onto a stack.

R

RAM overlay (OVLY): A bit in the processor mode status register (PMST) that determines whether or not on-chip RAM is mapped into the program space in addition to data space.

RC: See *repeat counter*.

REA: See *block-repeat end address*.

receive buffer half received (RH): A bit in the BSP control extension register (BSPCE) that indicates which half of the receive buffer has been received.

receive ready (RRDY): A bit in the serial port control register (SPC), buffered serial port control register (BSPC), and TDM serial port control register (TSPC) that transitions from 0 to 1 to indicate the data receive shift register (RSR) contents have been copied to the data receive register (DRR) and that data can be read.

receiver reset ($\overline{\text{RRST}}$): A bit in the serial port control register (SPC), buffered serial port control register (BSPC), and TDM serial port control register (TSPC) that resets the serial port receiver.

receive shift register full (RSRFULL): A bit in the serial port control register (SPC) and buffered serial port control register (BSPC) that indicates if the serial port receiver has experienced overrun.

register: A group of bits used for temporarily holding data or for controlling or specifying the status of a device.

repeat counter (RC): A 16-bit register used to specify the number of times a single instruction is executed.

reset: A means of bringing the CPU to a known state by setting the registers and control bits to predetermined values and signaling execution to start at a specified address.

RH: See *receive buffer half received*.

RINT, RINT0, RINT1: See *serial port receive interrupt*.

RRDY: See *receive ready*.

$\overline{\text{RRST}}$: See *receiver reset*.

RSA: See *block-repeat start address*.

RSR: *data receive shift register.* A 16-bit register that holds serial data received from the DR pin. See also *data receive register (DRR)*.

RSRFULL: See *receive shift register full*.

RTN: See *fast return register*.

S

SARAM: *single-access RAM*. Memory that can be read written once during one clock cycle.

saturation on multiplication (SMUL): A bit in the processor mode status register (PMST) that determines whether saturation of a multiplication result occurs before performing the accumulation in a MAC or MAS instruction.

saturation on store (SST): A bit in the processor mode status register (PMST) that determines whether saturation of the data from the accumulator occurs before storing in memory.

serial port interface: An on-chip full-duplex serial port interface that provides direct serial communication to serial devices with a minimum of external hardware, such as codecs and serial analog-to-digital (A/D) and digital-to-analog (D/A) converters. Status and control of the serial port is specified in the serial port control register (SPC).

serial port receive interrupt (RINT, RINT0, RINT1): A bit in the interrupt flag register (IFR) that indicates the data receive shift register (RSR) contents have been copied to the data receive register (DRR). RINT0 corresponds to synchronous serial port 0 and RINT1 corresponds to synchronous serial port 1.

serial port transmit interrupt (XINT, XINT0, XINT1): A bit in the interrupt flag register (IFR) that indicates the the data transmit register (DXR) contents has been copied to the data transmit shift register (XSR). XINT0 corresponds to synchronous serial port 0 and XINT1 corresponds to synchronous serial port 1.

shared-access mode (SAM): The mode that allows both the ’54x and the host to access HPI memory. In this mode, asynchronous host accesses are synchronized internally and, in case of conflict, the host has access priority and the ’54x waits one cycle.

shared-access mode (SMOD): A bit in the HPI control register (HPIC) that enables/disables the shared access mode (SAM). See also *shared-access mode (SAM)* and *host-only mode (HOM)*.

shifter: A hardware unit that shifts bits in a word to the left or to the right.

sign-control logic: Circuitry used to extend data bits (signed/unsigned) to match the input data format of the multiplier, ALU, and shifter.

sign extension: An operation that fills the high order bits of a number with the sign bit.

sign-extension mode (SXM): A bit in status register 1 (ST1) that enables sign extension in CPU operations.

SMUL: See *saturation on multiplication*.

Soft bit: A bit in the serial port control register (SPC), buffered serial port control register (BSPC), timer control register (TCR), and TDM serial port control register (TSPC) used in conjunction with the Free bit to determine the state of the serial port or timer clock when a breakpoint is encountered in the high-level language debugger. See also *Free bit*.

software interrupt (SINT): An interrupt caused by the execution of an INTR or TRAP instruction.

software wait-state register (SWWSR): A 16-bit register that selects the number of wait states for the program, data, and I/O spaces of off-chip memory.

SP: See *stack pointer*.

SPC, SPC0, SPC1: *serial port control register*. A 16-bit register that contains status and control bits for the synchronous serial port. SPC0 corresponds to synchronous serial port 0 and SPC1 corresponds to synchronous serial port 1.

SST: See *saturation on store*.

ST0: A 16-bit register that contains ’54x status and control bits. See also *PMST; ST1.*

ST1: A16-bit register that contains ’54x status and control bits. See also *PMST, ST0.*

stack: A block of memory used for storing return addresses for subroutines and interrupt service routines and for storing data.

stack pointer (SP): A register that always points to the last element pushed onto the stack.

SXM: See *sign-extension mode.*

T

TADD: *TDM address*. A single, bidirectional address line that identifies which devices on the four-wire serial bus should read in the data on the TDM data (TDAT) line.

TC: *test/control flag*. A bit in status register 0 (ST0) that is affected by test operations.

TCLK: *TDM clock*. A single, bidirectional clock line for TDM operation.

TCR: *timer control register*. A 16-bit memory-mapped register that contains status and control bits for the on-chip timer.

TCSR: *TDM channel select register*. A 16-bit memory-mapped register that specifies in which of the eight time slots (channels) a device on the four-wire serial bus is to transmit.

TDAT: *TDM data*. A single, bidirectional line from which all TDM data is carried.

TDM receive interrupt (TRINT): A bit in the interrupt flag register (IFR) that indicates the TDM data receive shift register (TRSR) contents have been copied to the TDM data receive register (TRCV).

TDM transmit interrupt (TXINT): A bit in the interrupt flag register (IFR) that indicates the TDM data transmit register (TDXR) contents have been copied to the data transmit shift register (XSR).

TDXR: *TDM data transmit register*. A 16-bit register used to transmit data through the TDM serial port. See also *XSR*.

temporary register (T): A 16-bit register that holds one of the operands for multiply and store instructions, the dynamic shift count for the add and subtract instructions, or the dynamic bit position for the bit test instructions.

TIM: *timer counter register*. A 16-bit memory-mapped register that specifies the current count for the on-chip timer.

time-division multiplexed (TDM): A bit in the TDM serial port control register (TSPC) that enables/disables the TDM serial port.

time-division multiplexing: The process by which a single serial bus is shared by up to eight ’54x devices with each device taking turns to communicate on the bus. There are a total of eight time slots (channels) available. During a time slot, a given device may talk to any combination of devices on the bus.

timer divide-down register (TDDR): A 4-bit field in the timer control register (TCR) that specifies the timer divide-down ratio (period) for the on-chip timer.

timer interrupt (TINT): A bit in the interrupt flag register (IFR) that indicates the timer counter register (TIM) has decremented past 0.

timer prescaler counter (PSC): A 4-bit field in the timer control register (TCR) that specifies the count for the on-chip timer.

timer reload (TRB): A bit in the timer control register (TCR) that resets the on-chip timer.

timer stop status (TSS): A bit in the timer control register (TCR) that stops and restarts the on-chip timer.

TINT: See *timer interrupt*.

TRAD: *TDM receive address register*. A 16-bit memory-mapped register that contains information about the status of the TADD line in the TDM serial port.

transition register (TRN): A 16-bit register that holds the transition decision for the path to new metrics to perform the Viterbi algorithm.

transmit buffer half transmitted (XH): A bit in the BSP control extension register (BSPCE) that indicates which half of transmit buffer transmitted.

transmit mode (TXM): A bit in the serial port control register (SPC), buffered serial port control register (BSPC), and TDM serial port control register (TSPC) that specifies the source of the frame synchronization transmit (FSX) pulse.

transmit ready (XRDY): A bit in the serial port control register (SPC), buffered serial port control register (BSPC), and TDM serial port control register (TSPC) that transitions from 0 to 1 to indicate the data transmit register (DXR) contents have been copied to the data transmit shift register (XSR) and that data is ready to be loaded with a new data word.

transmit shift register empty ($\overline{\text{XSREMPTY}}$): A bit in the serial port control register (SPC) and buffered serial port control register (BSPC) that indicates if the serial port transmitter has experienced underflow.

transmitter reset ($\overline{\text{XRST}}$): A bit in the serial port control register (SPC), buffered serial port control register (BSPC), and TDM serial port control register (TSPC) that resets the serial port transmitter.

TRCV: *TDM data receive register*. A register used to receive data through the TDM serial port.

TRINT: See *TDM receive interrupt*.

TRN: See *transition register.*

TRSR: *TDM data receive shift register*. A 16-bit register that holds serial data received from the TDM data (TDAT) line. See also *TRCV*.

TRTA: *TDM receive/transmit address register*. The lower half of this register specifies the receive address of the device; the upper half of this register specifies the transmit address.

TSPC: *TDM serial port control register*. A 16-bit memory-mapped register that contains status and control bits for the TDM serial port.

TXINT: See *TDM transmit interrupt*.

TXM: See *transmit mode*.

W

wait state: A period of time that the CPU must wait for external program, data, or I/O memory to respond when reading from or writing to that external memory. The CPU waits one extra cycle (one CLKOUT1 cycle) for every wait state.

warm boot: The process by which the processor transfers control to the entry address of a previously-loaded program.

X

XF: *XF status flag*. A bit in status register ST1 that indicates the status of the XF pin.

XH: See *transmit buffer half transmitted*.

XINT, XINT0, XINT1: See *serial port transmit interrupt*.

XPC: See *program counter extension.*

XRDY: See *transmit ready*.

$\overline{\text{XRST}}$: See *transmitter reset*.

XSR: *data transmit shift register*. A 16-bit register that holds serial data to be transmitted from the DX pin (or TDX pin when TDM = 1). See also *TDXR*.

$\overline{\text{XSREMPTY}}$: See *transmit shift register empty*.

Z

ZA: *zero detect A*. A signal that indicates when accumulator A contains a 0.

ZB: *zero detect B*. A signal that indicates when accumulator B contains a 0.

zero detect: See *ZA* and *ZB*.

zero fill: A method of filling the low- or high-order bits with zeros when loading a 16-bit number into a 32-bit field.

Appendix F

Summary of Updates in This Document

This appendix provides a summary of the updates in this revision of the document. Updates within paragraphs appear in a **bold typeface**.

Page:	Change or Add:
vi	Added the following sentence to the end of the first paragraph: Many of these documents are located on the internet at http://www.ti.com. Click DSPS Solutions, then click DSP Literature.
vii	Added the following document reference: ***TMS320C548/C549 Bootloader Technical Reference*** (literature number SPRU288) describes the process the bootloader uses to transfer user code from an external source to the program memory at power up. (Presently available only on the internet.)
1–3	Replaced Figure 1–1, *Evolution of the TMS320 Family*, with an updated family graphic.
1–6	Changed the first second-level bullet under Memory to: ■ 192K words × 16-bit addressable memory space (64K-words program, 64K-words data, and 64K-words I/O), with extended program memory (8M words) in the ’548 **and ’549**.
1–6	Added the following ’549 information to the configuration table at the bottom of the page:

Device	Program ROM	Program/Data ROM	DARAM†	SARAM‡
’549	16	16	8	24

		Changed the first sentence of the note below the option table to: †The ’**541B**, ’545**A**, ’546**A**, ’548, and **’549** are designated as LP-type devices.
1–8		Added the following ’549 information to the ports table at the top of the page:

		Serial Ports		
Device	Host Port Interface	Synchronous	Buffered	Time-Division Multiplexed
’549	1	0	2	1

Page: **Change or Add:**

1–8 Deleted the ’548 3-V power supply and 12.5-ns/10-ns speed designation in the table that lists power supply, speed, and package type information. Added the ’541B, ’545A, ’546A, and ’549 information shown in the revised table below. Deleted the note following the table regarding ’548 speed.

Device	Power Supply	Speed	Package
’541	5 V	25 ns	100-pin TQFP
	3 V / 3.3 V	25 ns/20 ns	100-pin TQFP
’541B	**3 V / 3.3 V**	**15 ns**	**100-pin TQFP**
’542	5 V	25 ns	144-pin TQFP
	3 V / 3.3 V	25 ns/20 ns	128-pin/144-pin TQFP
’543	3 V / 3.3 V	25 ns/20 ns	100-pin TQFP
’545	3 V / 3.3 V	25 ns/20 ns	128-pin TQFP
’545A	**3 V / 3.3 V**	**15 ns**	**128-pin TQFP**
’546	3 V / 3.3 V	25 ns/20 ns	100-pin TQFP
’546A	**3 V / 3.3 V**	**15 ns**	**100-pin TQFP**
’548	3.3 V	20 ns/15 ns	144-pin TQFP
’549	**3.3 V**	**15 ns/12.5 ns**	**144-pin TQFP/144-pin Micro Star™ BGA**
’VC549	**3.3 V (2.5 V core)**	**10 ns**	**144-pin TQFP/144-pin Micro Star™ BGA**

2–5 Added the following ’549 information to Table 2–2, *Program and Data Memory on the TMS320C54x Devices*:

Memory Type	’549
ROM:	**16K**
Program	**16K**
Program/data	**16K**
DARAM†	**8K**
SARAM†	**24K**

Page: **Change or Add:**

2–5 Changed the second paragraph in section 2.2.1, *On-Chip ROM*, to the following:

> On devices with a small amount of ROM (2K words), the ROM contains a boot-loader that is useful for booting to faster on-chip or external RAM. For boot-loading details on all ’54x devices except the ’548 and ’549, see *TMS320C54x DSP Reference Set, Volume 4: Applications Guide.* For boot-loading details on the ’548 and ’549, see *TMS320C548/549 Bootloader Technical Reference.*

2–6 Changed the second sentence in section 2.2.2, *On-chip Dual-Access RAM (DARAM)*, to:

> Because each DARAM block can be accessed twice per machine cycle, the central processing unit (CPU) **and peripherals such as the buffered serial port (BSP) and host port interface (HPI)** can read from and write to a DARAM memory address in the same cycle.

2–11 Changed the first sentence in section 2.7.2, *Software-Programmable Wait-State Generator*, to:

> The software-programmable wait-state generator extends external bus cycles up to seven machine cycles **(14 machine cycles in the ’549)** to interface with slower off-chip memory and I/O devices.

2–12 Added the following ’549 information to Table 2–3, *Host Port Interfaces on the TMS320C54x Devices*:

On-Chip Peripheral	’549
Host port interface	**1**

2–13 Added the following ’549 information to Table 2–4, *Serial Port Interfaces on the TMS320C54x Devices*:

Serial Ports	’549
Synchronous	**0**
Buffered	**2**
TDM	**1**

Page:	Change or Add:
2–14	Changed the first sentence in section 2.9, *External Bus Interface*, to: The ’54x can address up to 64K words of data memory, 64K words of program memory (8M words in the ’548 and **’549**), and up to 64K words of 16-bit parallel I/O ports.
3–1	Changed the fourth sentence in the first paragraph to: In some devices, such as the ’548 and **’549**, the memory structure has been modified through overlay and paging schemes.
3–2	Changed the third sentence in the first paragraph in section 3.1, *Memory Space*, to: Together, these three spaces provide a total address range of 192K words (except in the ’548 and ’**549**).
3–7	Added Figure 3–5, *Memory Maps for the TMS320C549.*
3–8	Changed the heading of section 3.1.1 to: Extended Program Memory (Available on TMS320C548**/549**) Changed the first paragraph to: The ’548 **and ’549** use a paged extended memory scheme in program space to allow access of up to 8192K words of program memory. To implement this scheme, the ’548 **and ’549** include several additional features: Changed the second paragraph to: Program memory in the ’548 **and ’549** is organized into 128 pages that are each 64K words in length, as shown in Figure 3–5.
3–9	Changed the sentence above the first bulleted list to: To facilitate page switching through software, the ’548 **and ’549** have six special instructions that affect the XPC: Changed the sentence above the second bulleted list to: In addition to these new instructions, two ’54x instructions are extended in the ’548 **and ’549** to use 23 bits.
3–10	Changed the first sentence of the first paragraph to: The external program memory on the ’54x devices (except on the ’548 **and ’549**) addresses up to 64K 16-bit words.

Page: **Change or Add:**

3–10 Added the following ’549 information to Table 3–1, *On-Chip Program Memory Available on the TMS320C54x Devices*:

Device	ROM (MP/$\overline{MC}$ = 0)	DARAM (OVLY = 1)	SARAM (OVLY = 1)
’549	16K	8K	24K

3–11 Added the following ’549 information to Figure 3–8, *On-Chip ROM Block Organization*:

’549

C000–CFFF
D000–DFFF
E000–EFFF
F000–FFFF

3–12 Changed the first paragraph in section 3.2.4, *On-Chip ROM Code Contents and Mapping*, to:

> ’54x devices have either large (**16K**, 24K, 28K, or 48K words) on-chip ROM or 2K words of on-chip ROM. A large on-chip ROM may be programmed with your code, while the content of a 2K-word on-chip ROM is defined by Texas Instruments. On ’54x devices **with on-chip bootloader** ROM, the 2K words (at F800h to FFFFh) **may** contain **one or more of the following, depending on the specific device:**

3–13 Changed the label above the second memory map to:

> ’542/543/548**/549**

3–14 Added the following ’549 information to Table 3–2, *On-Chip Data Memory Available on the TMS320C54x Devices:*

Device	Program/Data ROM (DROM = 1)	DARAM	SARAM
’549	16K	8K	24K

3–16 Changed the label above the RAM block on the far right to:

> ’548**/549**

Page: **Change or Add:**

3–17 Added 03FFh as the last address at DARAM block DP = 7.

3–19 Changed the XPC description at address 1E in Table 3–3, *CPU Memory-Mapped Registers,* to:

Address	Name	Description
1E	XPC	Program counter extension register ('548 **and '549**)

3–21 Changed the heading of section 3.3.4.11 to:

*Program Counter Extension Register (XPC, Available on '548**/549**)*

4–3 Deleted the following from the third sentence of the Bit 9 function description:

. . . . or when an expression that writes to B does not overflow.

5–6 Changed the second sentence in the note at the bottom of the page to:

However, because the '548 **and '549** have 23 address lines, the program-memory location in a '548 **or '549** device is specified by the lower 23 bits of accumulator A.

6–2 Changed the second sentence of the first paragraph to:

The '54x can address a total of 64K words of program memory using the program address bus (PAB); the '548 **and '549** have an additional seven address lines to provide external access to 128 64K-word pages.

Changed the sentence below the fifth bulleted item to:

One additional register is used in the '548 **and '549** to address extended memory:

6–5 Changed the first sentence in the first paragraph to:

XPC is a 7-bit register that selects the current page of program memory for the '548 **and '549**.

6–8 Changed the heading of section 6.3.3 to:

Far Branches (Available on TMS320C548**/549**)

Page:	Change or Add:
6–8	Changed the first sentence of section 6.3.3 to: To allow branches to extended memory, the ’548 **and ’549** include two far branch instructions: Changed the first sentence under the bulleted list to: Table 6–5 shows the far branch instructions in the ’548 **and ’549** (both nondelayed and delayed) and the number of cycles needed to execute these instructions.
6–11	Changed the heading of section 6.4.3 to: Far Calls (Available on TMS320C548**/549**) Changed the first sentence in section 6.4.3 to: To allow calls to extended memory, the ’548 **and ’549** include two far call instructions: Changed the first sentence of the paragraph below the bulleted list to: Table 6–8 shows the far call instructions in the ’548 **and ’549** (nondelayed and delayed) and the number of cycles needed to execute these instructions.
6–12	Changed the second paragraph in section 6.5, *Returns*, to: The ’548 **and ’549** offer an additional return instruction: an unconditional far return, both nondelayed and delayed.
6–14	Changed the heading of section 6.5.3 to: Far Returns (Available on TMS320C548**/549**) Changed the first sentence in section 6.5.3 to: To allow returns from extended memory, the ’548 **and ’549** include two far return instructions:
6–15	Changed the first sentence to: Table 6–11 shows the far return instructions in the ’548 **and ’549** (nondelayed and delayed) and the number of cycles needed to execute these instructions.
6–25	Changed the fifth bulleted item to: ❑ XPC is cleared (’548 **and ’549**)

Page: **Change or Add:**

6–28 Added the following register to Figure 6–2, *Interrupt Flag Register (IFR) Diagram*:

(g) ’549 IFR

15–14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Resvd	BMINT1	BMINT0	BXINT1	BRINT1	HPINT	INT3	TXINT	TRINT	BXINT0	BRINT0	TINT	INT2	INT1	INT0

6–29 Changed the first paragraph to:

Figure 6–3 shows how the ’54x uses a memory-mapped IMR for masking external and internal interrupts. If INTM = 0 in ST1, a 1 in any IMR bit enables the corresponding interrupt. Neither $\overline{\text{NMI}}$ nor $\overline{\text{RS}}$ is included in the IMR, because IMR has no effect on these interrupts. You can read or write to the IMR.

Added the following register to Figure 6–3, *Interrupt Mask Register (IMR) Diagram*:

(g) ’549 IMR

15–14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Resvd	BMINT1	BMINT0	BXINT1	BRINT1	HPINT	INT3	TXINT	TRINT	BXINT0	BRINT0	TINT	INT2	INT1	INT0

6–32 Added the following parenthetical sentence below item number 1 in section 6.10.5, *Phase 3: Execute Interrupt Service Routine (ISR)*:

(The program counter extension register, XPC, does not get pushed to the top of the stack; that is, it does not get saved on the stack.)

6–43 Added Table 6–24, *TMS320C549 Interrupt Locations and Priorities.*

7–28 Added the following ’549 information to Table 7–1, *DARAM Blocks*:

Device	Block Size	Number of Blocks
’549	2K words	4

8–1 Changed the fourth bulleted item to:

- ❏ Host port interface (’542, ’545, ’548, **and ’549**)

Changed the sixth bulleted item to:

- ❏ Buffered serial port (’542, ’543, ’545, ’546, ’548, **and ’549**)

Changed the seventh bulleted item to:

- ❏ Time-division multiplexed (TDM) serial port (’542, ’543, ’548, **and ’549**)

Page: **Change or Add:**

8–3 Changed the label above the memory-mapped register block on the far right of Figure 8–1, *TMS320C54x Peripheral Memory-Mapped Registers*, to:

’548**/549**

8–4 Changed the title of Table 8–1 to include the ’541B device and made the following changes to the table:

*TMS320C541**/541B** Peripheral Memory-Mapped Registers*

Address	Name	Description
33–**57**	–	Reserved
58	**CLKMD**	**Clock modes register (’541B only)**
59–57	**–**	**Reserved**

8–7 Changed the title of Table 8–4 to include the ’545A device and added the following to address 58:

*TMS320C545**/545A** Peripheral Memory-Mapped Registers*

Address	Name	Description
58	CLKMD	Clock mode register **(’545A only)**

8–8 Changed the title of Table 8–5 to include the ’546A device and added the following to address 58:

*TMS320C546**/546A** Peripheral Memory-Mapped Registers*

Address	Name	Description
58	CLKMD	Clock mode register **(’546A only)**

8–10 Added Table 8–7, *TMS320C549 Peripheral-Memory Mapped Registers.*

8–17 Changed the fifth sentence in the paragraph below the bulleted list to:

The ’545A, ’546A, ’548, **and ’549** devices use a software-programmable PLL and are referred to in this section as *LP* devices.

Changed the heading of section 8.4.2 to:

Software-Programmable PLL (Available on TMS320C**541B**/545A/546A/548**/549**)

Page: **Change or Add:**

8–19 Added the following to Table 8–11, *Clock Mode Setting at Reset*:

CLKMD1	CLKMD2	CLKMD3	CLKMD Reset Value	Clock Mode
1	1	1	7000h	Divide-by-2 with internal source†

† **Reserved on ’549**

8–22 Added the ’549 graph to Figure 8–11, *PLL Lockup Time Versus CLKOUT Frequency.*

8–28 Changed the third sentence in the second paragraph to:

In this mode, asynchronous host accesses are resynchronized internally and, in the case of a conflict between a ’54x and a host cycle **(where both accesses are reads or writes)**, the host has access priority and the ’54x waits one cycle.

9–2 Added the following ’549 information to Table 9–1, *Serial Ports on the TMS320C54x Devices*:

Device	Standard Synchronous Serial Ports	Buffered Serial Ports	Time-Division Multiplexed Serial Ports
’549	0	2	1

9–3 Changed the third and fourth sentences in the second paragraph to:

The TDM serial port interface is implemented on the ’542, ’543, ’548, **and ’549** devices. The ’542, ’543, ’545, ’546, ’548, **and ’549** devices include a buffered serial port (BSP) that implements an automatic buffering feature, which greatly reduces CPU overhead required in handling serial data transfers.

Changed the second sentence of the fourth paragraph to:

Therefore, when using the ’542, ’543, ’545, ’546, ’548, **and ’549** devices, you should consult section 9.3.

9–33 Added a signal for the Buffer Misalignment Interrupt (BMINT) to Figure 9–21, *BSP Block Diagram.*

Page: **Change or Add:**

9–40 Changed the last paragraph to the following:

> The ABU also implements CPU interrupts when transmit and receive buffers have been halfway or entirely filled or emptied. These interrupts take the place of the transmit and receive interrupts in standard mode operation (the receive interrupt is the CPU). They are not generated in autobuffering mode. This mechanism features an autodisabling capability which can be used to automatically terminate autobuffering when either the half-of-buffer or bottom-of-buffer boundary is crossed. These features are described in detail later in this section.

9–41 Added a signal for the Buffer Misalignment Interrupt (BMINT) to Figure 9–24, *ABU Block Diagram.*

9–49 Changed the last sentence in the last paragraph to:

> In this example, if the BFSX pulse occurs during the first two BCLKXs **after the transmit section is taken out of reset**, the transmit frame is ignored and BDX is placed in the high-impedance state.

9–50 Removed the graphical representation of data from the BDX line in Figure 9–28, *Standard Mode BSP Initialization Timing*.

9–52 Changed the first sentence of the second paragraph to:

> In Example 9–3 and Example 9–4, the transmit and receive interrupts used are those that the BSP occupies on the ’542, ’543, ’545, ’546, ’548, **and ’549**, the devices that include the BSP.

9–53 Added section 9.3.4, *Buffer Misalignment Interrupt (BMINT)* – ’549 Only.

10–2 Changed the heading row in Table 10–1, *Key External Interface Signals*, to include the ’549 device as follows:

Signal Name	’541’542, ’543, ’545, ’546	’548, ’549	Description
A0–A15	15–0	22–0	Address bus

10–5 Changed the first sentence of the first paragraph in section 10.3.1, *Wait State Generator, to:*

> The software-programmable wait-state generator can extend external bus cycles by up to seven machine cycles **(14 machine cycles on ’549 devices)**, providing a convenient means to interface the ’54x to slower external devices.

Page: **Change or Add:**

10–5 Changed the last sentence of the third paragraph in section 10.3.1, *Wait State Generator,* to:

> These fields are shown in Figure 10–2 and described in Table 10–2. The ’548 **and ’549** are described in Table 10–3.

Changed the note below Figure 10–2, *Software Wait-State Register (SWWSR) Diagram*, to:

> †XPA bit on ’548 **and ’549** only

Added the following sentence below Figure 10–2, *Software Wait-State Register (SWWSR) Diagram*:

> The ’549 has an extra bit (software wait-state multiplier, SWWSM) that resides in XSWWR, which is memory-mapped to address 002Bh in data space.

10–6 Added Figure 10–3, *Extended Software-Wait State Register (XSWWR) Diagram.*

Added the following sentence below figure 10–3:

> When SWWSM is set to 1, the wait states are multiplied by two, extending the number of wait states from 7 to 14.

Changed the bit 15 function in Table 10–2, *Software Wait-State Register (SWWSR) Bit Summary,* to:

Bit	Name	Reset Value	Function
15	Reserved	0	Reserved. In the ’548 **and ’549**, this bit changes the operation of the program fields (see Table 10–3).

Changed the title of Table 10–3 to:

> *TMS320C548**/549** Software Wait-State Register (SWWSR) Bit Summary*

10–11 Changed the third bulleted item to:

- A program-memory read followed by another program-memory read from a different page (with the ’548 **and ’549**)

10–18 Changed the I/O write (increased the width) at D(15–0) in Figure 10–12, *Parallel I/O Operation for Read-Write-Read (I/O-Space Wait States).*

Page: **Change or Add:**

A-1 Added the following term and definition to the table:

BMINT1, BMINT0	**Buffer misalignment interrupt**

A-4 Added the following register to Figure A–5, *Interrupt Flag Register (IFR) Diagram*:

(g) ’549 IFR

15–14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Resvd	BMINT1	BMINT0	BXINT1	BRINT1	HPINT	INT3	TXINT	TRINT	BXINT0	BRINT0	TINT	INT2	INT1	INT0

A-5 Added the following register to Figure A–6, *Interrupt Mask Register (IMR) Diagram*:

(g) ’549 IMR

15–14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Resvd	BMINT1	BMINT0	BXINT1	BRINT1	HPINT	INT3	TXINT	TRINT	BXINT0	BRINT0	TINT	INT2	INT1	INT0

A-6 Changed the note at the bottom of Figure A–9, *Software Wait-State Register (SWWSR),* to:

†XPA bit on ’548 **and ’549** only

Added Figure A–10*, Extended Software Wait-State Register (XSWWR).*

C-6 Added ’549 to the list of devices and Micro Star™ BGA to the list of package types in Figure C–1, *TMS320C54x Device Nomenclature.*

E-4 Added the following glossary definition below *block-repeat start address register (RSA)*:

BMINT: See *buffer misalignment interrupt*.

E-5 Added the following glossary definition below BSPCE, BSPCE0, BSPCE1:

buffer misalignment interrupt (BMINT): A ’549 feature that detects potential error conditions and indicates lost words on a serial port interface.

Index

B

C

D

E

F

G

H

I

J

L

M

N

O

P

R

S

T

U

V

W

X

Z