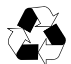
TMS320C62xx CPU and Instruction Set Reference Guide

Literature Number: SPRU189B
Manufacturing Part Number: D426008-9761 revision A
July 1997



Printed on Recycled Paper

IMPORTANT NOTICE

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

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

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

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

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

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

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

Copyright © 1997, Texas Instruments Incorporated

Preface

Read This First

About This Manual

This reference guide describes the CPU architecture, pipeline, instruction set, and interrupts for the TMS320C62xx digital signal processors (DSPs). Unless otherwise specified, all references to the ’C6x refer to the TMS320C6x generation of DSPs and ’C62xx refer to the TMS320C62xx DSPs in the ’C6x generation.

How to Use This Manual

Use this manual as a reference for the architecture of the TMS320C62xx CPU. First-time readers should read Chapter 1 for general information about TI DSPs, the features of the TMS320C62xx, and the applications for which the TMS320C62xx is best suited.

Read chapters 2, 4, and 5 to grasp the concepts of the architecture. Chapter 3 contains detailed information about each instruction and is best used as reference material; however, you may want to read sections 3.1 through 3.8 for general information about the instruction set and to understand the instruction descriptions, then browse through Chapter 3 to familiarize yourself with the instructions.

The following table gives chapter references for specific information:

If you are looking for information about:	Turn to these chapters:
Addressing modes	Chapter 3, *Instruction Set*
Conditional operations	Chapter 3, *Instruction Set*
Control registers	Chapter 2, *CPU Overview*
CPU architecture	Chapter 2, *CPU Overview*
CPU data paths	Chapter 2, *CPU Overview*
Delay slots	Chapter 3, *Instruction Set* Chapter 4, *Pipeline Operation*

If you are looking for information about:	**Turn to these chapters:**
General-purpose register files	Chapter 2, *CPU Overview*
Instruction set	Chapter 3, *Instruction Set*
Interrupt control registers	Chapter 5, *Interrupts*
Interrupts	Chapter 5, *Interrupts*
Parallel operations	Chapter 3, *Instruction Set*
Pipeline operation	Chapter 4, *Pipeline Operation*
Pipeline phases	Chapter 4, *Pipeline Operation*
Reset	Chapter 5, *Interrupts*

If you are interested in topics that are not listed here, check *Related Documentation From Texas Instruments*, page v, for brief descriptions of other 'C62xx-related books that are available.

Notational Conventions

This document uses the following conventions:

- Program listings and program examples are shown in a `special font`. Here is a sample program listing:

```
LDW .D1    *A0,A1
ADD .L1    A1,A2,A3
NOP        3
MPY .M1    A1,A4,A5
```

- To help you easily recognize instructions and parameters throughout the book, instructions are in **bold face** and parameters are in *italics* (except in program listings).

- In instruction syntaxes, portions of a syntax that are in **bold** should be entered as shown; portions of a syntax that are in *italics* describe the *type* of information that should be entered. Here is an example of an instruction:

  MPY *src1,src2,dst*

  MPY is the instruction mnemonic. When you use **MPY**, you must supply two source operands (*src1* and *src2*) and a destination operand (*dst*) of appropriate types as defined in Chapter 3, *Instruction Set*.

  Although the instruction mnemonic (**MPY** in this example) is in capital letters, the 'C6x assembler *is not case sensitive* — it can assemble mnemonics entered in either upper or lower case.

- Square brackets, [and], and parentheses, (and), are used to identify optional items. If you use an optional item, you must specify the information within brackets or parentheses; however, you do not enter the brackets or parentheses themselves. Here is an example of an instruction that has optional items.

  [*label*] **EXTU** (*.unit*) *src2, csta, cstb, dst*

  The **EXTU** instruction is shown with a label and several parameters. The [*label*] and the parameter (*.unit*) are optional. The parameters *src2, csta, cstb,* and *dst* are not optional.

- Throughout this book MSB means *most significant bit* and LSB means *least significant bit*.

Related Documentation From Texas Instruments

The following books describe the TMS320C6x generation 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.

TMS320C62xx Technical Brief (literature number SPRU197) gives an introduction to the 'C62xx digital signal processor, development tools, and third-party support.

TMS320C62xx Peripherals Reference Guide (literature number SPRU190) describes common peripherals available on the TMS320C62xx digital signal processors. This book includes information on the internal data and program memories, the external memory interface (EMIF), the host port, serial ports, direct memory access (DMA), clocking and phase-locked loop (PLL), and the power-down modes.

TMS320C62xx Programmer's Guide (literature number SPRU198) describes ways to optimize C and assembly code and includes application program examples.

TMS320C6x Assembly Language Tools User's Guide (literature number SPRU186) 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 'C6x generation of devices.

TMS320C6x Optimizing C Compiler User's Guide (literature number SPRU187) describes the 'C6x C compiler. This C compiler accepts ANSI standard C source code and produces assembly language source code for the 'C6x generation of devices. This book also describes the assembly optimizer, which helps you optimize your assembly code.

TMS320C6x C Source Debugger User's Guide (literature number SPRU188) tells you how to invoke the 'C6x simulator and emulator versions of the C source debugger interface. This book discusses various aspects of the debugger, including command entry, code execution, data management, breakpoints, profiling, and analysis.

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.

TMS320C6201 Digital Signal Processor Data Sheet (literature number SPRS051) describes the features of the TMS320C6xx and provides pinouts, electrical specifications, and timings for the device.

Trademarks

XDS510, VelociTI, and 320 Hotline On-line are trademarks of Texas Instruments Incorporated.

Windows and Windows NT are registered trademarks of Microsoft Corporation.

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: comments@books.sc.ti.com

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

Contents

1 Introduction . . . 1-1

Summarizes the features of the TMS320 family of products and presents typical applications. Describes the TMS320C62xx DSP and lists its key features.

1.1 TMS320 Family Overview . . . 1-2
1.1.1 History of TMS320 DSPs . . . 1-2
1.1.2 Typical Applications for the TMS320 Family . . . 1-2
1.2 Overview of the TMS320C6x Generation of Digital Signal Processors . . . 1-4
1.3 Features and Options of the TMS320C62xx . . . 1-4
1.4 TMS320C62xx Architecture . . . 1-6
1.4.1 Central Processing Unit (CPU) . . . 1-7
1.4.2 Internal Memory . . . 1-8
1.4.3 Peripherals . . . 1-8

2 CPU Data Paths and Control . . . 2-1

Summarizes the TMS320C62xx architecture and describes the primary components of the CPU.

2.1 CPU Data Paths . . . 2-2
2.1.1 General-Purpose Register Files . . . 2-2
2.1.2 Functional Units . . . 2-3
2.1.3 Register File Cross Paths . . . 2-5
2.1.4 Memory, Load, and Store Paths . . . 2-5
2.1.5 Data Address Paths . . . 2-5
2.2 Control Register File . . . 2-6
2.3 Addressing Mode Register (AMR) . . . 2-7
2.4 Control Status Register (CSR) . . . 2-9

3 Instruction Set . . . 3-1

Describes the TMS320C62xx instruction set, including examples of each instruction. Provides information about addressing modes, resource constraints, parallel operations, and conditional operations.

3.1 Instruction Operation and Execution Notations . . . 3-2
3.2 Mapping Between Instructions and Functional Units . . . 3-4
3.3 TMS320C62xx Opcode Map . . . 3-7
3.4 Delay Slots . . . 3-9
3.5 Parallel Operations . . . 3-10

3.5.1 Example Parallel Code 3-12
3.5.2 Branching Into the Middle of an Execute Packet 3-12
3.6 Conditional Operations 3-13
3.7 Resource Constraints 3-14
3.7.1 Constraints on Instructions Using the Same Functional Unit 3-14
3.7.2 Constraints on Cross Paths (1X and 2X) 3-14
3.7.3 Constraints on Loads and Stores 3-15
3.7.4 Constraints on Long (40-Bit) Data 3-15
3.7.5 Constraints on Register Reads 3-16
3.7.6 Constraints on Register Writes 3-16
3.8 Addressing Modes 3-18
3.8.1 Linear Addressing Mode 3-18
3.8.2 Circular Addressing Mode 3-18
3.8.3 Syntax for Load/Store Address Generation 3-20
3.9 Individual Instruction Descriptions 3-21

4 TMS320C62xx Pipeline 4-1

Describes pipeline phases, pipeline operation, and pipeline discontinuities for the TMS320C62xx CPU pipeline.

4.1 Pipeline Operation Overview 4-2
4.1.1 Fetch 4-3
4.1.2 Decode 4-4
4.1.3 Execute 4-5
4.1.4 Summary of Pipeline Operation 4-6
4.2 Pipeline Execution of Instruction Types 4-10
4.2.1 Single-Cycle Instructions 4-11
4.2.2 Multiply Instructions 4-11
4.2.3 Store Instructions 4-12
4.2.4 Load Instructions 4-14
4.2.5 Branch Instructions 4-15
4.3 Performance Considerations 4-17
4.3.1 Pipeline Operation With Multiple Execute Packets in a Fetch Packet 4-17
4.3.2 Multicycle NOPs 4-18
4.3.3 Memory Considerations 4-21

5 Interrupts 5-1

Describes the TMS320C62xx interrupts, including reset and nonmaskable interrupts (NMI) and explains interrupt control, detection, and processing.

5.1 Overview of Interrupts 5-2
5.1.1 Types of Interrupts and Signals Used 5-2
5.1.2 Interrupt Service Table (IST) 5-5
5.1.3 Summary of Interrupt Control Registers 5-10
5.2 Globally Enabling and Disabling Interrupts (Control Status Register–CSR) 5-11

5.3 Individual Interrupt Control 5-13
5.3.1 Enabling and Disabling Interrupts (Interrupt Enable Register–IER) 5-13
5.3.2 Status of, Setting, and Clearing Interrupts (Interrupt Flag, Set, and Clear Registers – IFR, ISR, ICR) 5-14
5.3.3 Returning From Interrupt Servicing 5-16
5.4 Interrupt Detection and Processing 5-18
5.4.1 Setting the Interrupt Flag – Nonreset 5-18
5.4.2 Conditions for Processing an Interrupt – Nonreset 5-18
5.4.3 Actions Taken During Interrupt Processing – Nonreset 5-20
5.4.4 Setting the Interrupt Flag – $\overline{\text{RESET}}$ 5-21
5.4.5 Actions Taken During Interrupt Processing – $\overline{\text{RESET}}$ 5-22
5.5 Performance Considerations 5-23
5.5.1 General Performance 5-23
5.5.2 Pipeline Interaction 5-23
5.6 Programming Considerations 5-24
5.6.1 Single Assignment Programming 5-24
5.6.2 Nested Interrupts 5-25
5.6.3 Manual Interrupt Processing 5-25
5.6.4 Traps 5-26

A Glossary A-1
Defines terms and abbreviations used throughout this book.

Figures

1–1. TMS320C62xx Block Diagram 1-6
2–1. Storage Scheme for 40-Bit Data in a Register Pair 2-3
2–2. TMS320C62xx CPU Data Paths 2-4
2–3. Addressing Mode Register (AMR) 2-7
2–4. Control Status Register (CSR) 2-9
3–1. TMS320C62xx Opcode Map 3-7
3–2. Basic Format of a Fetch Packet 3-10
3–3. Examples of the Detectability of Write Conflicts by the Assembler 3-17
4–1. Pipeline Stages 4-2
4–2. Fetch Phases of the Pipeline 4-3
4–3. Decode Phases of the Pipeline 4-4
4–4. Execute Phases of the Pipeline and Functional Block Diagram 4-5
4–5. Pipeline Phases 4-6
4–6. Pipeline Operation: One Execute Packet per Fetch Packet 4-6
4–7. Functional Block Diagram Based on Pipeline Phases 4-8
4–8. Single-Cycle Instruction Phases 4-11
4–9. Single-Cycle Execution Block Diagram 4-11
4–10. Multiply Instruction Phases 4-11
4–11. Multiply Execution Block Diagram 4-12
4–12. Store Instruction Phases 4-12
4–13. Store Execution Block Diagram 4-13
4–14. Load Instruction Phases 4-14
4–15. Load Execution Block Diagram 4-14
4–16. Branch Instruction Phases 4-15
4–17. Branch Execution Diagram 4-16
4–18. Pipeline Operation: Fetch Packets With Different Numbers of Execute Packets 4-18
4–19. Multicycle NOP in an Execute Packet 4-19
4–20. Branching and Multicycle NOPs 4-20
4–21. Pipeline Phases Used During Memory Accesses 4-21
4–22. Program and Data Memory Stalls 4-22
4–23. 4-Bank Interleaved Memory 4-23
4–24. 4-Bank Interleaved Memory With Two Memory Spaces 4-24
5–1. Interrupt Service Table 5-5
5–2. Interrupt Service Fetch Packet 5-6
5–3. IST With Branch to Additional Interrupt Service Code Located Outside the IST 5-7
5–4. Interrupt Service Table Pointer (ISTP) 5-8

5–5. Control Status Register (CSR) 5-11
5–6. Interrupt Enable Register (IER) 5-13
5–7. Interrupt Flag Register (IFR) 5-14
5–8. Interrupt Set Register (ISR) 5-15
5–9. Interrupt Clear Register (ICR) 5-15
5–10. NMI Return Pointer (NRP) 5-16
5–11. Interrupt Return Pointer (IRP) 5-17
5–12. Interrupt Detection and Processing: Pipeline Operation–Nonreset 5-19
5–13. Interrupt Detection and Processing: Pipeline Operation–$\overline{\text{RESET}}$ 5-21

Tables

1–1. Typical Applications for the TMS320 DSPs 1-3
2–1. Long (40-Bit) Register Pairs 2-2
2–2. Functional Units and Operations Performed 2-3
2–3. Control Registers 2-6
2–4. Addressing Mode Field Encoding 2-7
2–5. Block Size Calculations 2-8
2–6. Control Status Register: Fields, Read/Write Status, and Function 2-9
3–1. Instruction Operation and Execution Notations 3-2
3–2. Instruction to Functional Unit Mapping 3-4
3–3. Functional Unit to Instruction Mapping 3-5
3–4. Delay Slot Summary 3-9
3–5. Registers That Can Be Tested by Conditional Operations 3-13
3–6. Indirect Address Generation for Load/Store 3-20
3–7. Relationships Between Operands, Operand Size, Signed/Unsigned, Functional Units, and Opfields for Example Instruction (ADD Instruction) 3-23
3–8. Program Counter Values for Branch Using a Displacement Example 3-35
3–9. Program Counter Values for Branch Using a Register Example 3-37
3–10. Program Counter Values for B IRP Example 3-39
3–11. Program Counter Values for B NRP Example 3-41
3–12. Data Types Supported by Loads 3-60
3–13. Address Generator Options 3-60
3–14. Data Types Supported by Loads 3-64
3–15. Register Addresses for Accessing the Control Registers 3-75
3–16. Data Types Supported by Stores 3-106
3–17. Address Generator Options 3-106
3–18. Data Types Supported by Stores 3-109
4–1. Operations Occurring During Pipeline Phases 4-7
4–2. Execution Stage Length Description for Each Instruction Type 4-10
4–3. Program Memory Accesses Versus Data Load Accesses 4-21
4–4. Loads in Pipeline From Example 4–3 4-23
5–1. Interrupt Priorities 5-3
5–2. ISTP Fields 5-8
5–3. Interrupt Control Registers 5-10
5–4. CSR Interrupt Control Field Descriptions 5-11

Examples

3–1. Fully Serial *p*-Bit Pattern in a Fetch Packet 3-11
3–2. Fully Parallel *p*-Bit Pattern in a Fetch Packet 3-11
3–3. Partially Serial *p*-Bit Pattern in a Fetch Packet 3-12
3–4. LDW in Circular Mode 3-19
3–5. ADDAH in Circular Mode 3-19
4–1. Execute Packet in Dispatch (DP) Phase in Figure 4–7 4-9
4–2. Execute Packet in E1 Phase of Execution in Figure 4–7 4-9
4–3. Load From Memory Banks 4-23
5–1. Relocation of Interrupt Service Table 5-9
5–2. Code Sequence to Disable Maskable Interrupts Globally 5-12
5–3. Code Sequence to Enable Maskable Interrupts Globally 5-12
5–4. Code Sequence to Enable an Individual Interrupt–INT9 5-14
5–5. Code Sequence to Disable an Individual Interrupt–INT9 5-14
5–6. Code to Set an Individual Interrupt (INT6) and Read the Flag Register 5-15
5–7. Code to Clear an Individual Interrupt (INT6) and Read the Flag Register 5-15
5–8. Code to Return from NMI 5-16
5–9. Code to Return from a Maskable Interrupt 5-17
5–10. Code Without Single Assignment: Multiple Assignment of A1 5-24
5–11. Code Using Single Assignment 5-24
5–12. Manual Interrupt Processing 5-25
5–13. Code Sequence to Invoke a Trap 5-26
5–14. Code Sequence for Trap Return 5-26

Chapter 1

Introduction

The TMS320C6x generation of digital signal processors is part of the TMS320 family of digital signal processors (DSPs). The TMS320C62xx devices are fixed-point DSPs in the TMS320C6x generation. The TMS320C62xx is the first DSP to use the VelociTI™ architecture, a high-performance, advanced VLIW (very long instruction word) architecture, making the 'C62xx an excellent choice for multichannel and multifunction applications.

The 'C62xx's VelociTI architecture makes it the first off-the-shelf DSP to use advanced VLIW to achieve high performance through increased instruction-level parallelism. A traditional VLIW architecture consists of multiple execution units running in parallel, performing multiple instructions during a single clock cycle. Parallelism is the key to extremely high performance, taking these DSPs well beyond the performance capabilities of traditional superscalar designs. VelociTI is a highly deterministic architecture, having few restrictions on how or when instructions are fetched, executed, or stored. It is this architectural flexibility that is key to the breakthrough efficiency levels of the 'C6x compiler. VelociTI's advanced features include:

- Instruction packing: reduced code size
- All instructions can operate conditionally: flexibility of code
- Variable-width instructions: flexibility of data types
- Fully pipelined branches: zero-overhead branching

Topic **Page**

1.1 TMS320 Family Overview 1-2

1.2 Overview of the TMS320C6x Generation of Digital Signal Processors 1-4

1.3 Features and Options of the TMS320C62xx 1-4

1.4 TMS320C62xx Architecture 1-6

1.1 TMS320 Family Overview

The TMS320 family consists of fixed-point, floating-point, and multiprocessor digital signal processors (DSPs). TMS320 DSPs have an architecture designed specifically for real-time signal processing.

1.1.1 History 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 many generations: ’C1x, ’C2x, ’C2xx, ’C5x, and ’C54x fixed-point DSPs; ’C3x and ’C4x floating-point DSPs, and ’C8x multiprocessor DSPs. Now there is a new generation of DSPs, the TMS320C6x generation, with performance and features that are reflective of Texas Instruments commitment to lead the world in DSP solutions.

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 adaptable approaches to traditional signal-processing problems. 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
Global positioning	Radar detectors	Servo control
Navigation	Solid-state answering machines	
Vibration analysis		
Voice commands		
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 56 600-bps modems	Faxing	Speaker verification
Adaptive equalizers	Future terminals	Speech enhancement
ADPCM transcoders	Line repeaters	Speech recognition
Base stations	Personal communications systems (PCS)	Speech synthesis
Cellular telephones		Speech vocoding
Channel multiplexing	Personal digital assistants (PDA)	Text-to-speech
Data encryption	Speaker phones	Voice mail
Digital PBXs	Spread spectrum communications	
Digital speech interpolation (DSI)	Digital subscriber loop (xDSL)	
DTMF encoding/decoding	Video conferencing	
Echo cancellation	X.25 packet switching	

1.2 Overview of the TMS320C6x Generation of Digital Signal Processors

With a performance of up to 1600 million instructions per second (MIPS) and an efficient C compiler, the TMS320C6x DSPs give system architects unlimited possibilities to differentiate their products. High performance, ease of use, and affordable pricing make the TMS320C6x generation the ideal solution for multichannel, multifunction applications, such as:

- Pooled modems
- Wireless base stations
- Remote access servers (RAS)
- Digital subscriber loop (DSL) systems
- Cable modems
- Multichannel telephony systems

The TMS320C6x generation is also an ideal solution for exciting new applications; for example:

- Personalized home security with face and hand/fingerprint recognition
- Advanced cruise control with GPS navigation and accident avoidance
- Remote medical diagnostics

1.3 Features and Options of the TMS320C62xx

At 200 MHz, the 'C62xx devices operate at a 5-ns cycle time, executing up to eight 32-bit instructions every cycle. The device's core CPU consists of 32 general-purpose registers of 32-bit word length and eight functional units:

- Two multipliers
- Six ALUs

The 'C62xx has a complete set of optimized development tools, including an efficient C compiler, an assembly optimizer for simplified assembly-language programming and scheduling, and a Windows™ based debugger interface for visibility into source code execution characteristics. A hardware emulation board, compatible with the TI XDS510™ emulator interface, is also available. This tool complies with IEEE Standard 1149.1–1990, IEEE Standard Test Access Port and Boundary-Scan Architecture.

Features of the ’C62xx include:

- Advanced VLIW CPU with eight functional units, including two multipliers and six arithmetic units
  - Executes up to eight instructions per cycle for up to ten times the performance of typical DSPs
  - Allows designers to develop highly effective RISC-like code for fast development time
- Instruction packing
  - Gives code size equivalence for eight instructions executed serially or in parallel
  - Reduces code size, program fetches, and power consumption.
- All instructions execute conditionally.
  - Reduces costly branching
  - Increases parallelism for higher sustained performance
- Code executes as programmed on independent functional units.
  - Industry’s most efficient C compiler on DSP benchmark suite
  - Industry’s first assembly optimizer for fast development and improved parallelization
- 8/16/32-bit data support, providing efficient memory support for a variety of applications
- 40-bit arithmetic options add extra precision for vocoders and other computationally intensive applications
- Saturation and normalization provide support for key arithmetic operations.
- Field manipulation and instruction extract, set, clear, and bit counting support common operation found in control and data manipulation applications.

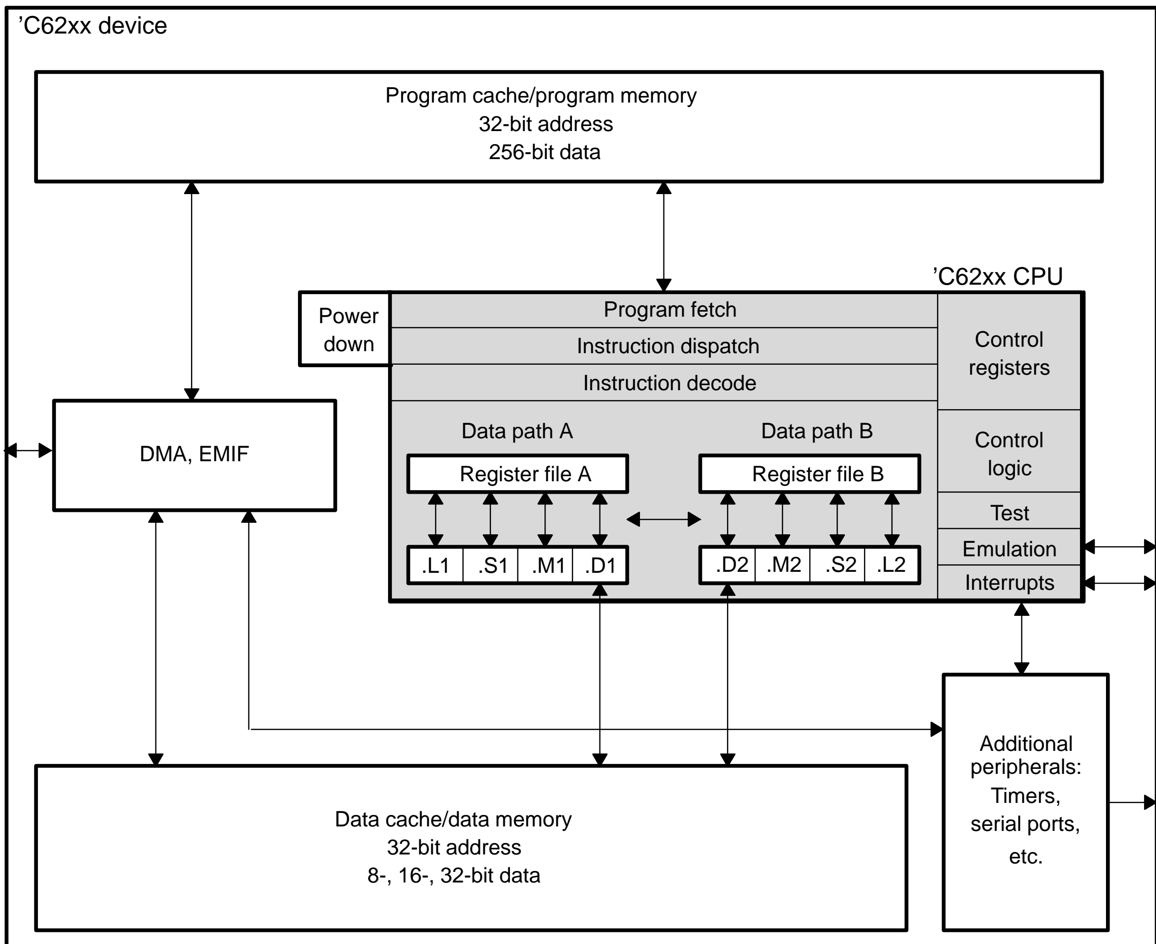
A variety of memory and peripheral options are available for the ’C62xx:

- Large on-chip RAM for fast algorithm execution
- 32-bit external memory interface supports SDRAM, SBSRAM, SRAM, and other asynchronous memories, for a broad range of external memory requirements and maximum system performance
- 16-bit host port for host to access ’C62xx memory and peripherals
- Multichannel DMA controller
- Multichannel serial port(s)
- 32-bit timer(s)

1.4 TMS320C62xx Architecture

Figure 1–1 is the block diagram for the TMS320C62xx DSPs. ’C62xx DSPs are based on the ’C62xx CPU, shown in the right center of the figure. ’C62xx devices come with program memory, which, on some devices, can be used as a program cache. The devices also have varying sizes of data memory. Peripherals such as a direct memory access (DMA) controller, power-down logic, and external memory interface (EMIF) usually come with the CPU, while peripherals such as serial ports and host ports are on only certain devices. Check the data sheet for your device to determine the specific peripheral configurations you have.

Figure 1–1. TMS320C62xx Block Diagram



1.4.1 Central Processing Unit (CPU)

The ’C62xx CPU, shaded in Figure 1–1, is common to all the ’C62xx devices. The CPU contains:

- Program fetch unit
- Instruction dispatch unit
- Instruction decode unit
- Two data paths, each with four functional units
- 32 32-bit registers
- Control registers
- Control logic
- Test, emulation, and interrupt logic

The program fetch, instruction dispatch, and instruction decode units can deliver up to eight 32-bit instructions to the functional units every CPU clock cycle. The processing of instructions occurs in each of the two data paths (A and B), each of which contains four functional units (.L, .S, .M, and .D) and 16 32-bit general-purpose registers. The data paths are described in more detail in section 2.1. A control register file provides the means to configure and control various processor operations. To understand how instructions are fetched, dispatched, decoded, and executed in the data path, see Chapter 4, *TMS320C62xx Pipeline*.

1.4.2 Internal Memory

The ’C62xx has a 32-bit, byte-addressable address space. Internal (on-chip) memory is organized in separate data and program spaces. When off-chip memory is used, these spaces are unified on most devices to a single memory space via the external memory interface (EMIF).

The ’C62xx has two internal ports to access data memory, each with 32 bits of data and a 32-bit byte address reach. The ’C62xx has a single port to access program memory, with an instruction-fetch width of 256 bits and a 30-bit word address, equivalent to a 32-bit byte address.

1.4.3 Peripherals

The following peripheral modules can complement the CPU on the ’C62xx DSPs. Your particular device has a subset of these peripherals but may not have all of them.

- Serial ports
- Timers
- External memory interface (EMIF) that supports synchronous and asynchronous SRAM and synchronous DRAM
- DMA controller
- Host port
- Power-down logic that can halt CPU activity, peripheral activity, and PLL activity to reduce power consumption

Chapter 2

CPU Data Paths and Control

This chapter provides an overview of the ’C62xx architecture. It focuses on the CPU, providing information about the data paths and control registers.

Topic **Page**

2.1 CPU Data Paths 2-2

2.2 Control Register File 2-6

2.3 Addressing Mode Register (AMR) 2-7

2.4 Control Status Register (CSR) 2-9

2.1 CPU Data Paths

Figure 2–2 on page 2-4 shows the ’C62xx CPU data paths, which consist of:

- Two general-purpose register files (A and B)
- Eight functional units (.L1, .L2, .S1, .S2, .M1, .M2, .D1, and .D2)
- Two load-from-memory paths (LD1 and LD2)
- Two store-to-memory paths (ST1 and ST2)
- Two register file cross paths (1X and 2X)

2.1.1 General-Purpose Register Files

There are two general-purpose register files (A and B) in the ’C62xx data paths. Each of these files contains 16 32-bit registers (A0–A15 for file A and B0–B15 for file B). The general purpose registers can be used for data, data address pointers, or condition registers.

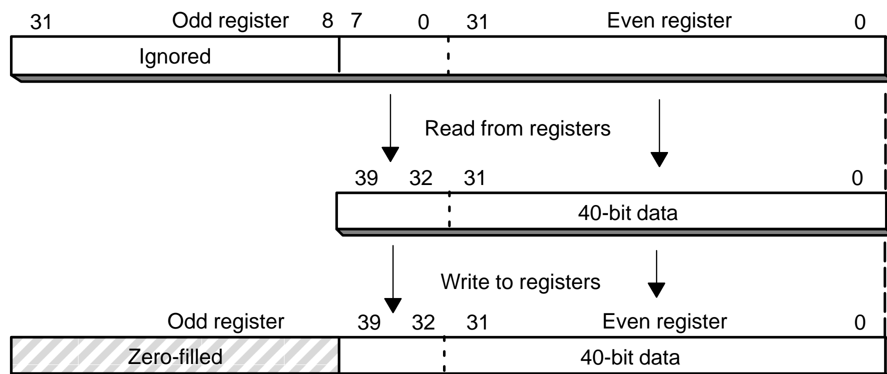
The general-purpose register file supports 32- and 40-bit data. Thirty-two-bit data can be contained in any general-purpose register. Forty-bit data is contained across two registers; the 32 LSBs of the data are placed in an even register and the remaining 8 MSBs are placed in the 8 LSBs of the next upper register (which is always an odd register). There are 16 valid register pairs for 40-bit data as shown in Table 2–1. In assembly language syntax, the register pairs are denoted by a colon between the register names, and the odd register is specified first.

Table 2–1. Long (40-Bit) Register Pairs

Register File	
A	**B**
A1:A0	B1:B0
A3:A2	B3:B2
A5:A4	B5:B4
A7:A6	B7:B6
A9:A8	B9:B8
A11:A10	B11:B10
A13:A12	B13:B12
A15:A14	B15:B14

Figure 2–1 illustrates the register storage scheme for 40-bit data. Operations requiring a long input ignore the 24 MSBs of the odd register. Operations producing a long result zero-fill the 24 MSBs of the odd register. The even register is encoded in the opcode.

Figure 2–1. Storage Scheme for 40-Bit Data in a Register Pair



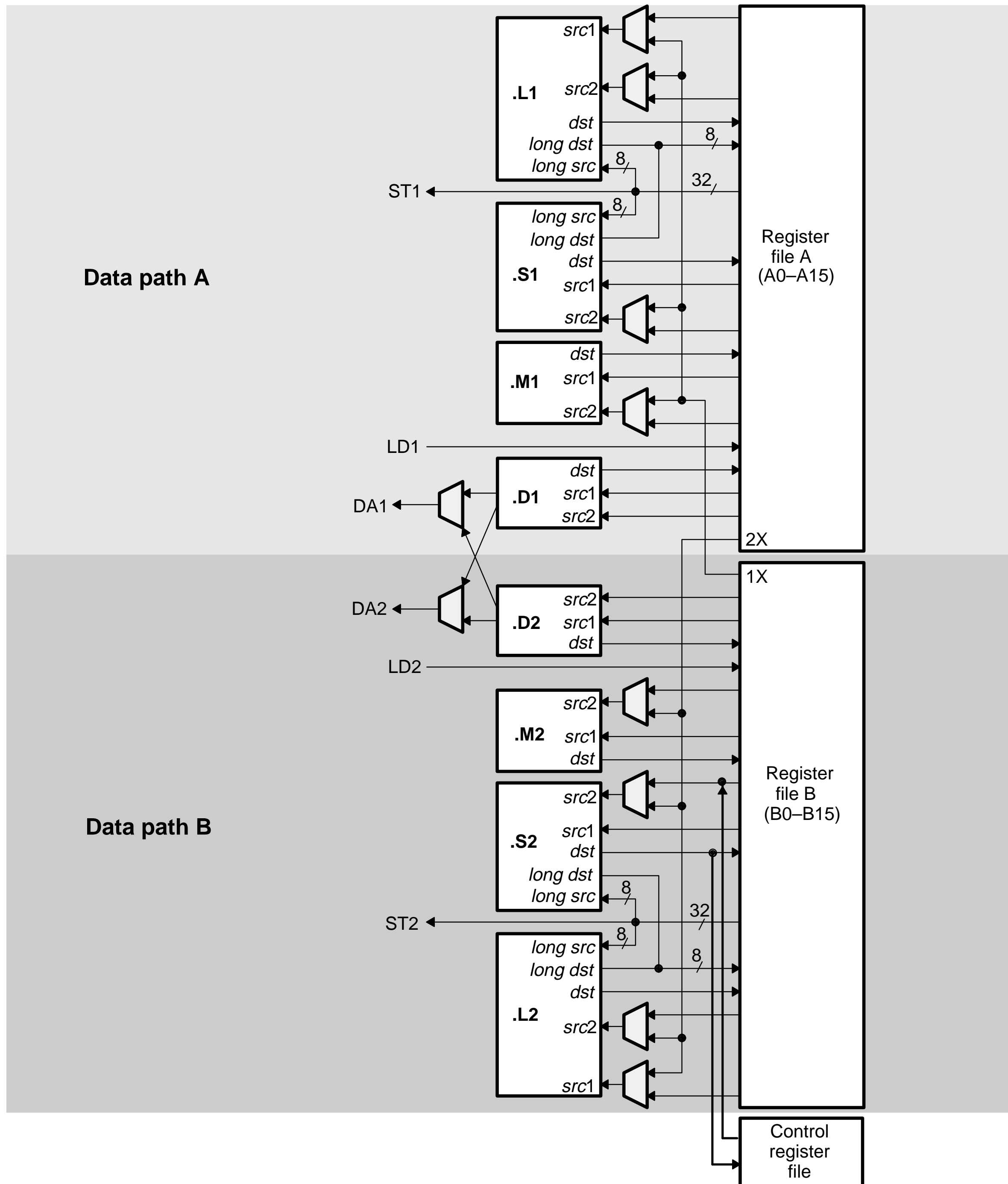
2.1.2 Functional Units

The eight functional units in the 'C62xx data paths can be divided into two groups of four; each functional unit in one data path is almost identical to the corresponding unit in the other data path. The functional units are described in Table 2–2.

Table 2–2. Functional Units and Operations Performed

Functional Unit	Operations
.L Unit (.L1,.L2)	32/40-bit arithmetic and compare operations Leftmost 1 or 0 bit counting for 32 bits Normalization count for 32 and 40 bits 32-bit logical operations
.S Unit (.S1, .S2)	32-bit arithmetic operations 32/40-bit shifts and 32-bit bit-field operations 32-bit logical operations Branches Constant generation Register transfers to/from the control register file (.S2 only)
.M Unit (.M1, .M2)	16 × 16 bit multiply operations
.D Unit (.D1, .D2)	32-bit add, subtract, linear and circular address calculation Loads and stores with a 5-bit constant offset Loads and stores with 15-bit constant offset (.D2 only)

Figure 2–2. TMS320C62xx CPU Data Paths



Most data lines in the CPU support 32-bit operands, and some support long (40-bit) operands. Each functional unit has its own 32-bit write port into a general-purpose register file. All units ending in 1 (for example, .L1) write to register file A and all units ending in 2 write to register file B. Each functional unit has two 32-bit read ports for source operands *src*1 and *src*2. Four units (.L1, .L2, .S1, .S2) have an extra 8-bit wide port for 40-bit long writes as well as an 8-bit input for 40-bit long reads. Because each unit has its own 32-bit write port, all eight units can be used in parallel every cycle.

2.1.3 Register File Cross Paths

Each functional unit reads directly from and writes directly to the register file within its own data path. That is, the .L1, .S1, .D1, and .M1 units write to register file A and the .L2, .S2, .D2, and .M2 units write to register file B. The register files are connected to the opposite-side register file's functional units via the 1X and 2X cross paths. These cross paths allow functional units from one data path to access a 32-bit operand from the opposite side's register file. The 1X cross path allows data path A's functional units to read their source from register file B and the 2X cross path allows data path B's functional units to read their source from register file A.

Six of the functional units have access to the opposite side's register file via a cross path. The .M1, .M2, .S1, and .S2 units' *src2* inputs are multiplex-selectable between the cross path and the same side register file. The .L1 and .L2 units' *src1* and *src2* inputs are also multiplex-selectable between the cross path and the same side register file.

There are only two cross paths in the 'C62xx CPU, 1X and 2X. This limits one source read from each data path's opposite register file per cycle, or two cross-path source reads per cycle.

2.1.4 Memory, Load, and Store Paths

There are two 32-bit paths for loading data from memory to the register file: one (LD1) for register file A, and one (LD2) for register file B. There are also two 32-bit paths, ST1 and ST2, for storing register values to memory from each register file. The store paths are shared with the .L and .S long read paths.

2.1.5 Data Address Paths

Locate the .D1 and .D2 units in Figure 2–2. The data address paths (DA1 and DA2) coming out of the .D units allow data addresses generated from one register file to support loads and stores to memory from the other register file.

2.2 Control Register File

Locate the control register file in Figure 2–2. One unit (.S2) can read from and write to the control register file. Table 2–3 lists the control registers contained in the control register file and describes each. If more information is available on a control register, the table lists where to look for that information. Each control register is accessed by the **MVC** instruction. See the **MVC** instruction description in Chapter 3, *Instruction Set*, for information on how to use this instruction.

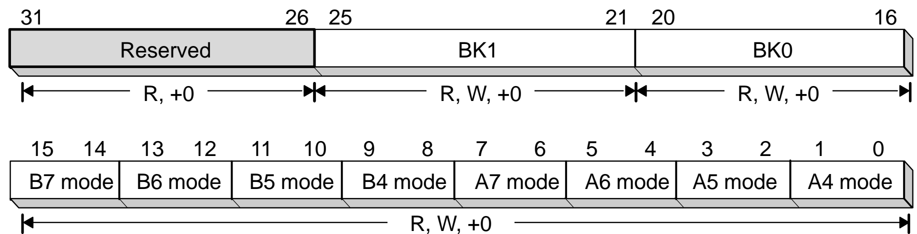
Table 2–3. Control Registers

Register			
Abbreviation	**Name**	**Description**	**Page No.**
AMR	Addressing mode register	Specifies whether to use linear or circular addressing for one of eight registers; also contains sizes for circular addressing	2-7
CSR	Control status register	Contains the global interrupt enable bit, cache control bits, and other miscellaneous control and status bits	2-9
IFR	Interrupt flag register	Displays status of interrupts	5-14
ISR	Interrupt set register	Allows you to set pending interrupts manually	5-14
ICR	Interrupt clear register	Allows you to clear pending interrupts manually	5-14
IER	Interrupt enable register	Allows enabling/disabling of individual interrupts	5-13
ISTP	Interrupt service table pointer	Points to the beginning of the interrupt service table	5-8
IRP	Interrupt return pointer	Contains the address to be used to return from a maskable interrupt	5-16
NRP	Nonmaskable interrupt return pointer	Contains the address to be used to return from a nonmaskable interrupt	5-16
PCE1	Program counter	Contains the address of the fetch packet that contains the execute packet in the E1 pipeline stage	

2.3 Addressing Mode Register (AMR)

Figure 2–3 shows the AMR. Eight registers (A4–A7, B4–B7) can perform circular addressing. For each of these registers, the AMR specifies the addressing mode. A 2-bit field for each register is used to select the address modification mode: linear (the default) or circular mode. With circular addressing, the field also specifies which BK (block size) field to use for a circular buffer. In addition, the buffer must be aligned on a byte boundary equal to the block size. The mode select field encoding is shown in Table 2–4.

Figure 2–3. Addressing Mode Register (AMR)



Legend: R Readable by the **MVC** instruction
W Writeable by the **MVC** instruction
+0 Value is zero after reset

Table 2–4. Addressing Mode Field Encoding

Mode	Description
0 0	Linear modification (default at reset)
0 1	Circular addressing using the BK0 field
1 0	Circular addressing using the BK1 field
1 1	Reserved

The reserved portion of AMR is always 0. The AMR is initialized to zero at reset.

The block size fields, BK0 and BK1, contain 5-bit values used in calculating block sizes for circular addressing.

Block size (in bytes) = $2^{(N+1)}$
where N is the 5-bit value in BK0 or BK1

Table 2–5 shows block size calculations for all 32 possibilities.

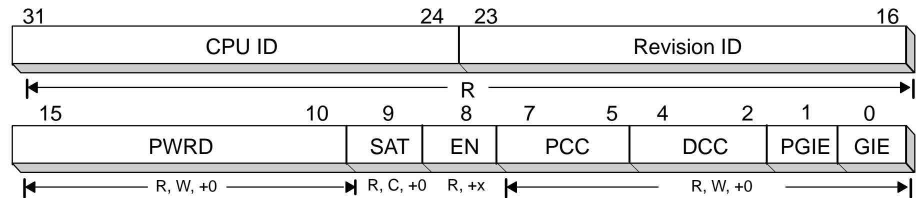
Table 2–5. Block Size Calculations

N	Block Size	N	Block Size
00000	2	10000	131 072
00001	4	10001	262 144
00010	8	10010	524 288
00011	16	10011	1 048 576
00100	32	10100	2 097 152
00101	64	10101	4 194 304
00110	128	10110	8 388 608
00111	256	10111	16 777 216
01000	512	11000	33 554 432
01001	1 024	11001	67 108 864
01010	2 048	11010	134 217 728
01011	4 096	11011	268 435 456
01100	8 192	11100	536 870 912
01101	16 384	11101	1 073 741 824
01110	32 768	11110	2 147 483 648
01111	65 536	11111	4 294 967 296

2.4 Control Status Register (CSR)

The CSR, shown in Figure 2–4, contains control and status bits. The functions of the fields in the CSR are shown in Table 2–6. For the EN, PWRD, PCC, and DCC fields, see your data sheet to see if your device supports the options that these fields control, and see the *TMS320C62xx Peripherals Reference Guide* for more information on these options.

Figure 2–4. Control Status Register (CSR)



Legend: R Readable by the **MVC** instruction
W Writeable by the **MVC** instruction
+x Value undefined after reset
+0 Value is zero after reset
C Clearable using the **MVC** instruction

Table 2–6. Control Status Register: Fields, Read/Write Status, and Function

Bit Position	Width	Field Name	Function
31-24	8	CPU ID	CPU ID; defines which CPU
23-16	8	Rev ID	Revision ID; defines silicon revision of the CPU
15-10	6	PWRD	Control power down modes; the values are always read as zero.†
9	1	SAT	The saturate bit, set when any unit performs a saturate, can be cleared only by the **MVC** instruction and can be set only by a functional unit. The set by a functional unit has priority over a clear (by the **MVC** instruction) if they occur on the same cycle. The saturate bit is set one full cycle (one delay slot) after a saturate occurs.
8	1	EN	Endian bit: 1 = little endian, 0 = big endian †
7-5	3	PCC	Program cache control mode†
4-2	3	DCC	Data cache control mode†
1	1	PGIE	Previous GIE (global interrupt enable); saves GIE when an interrupt is taken
0	1	GIE	Global interrupt enable; enables (1) or disables (0) all interrupts except the reset interrupt and NMI (nonmaskable interrupt)

† See the *TMS320C62xx Peripherals Reference Guide* for more information.

Chapter 3

Instruction Set

This chapter describes the assembly language instructions and addressing modes for the 'C62xx digital signal processor. Also described are parallel operations, conditional operations, and resource constraints.

Topic **Page**

3.1 Instruction Operation and Execution Notations 3-2
3.2 Mapping Between Instructions and Functional Units 3-4
3.3 TMS320C62xx Opcode Map 3-7
3.4 Delay Slots 3-9
3.5 Parallel Operations 3-10
3.6 Conditional Operations 3-13
3.7 Resource Constraints 3-14
3.8 Addressing Modes 3-18
3.9 Individual Instruction Descriptions 3-21

3.1 Instruction Operation and Execution Notations

Table 3–1 explains the symbols used in this chapter.

Table 3–1. Instruction Operation and Execution Notations

Symbol	Meaning
int	32-bit register value
long	40-bit register value
creg	3-bit field specifying a conditional register
*cst*n	n-bit constant field
LSBn	n least significant bits
MSBn	n most significant bits
→	Assignment
+	Addition
–	Subtraction
×	Multiplication
+a	Perform 2s-complement addition using the addressing mode defined by the AMR
–a	Perform 2s-complement subtraction using the addressing mode defined by the AMR
and	Bitwise AND
or	Bitwise OR
xor	Bitwise exclusive OR
not	Bitwise logical complement
$b_{y..z}$	Selection of bits y through z of bit string b
<<	Shift left
>>s	Shift right with sign extension
>>z	Shift right with a zero fill
x clear *b,e*	Clear a field in x, specified by b (beginning bit) and e (ending bit)

Table 3–1. Instruction Operation and Execution Notations (Continued)

Symbol	**Meaning**
x exts *l,r*	Extract and sign-extend a field in x, specified by l (shift left value) and r (shift right value)
x extu *l,r*	Extract an unsigned field in x, specified by l (shift left value) and r (shift right value)
+s	Perform 2s-complement addition and saturate the result to the result size if an overflow occurs
–s	Perform 2s-complement subtraction and saturate the result to the result size if an overflow occurs
x set *b,e*	Set field in x to all 1s, specified by b (beginning bit) and e (ending bit)
abs(x)	Absolute value of x
lmb0(x)	Leftmost 0 bit search of x
lmb1(x)	Leftmost 1 bit search of x
norm(x)	Leftmost nonredundant sign bit of x
R	Any general-purpose register
cond	Check for either *creg* equal to zero or *creg* not equal to zero
nop	No operation

3.2 Mapping Between Instructions and Functional Units

Table 3–2 and Table 3–3 define the mapping between instructions and functional units.

Table 3–2. Instruction to Functional Unit Mapping

.L Unit	.M Unit	.S Unit	.D Unit
ABS	MPY	ADD	ADD
ADD	SMPY	ADDK	ADDA
AND		ADD2	LD mem
CMPEQ		AND	LD mem (15-bit offset)‡
CMPGT		B disp	MV
CMPGTU		B IRP†	NEG
CMPLT		B NRP†	ST mem
CMPLTU		B reg	ST mem (15-bit offset)‡
LMBD		CLR	SUB
MV		EXT	SUBA
NEG		EXTU	ZERO
NORM		MVC†	
NOT		MV	
OR		MVK	
SADD		MVKH	
SAT		NEG	
SSUB		NOT	
SUB		OR	
SUBC		SET	
XOR		SHL	
ZERO		SHR	
		SHRU	
		SSHL	
		SUB	
		SUB2	
		XOR	
		ZERO	

† .S2 only
‡ .D2 only

Table 3–3. Functional Unit to Instruction Mapping

Instruction	'C62xx Functional Units			
	.L Unit	.M Unit	.S Unit	.D Unit
ABS	✓			
ADD	✓		✓	✓
ADDA				✓
ADDK			✓	
ADD2			✓	
AND	✓		✓	
B			✓	
B IRP			✓†	
B NRP			✓†	
B reg			✓†	
CLR			✓	
CMPEQ	✓			
CMPGT	✓			
CMPGTU	✓			
CMPLT	✓			
CMPLTU	✓			
EXT			✓	
EXTU			✓	
IDLE				
LD mem				✓
LD mem (15-bit offset)				✓‡
LMBD	✓			
MPY		✓		
MVC†			✓	
MV	✓		✓	✓
MVK			✓	

† .S2 only
‡ .D2 only

Table 3–3. Functional Unit to Instruction Mapping (Continued)

	’C62xx Functional Units			
Instruction	**.L Unit**	**.M Unit**	**.S Unit**	**.D Unit**
MVKH			✓	
NEG	✓		✓	✓
NOP				
NORM	✓			
NOT	✓		✓	
OR	✓		✓	
SADD	✓			
SAT	✓			
SET			✓	
SHL			✓	
SHR			✓	
SHRU			✓	
SMPY		✓		
SSHL			✓	
SSUB	✓			
ST mem				✓
ST mem (15-bit offset)				✓‡
SUB	✓		✓	✓
SUBA				✓
SUBC	✓			
SUB2			✓	
SWI				
XOR	✓		✓	
ZERO	✓		✓	✓

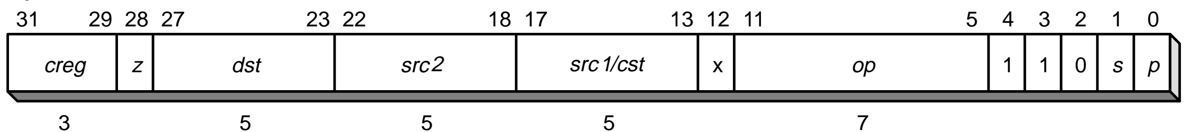
† .S2 only
‡ .D2 only

3.3 TMS320C62xx Opcode Map

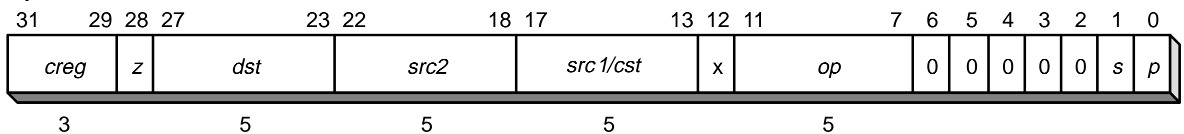
The ’C62xx opcode map is shown in Figure 3–1. Refer to Table 3–1 and the instruction descriptions in this chapter for explanations of the field syntaxes and values.

Figure 3–1. TMS320C62xx Opcode Map

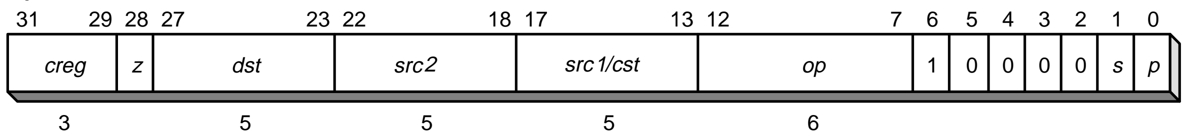
Operations on the .L unit



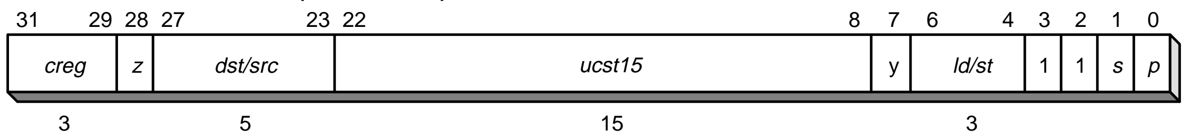
Operations on the .M unit



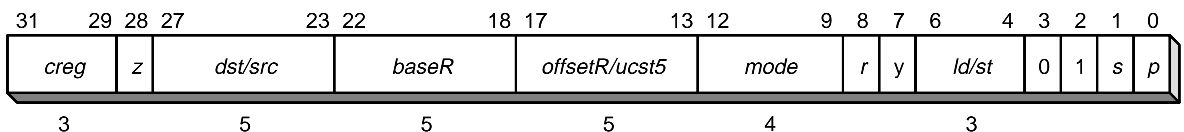
Operations on the .D unit



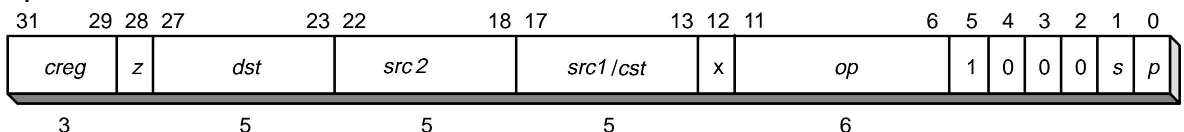
Load/store with 15-bit offset (on the .D unit)



Load/store baseR + offsetR/cst on the .D unit



Operations on the .S unit



ADDK on the .S unit

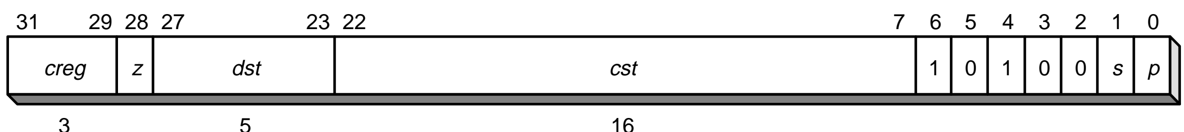
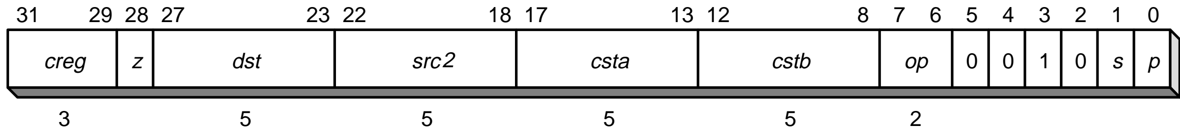
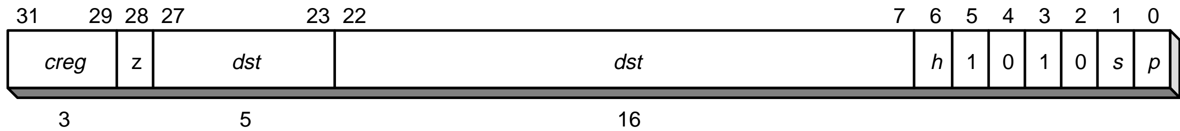


Figure 3–1. TMS320C62xx Opcode Map (Continued)

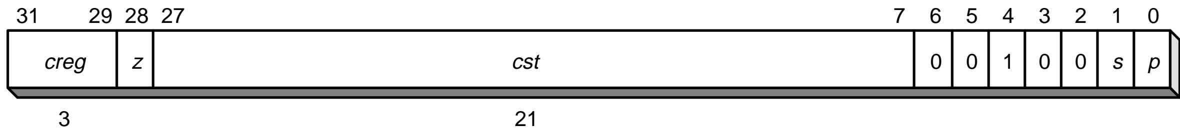
Field operations (immediate forms) on the .S unit



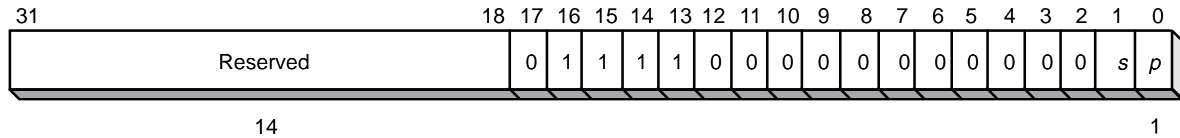
MVK and MVKH on the .S unit



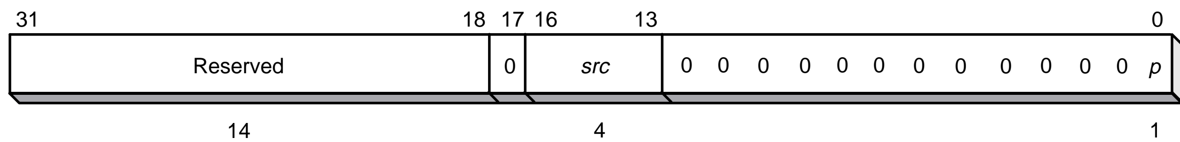
Bcond disp on the .S unit



IDLE



NOP



3.4 Delay Slots

The execution of instructions can be defined in terms of delay slots. The numbered delay slots is equivalent to the number of extra cycles required before a result is available for reading after the source operands are read. For a single-cycle type instruction (such as **ADD**), source operands read in cycle i produce a result that can be read in cycle $i + 1$. For a multiply instruction (**MPY**), source operands read in cycle i produce a result that can be read in cycle $i + 2$. Table 3–4 shows the number of delay slots associated with each type of instruction.

Table 3–4. Delay Slot Summary

Instruction Type	Delay Slots
NOP (no execution pipeline operation)	0
Store	0
Single cycle	0
Multiply	1
Load (LD) (address modification occurs in E1)	4
Branch (The cycle when the target enters E1)	5

3.5 Parallel Operations

Instructions are always fetched eight at a time. This constitutes a *fetch packet*. The basic format of a fetch packet is shown in Figure 3–2. The execution grouping of the fetch packet is specified by the *p*-bit (bit 0) of each instruction. Fetch packets are aligned on 256-bit (8-word) boundaries.

Figure 3–2. Basic Format of a Fetch Packet

	Instruction A	Instruction B	Instruction C	Instruction D	Instruction E	Instruction F	Instruction G	Instruction H
Bits	31 … 0 (*p*)	31 … 0 (*p*)	31 … 0 (*p*)	31 … 0 (*p*)	31 … 0 (*p*)	31 … 0 (*p*)	31 … 0 (*p*)	31 … 0 (*p*)
LSBs of the byte address	00000_2	00100_2	01000_2	01100_2	10000_2	10100_2	11000_2	11100_2

The *p*-bit controls the parallel execution of instructions. The *p*-bits are scanned from left to right (lower to higher address). If the *p*-bit of instruction *i* is 1, then instruction *i* + 1 is to be executed in parallel with (in the the same cycle as) instruction *i*. If the *p*-bit of instruction *i* is 0, then instruction *i* + 1 is executed in the cycle after instruction *i*. All instructions executing in parallel constitute an *execute packet*. An execute packet can contain up to eight instructions. Each instruction in an execute packet must use a different functional unit.

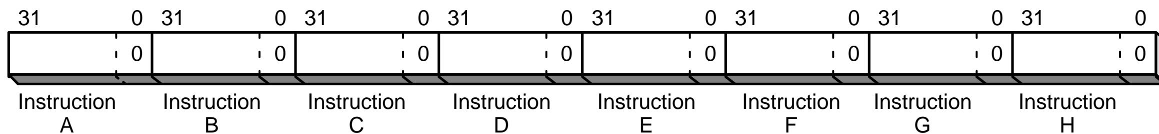
An execute packet cannot cross an 8-word boundary. Therefore, the last *p*-bit in a fetch packet is always set to 0, and each fetch packet starts a new execute packet. There are three types of *p*-bit patterns for fetch packets. These three *p*-bit patterns result in the following execution sequences for the eight instructions:

- Fully serial
- Fully parallel
- Partially serial

Example 3–1 through Example 3–3 illustrate the conversion of a *p*-bit sequence into a cycle-by-cycle execution stream of instructions.

Example 3–1. Fully Serial p-Bit Pattern in a Fetch Packet

This *p*-bit pattern:



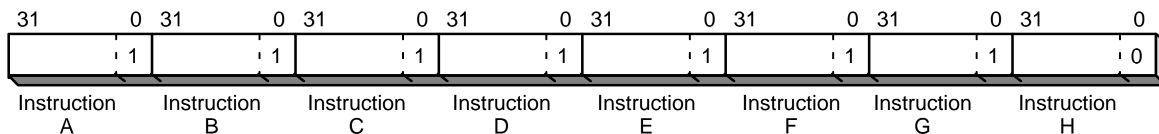
results in this execution sequence:

Cycle/Execute Packet	Instructions
1	A
2	B
3	C
4	D
5	E
6	F
7	G
8	H

The eight instructions are executed sequentially.

Example 3–2. Fully Parallel p-Bit Pattern in a Fetch Packet

This *p*-bit pattern:



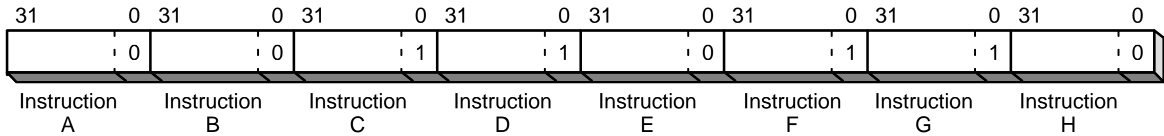
results in this execution sequence:

Cycle/Execute Packet	Instructions							
1	A	B	C	D	E	F	G	H

All eight instructions are executed in parallel.

Example 3–3. Partially Serial p-Bit Pattern in a Fetch Packet

This *p*-bit pattern:



results in this execution sequence:

Cycle/Execute Packet	Instructions		
1	A		
2	B		
3	C	D	E
4	F	G	H

Note: Instructions C, D, and E do not use any of the same functional units, cross paths, or other data path resources. This is also true for instructions F, G, and H.

3.5.1 Example Parallel Code

The || characters signify that an instruction is to execute in parallel with the previous instruction. The code for the fetch packet in Example 3–3 would be represented as this:

```
   instruction A

   instruction B

   instruction C
|| instruction D
|| instruction E

   instruction F
|| instruction G
|| instruction H
```

3.5.2 Branching Into the Middle of an Execute Packet

If a branch into the middle of an execution packet occurs, all instructions at lower addresses are ignored. In Example 3–3, if a branch to the address containing instruction D occurs, then only D and E execute. Even though instruction C is in the same execute packet, it is ignored. Instructions A and B are also ignored because they are in earlier execute packets.

3.6 Conditional Operations

All instructions can be conditional. The condition is controlled by a 3-bit opcode field (*creg*) that specifies the condition register tested, and a 1-bit field (*z*) that specifies a test for zero or nonzero. The four MSBs of every opcode are *creg* and *z*. The register is tested at the beginning of the E1 pipeline stage for all instructions. For more information on the pipeline, see Chapter 4, *Pipeline Operation*. If *z* = 1, the test is for equality with zero. If *z* = 0, the test is for nonzero. The case of *creg* = 0 and *z* = 0 is treated as always true to allow instructions to be executed unconditionally. The *creg* field is encoded in the instruction opcode as shown in Table 3–5.

Table 3–5. Registers That Can Be Tested by Conditional Operations

Conditional Register		***creg***			***z***
	Bit	***31***	***30***	***29***	***28***
Unconditional		0	0	0	0
Reserved		0	0	0	1
B0		0	0	1	*z*
B1		0	1	0	*z*
B2		0	1	1	*z*
A1		1	0	0	*z*
A2		1	0	1	*z*
Reserved		1	1	x	x

Note: x can be any value in reserved cases.

Conditional instructions are represented by using square brackets, [ ], surrounding the condition register. The following execute packet contains two **ADD** instructions in parallel. The first **ADD** is conditional on B0 being nonzero. The second **ADD** is conditional on B0 being zero. The character ! indicates the 'not' of the condition.

```
   [B0]  ADD   .L1   A1,A2,A3
|| [!B0] ADD   .L2   B1,B2,B3
```

The above instructions are mutually exclusive. This means that only one will execute.

If they are scheduled in parallel, mutually exclusive instructions are constrained as described in section 3.7. If mutually exclusive instructions share any resources as described in section 3.7, they cannot be scheduled in parallel (put in the same execute packet), even though only one will execute.

3.7 Resource Constraints

No two instructions within the same execute packet can use the same resources. Also, no two instructions can write to the same register during the same cycle. The following sections, 3.7.1 to 3.7.5, describe each of the resources an instruction can use.

3.7.1 Constraints on Instructions Using the Same Functional Unit

Two instructions using the same functional unit cannot be issued in the same execute packet.

The following execute packet is invalid:

```
      ADD .S1   A0, A1, A2 ; \ .S1 is used for
||    SHR .S1   A3, 15, A4 ; / both instructions
```

The following execute packet is valid:

```
      ADD .L1   A0, A1, A2 ; \ Two different functional
||    SHR .S1   A3, 15, A4 ; / units are used
```

3.7.2 Constraints on Cross Paths (1X and 2X)

One unit (either a .S, .L, or .M unit) per data path, per execute packet, can read a source operand from its opposite register file via the cross paths (1X and 2X). For example, .S1 can read both of an instruction's operands from the A register file, or it can read one operand from the B register file using the 1X cross path and the other from the A register file. This is denoted by an X following the unit name in the instruction syntax.

Two instructions using the same cross path between register files cannot be issued in the same execute packet, because there is only one path from A to B and one path from B to A.

The following execute packet is invalid:

```
      ADD.L1X   A0,B1,A1 ; \ 1X cross path is used
||    MPY.M1X   A4,B4,A5 ; / for both instructions
```

The following execute packet is valid:

```
      ADD.L1X   A0,B1,A1 ; \ Instructions use the 1X and
||    MPY.M2X   B4,A4,B2 ; / 2X cross paths
```

The operand will come from a register file opposite of the destination if the x bit in the instruction field is set (shown in the opcode map located in section 3.3).

3.7.3 Constraints on Loads and Stores

Loads and stores can use an address pointer from one register file while loading to or storing from the other register file. Two loads and/or stores using an address pointer from the same register file cannot be issued in the same execute packet.

The following execute packet is invalid:

```
   LDW.D1    *A0,A1 ; \ Address registers from the same
|| LDW.D1    *A2,B2 ; / register file
```

The following execute packet is valid:

```
   LDW.D1    *A0,A1 ; \ Address registers from different
|| LDW.D2    *B0,B2 ; / register files
```

Two loads and/or stores loading to and/or storing from the same register file cannot be issued in the same execute packet.

The following execute packet is invalid:

```
   LDW.D1    *A4,A5 ; \ Loading to and storing from the
|| STW.D2    A6,*B4 ; / same register file
```

The following execute packet is valid:

```
   LDW.D1    *A4,B5 ; \ Loading to, and storing from
|| STW.D2    A6,*B4 ; / different register files
```

3.7.4 Constraints on Long (40-Bit) Data

Because the .S and .L units share a read register port for long source operands and a write register port for long results, only one long result may be issued per register file in an execute packet. See section 2.1.1, on page 2-2 for the order for long pairs.

The following execute packet is invalid:

```
   ADD.L1    A5:A4,A1,A3:A2  ; \ Two long writes
|| SHL.S1    A8,A9,A7:A6     ; / on A register file
```

The following execute packet is valid:

```
   ADD.L1    A5:A4,A1,A3:A2   ; \ One long write for
|| SHL.S2    B8,B9,B7:B6      ; / each register file
```

Because the .L and .S units share their long read port with the store port, operations that read a long value cannot be issued on the .L and/or .S units in the same execute packet as a store.

The following execute packet is invalid:

```
   ADD.L1    A5:A4,A1,A3:A2   ; \ Long read operation and a
|| STW.D1    A8,*A9           ; / store
```

The following execute packet is valid:

```
   ADD.L1    A4, A1, A3:A2    ; \ No long read with
|| STW.D1    A8,*A9           ; / with the store
```

3.7.5 Constraints on Register Reads

More than four reads of the same register cannot occur on the same cycle. Conditional registers are not included in this count.

The following code sequence is invalid:

```
         MPY    .M1 A1,A1,A4 ; five reads of register A1
||       ADD    .L1 A1,A1,A5
||       SUB    .D1 A1,A2,A3
```

This code sequence is valid:

```
         MPY    .M1 A1,A1,A4 ; only four reads of A1
|| [A1]  ADD    .L1 A0,A1,A5
||       SUB    .D1 A1,A2,A3
```

3.7.6 Constraints on Register Writes

Multiple writes to the same register on the same cycle can occur if instructions with different latencies writing to the same register are issued on different cycles. For example, an **MPY** issued on cycle *i* followed by an **ADD** on cycle *i* + 1 cannot write to the same register because both instructions write a result on cycle *i* + 1. Therefore, the following code sequence is invalid:

```
MPY    .M1 A0,A1,A2
ADD    .L1 A4,A5,A2
```

Figure 3–3 shows different multiple-write conflicts. For example, **ADD** and **SUB** in execute packet L1 write to the same register. This conflict is easily detectable.

MPY in packet L2 and **ADD** in packet L3 might both write to B2 simultaneously; however, if a branch instruction causes the execute packet after L2 to be something other than L3, a conflict would not occur. Thus, the potential conflict in L2 and L3 might not be detected by the assembler. The instructions in L4 do not constitute a write conflict because they are mutually exclusive. In contrast, because it is not obvious that the instructions in L5 are mutually exclusive, the assembler cannot determine a conflict. If the pipeline does receive commands to perform multiple writes to the same register, the result is undefined.

Figure 3–3. Examples of the Detectability of Write Conflicts by the Assembler

```
L1:       ADD.L2   B5,B6,B7 ; \ detectable, conflict
||        SUB.S2   B8,B9,B7 ; /

L2:       MPY.M2   B0,B1,B2 ; \ not detectable

L3:       ADD.L2   B3,B4,B2 ; /

L4:[!B0]  ADD.L2   B5,B6,B7 ; \ detectable, no conflict
|| [B0]   SUB.S2   B8,B9,B7 ; /

L5:[!B1]  ADD.L2   B5,B6,B7 ; \ not detectable
|| [B0]   SUB.S2   B8,B9,B7 ; /
```

3.8 Addressing Modes

The addressing modes on the ’C62xx are linear, circular using BK0, and circular using BK1. The mode is specified by the addressing mode register, or AMR (defined in Chapter 2).

Eight registers can perform circular addressing. A4-A7 are used by the .D1 unit and B4-B7 are used by the .D2 unit. No other units can perform circular addressing. All registers can perform linear addressing. **LD(B)(H)(W)**, **ST(B)(H)(W)**, **ADDA(B)(H)(W)**, and **SUBA(B)(H)(W)** instructions all use the AMR to determine what type of address calculations are performed for these registers.

3.8.1 Linear Addressing Mode

LD/ST instructions

Linear mode simply shifts the *offsetR/cst* operand to the left by 2, 1, or 0 for word, half-word, or byte access, respectively, and then performs an add or a subtract to *baseR* (depending on the operation specified).

ADDA/SUBA instructions

Linear mode simply shifts the *src1/cst* operand to the left by 2, 1, or 0 for word, halfword, or byte data sizes, respectively, and then performs the add or subtract specified.

3.8.2 Circular Addressing Mode

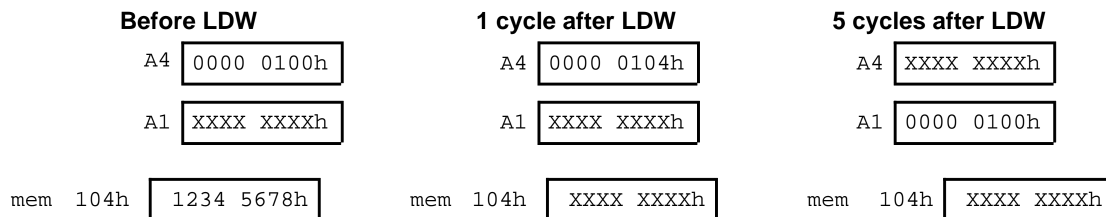
The BK0 and BK1 fields in the AMR specify block sizes for circular addressing. See section 2.3 for more information on the AMR.

LD/ST Instructions

After shifting *offsetR/cst* to the left by 2, 1, or 0 for **LDW**, **LDH**, or **LDB**, respectively, an add or subtract is performed with the carry/borrow inhibited between bits N and N + 1. Bits N + 1 to 31 of *baseR* remain unchanged. All other carries/borrows propagate as usual. If you specify an *offsetR/cst* greater than the circular buffer size, $2^{(N + 1)}$, the effective *offsetR/cst* is modulo the circular buffer size (see Example 3–4). The circular buffer size in the AMR is not scaled; for example: a block size of 4 is 4 bytes, not 4 × data size (byte, halfword, word). So, to perform circular addressing on an array of 8 words, a size of 32 should be specified, or N = 4. Example 3–4 shows a **LDW** performed with register A4 in circular mode and BK0 = 4, so the buffer size is 32 bytes, 16 halfwords, or 8 words. The value put in the AMR for this example is 0004 0001h.

Example 3–4. LDW in Circular Mode

```
LDW    .D1    *++A4[9],A1
```



Note: 9h words is 24h bytes. 24h bytes is 4 bytes beyond the 32-byte (20h) boundary 100h–11Fh; thus, it is wrapped around to 104h.

ADDA/SUBA Instructions

After shifting *src*1/*cst* to the left by 2, 1, or 0 for **ADDAW**, **ADDAH**, or **ADDAB**, respectively, an add or a subtract is performed with the carry/borrow inhibited between bits N and N + 1. Bits N + 1 to 31 (inclusive) of *src*2 remain unchanged. All other carries/borrows propagate as usual. If you specify *src*1 greater than the circular buffer size, $2^{(N+1)}$, the effective *offsetR/cst* is modulo the circular buffer size (see Example 3–5). The circular buffer size in the AMR is not scaled; for example, a block size of 4 is 4 bytes, not 4 × data size (byte, half-word, word). So, to perform circular addressing on an array of 8 words, a size of 32 should be specified, or N = 4. Example 3–5 shows an **ADDAH** performed with register A4 in circular mode and BK0 = 4, so the buffer size is 32 bytes, 16 halfwords, or 8 words. The value put in the AMR for this example is 0004 0001h.

Example 3–5. ADDAH in Circular Mode

```
ADDAH     .D1    A4,A1,A1
```

Note: 13h halfwords is 26h bytes. 26h bytes is 6 bytes beyond the 32-byte (20h) boundary 100h–11Fh; thus, it is wrapped around to 106h.

3.8.3 Syntax for Load/Store Address Generation

The ’C62xx CPU has a load/store architecture, which means that the only way to access data in memory is with a load or store instruction. Table 3–6 shows the syntax of an indirect address to a memory location.

Table 3–6. Indirect Address Generation for Load/Store

Addressing Type	No Modification of Address Register	Preincrement or Decrement of Address Register	Postincrement Decrement of Address Register
Register indirect	*R	*++R *– –R	*R++ *R– –
Register relative	*+R[*ucst5*] *–R[*ucst5*]	*++R[*ucst5*] *– –R[*ucst5*]	*R++[*ucst5*] *R– –[*ucst5*]
Base + index	*+R[*offsetR*] *–R[*offsetR*]	*++R[*offsetR*] *– –R[*offsetR*]	*R++[*offsetR*] *R– –[*offsetR*]

3.9 Individual Instruction Descriptions

This section gives detailed information on the instruction set for the ’C62xx. Each instruction presents the following information:

- Assembler syntax
- Functional units
- Operands
- Opcode
- Description
- Execution
- Instruction type
- Delay slots
- Examples

An example instruction description is provided before the actual instruction descriptions (starting with the ABS instruction description). The example is provided to familiarize you with how the instructions are described. It includes and describes information pulled from the ADD instruction description, giving basic information about what each description category provides and where to go for more information.

Syntax

EXAMPLE (.unit) *src*, *dst*
.unit = .L1, .L2, .S1, .S2, .D1, .D2

src and *dst* indicate source and destination, respectively. The (.unit) dictates which functional unit the instruction is mapped to (.L1, .L2, .S1, .S2, .M1, .M2, .D1, or .D2).

A table is provided for each instruction that gives the opcode map fields, units the instruction is mapped to, types of operands, and the opcode.

The opcode map breaks down the various bit fields that make up each instruction. These fields are illustrated in section 3.3.

There are instructions that can be executed on more than one functional unit. Table 3–7 shows how this situation is documented for the **ADD** instruction. This instruction has three opcode map fields: *src1*, *src2*, and *dst*. In the seventh row, the operands have the types *cst5*, *long*, and *long* for *src1*, *src2*, and *dst*, respectively. The ordering of these fields implies *cst5* + *long* → *long*, where + represents the operation being performed by the **ADD**. This operation can be done on .L1 or .L2 (both are specified in the unit column). The s in front of each operand signifies that *src*1 (s*cst5*), *src*2 (slong), and *dst* (slong) are all signed values.

In the third row, *src*1, *src*2, and *dst* are int, int, and long, respectively. The u in front of each operand signifies that all operands are unsigned. Any operand that begins with x can be read from a register file that is different from the destination register file. The operand comes from the register file opposite the destination if the x bit in the instruction field is set (shown in the opcode map located in section 3.3).

Table 3–7. Relationships Between Operands, Operand Size, Signed/Unsigned, Functional Units, and Opfields for Example Instruction (ADD Instruction)

Opcode map field used...	For operand type...	Unit	Opfield	Mnemonic
src1 *src2* *dst*	sint xsint sint	.L1, .L2	0000011	ADD
src1 *src2* *dst*	sint xsint slong	.L1, .L2	0100011	ADD
src1 *src2* *dst*	uint xunit ulong	.L1, .L2	0101011	ADDU
src1 *src2* *dst*	xsint slong slong	.L1, .L2	0100001	ADD
src1 *src2* *dst*	xuint ulong ulong	.L1, .L2	0101001	ADDU
src1 *src2* *dst*	*scst*5 xsint sint	.L1, .L2	0000010	ADD
src1 *src2* *dst*	*scst*5 slong slong	.L1, .L2	0100000	ADD
src1 *src2* *dst*	sint xsint sint	.S1, .S2	000111	ADD
src1 *src2* *dst*	*scst*5 xsint sint	.S1, .S2	000110	ADD
src2 *src1* *dst*	sint sint sint	.D1, .D2	010000	ADD
src2 *src1* *dst*	sint *ucst*5 sint	.D1, .D2	010010	ADD

Description Instruction execution and its effect on the rest of the processor or memory contents are described. Any constraints on the operands imposed by the processor or the assembler are discussed. The description parallels and supplements the information given by the execution block.

Execution for .L1, .L2 and .S1, .S2 opcodes

if (cond) $src1 + src2 \rightarrow dst$
else nop

Execution for .D1, .D2 opcodes

if (cond) $src2 + src1 \rightarrow dst$
else nop

The execution describes the processing that takes place when the instruction is executed. The symbols are defined in Table 3–1 on page 3-2.

Instruction Type This section gives the type of instruction. See section 4.2.

Delay Slots This section gives the number of delay slots the instruction takes to execute (see section 3.4).

Example Examples of instruction execution. If applicable, register and memory values are given before and after instruction execution.

Syntax

ABS (.unit) *src2*, *dst*

.unit = .L1, .L2

Opcode map field used...	For operand type...	Unit	Opfield
src2 *dst*	xsint sint	.L1, .L2	0011010
src2 *dst*	slong slong	.L1, L2	0111000

Description

The absolute value of *src2* is placed in *dst*.

Execution

if (cond) abs(*src2*) → *dst*
else nop

The absolute value of *src2* when *src2* is an int is determined as follows:

1) If $src2 \geq 0$, then $src2 \rightarrow dst$
2) If $src2 < 0$ and $src2 \neq -2^{31}$, then $-src2 \rightarrow dst$
3) If $src2 = -2^{31}$, then $2^{31}-1 \rightarrow dst$

The absolute value of *src2* when *src2* is a long is determined as follows:

1) If $src2 \geq 0$, then $src2 \rightarrow dst$
2) If $src2 < 0$ and $src2 \neq -2^{39}$, then $-src2 \rightarrow dst$
3) If $src2 = -2^{39}$, then $2^{39} - 1 \rightarrow dst$

Instruction Type

Single-cycle

Delay Slots

0

Example 1

```
ABS .L1    A1,A5
```

	Before instruction			1 cycle after instruction	
A1	8000 4E3Dh	−2147463619	A1	8000 4E3Dh	−2147463619
A5	XXXX XXXXh		A5	7FFF B1C3h	2147463619

Example 2

```
ABS .L1    A1,A5
```

	Before instruction			1 cycle after instruction	
A1	3FF6 0010h	1073086480	A1	3FF6 0010h	1073086480
A5	XXXX XXXXh		A5	3FF6 0010h	1073086480

Syntax

ADD (.unit) *src1*, *src2*, *dst*
or
ADDU (.unit) *src1*, *src2*, *dst*
or
ADD (.unit) *src2*, *src1*, *dst*

.unit = .L1, .L2, .S1, .S2, .D1, .D2

Opcode map field used...	For operand type...	Unit	Opfield
src1 *src2* *dst*	sint xsint sint	.L1, .L2	0000011
src1 *src2* *dst*	sint xsint slong	.L1, .L2	0100011
src1 *src2* *dst*	uint xunit ulong	.L1, .L2	0101011
src1 *src2* *dst*	xsint slong slong	.L1, .L2	0100001
src1 *src2* *dst*	xuint ulong ulong	.L1, .L2	0101001
src1 *src2* *dst*	*scst*5 xsint sint	.L1, .L2	0000010
src1 *src2* *dst*	*scst*5 slong slong	.L1, .L2	0100000
src1 *src2* *dst*	sint xsint sint	.S1, .S2	000111
src1 *src2* *dst*	*scst*5 xsint sint	.S1, .S2	000110
src2 *src1* *dst*	sint sint sint	.D1, .D2	010000
src2 *src1* *dst*	sint *ucst*5 sint	.D1, .D2	010010

Description for .L1, .L2 and .S1, .S2 opcodes

src2 is added to *src1*. The result is placed in *dst*.

Execution for .L1, .L2 and .S1, .S2 opcodes

if (cond) *src1* + *src2* → *dst*
else nop

Description for .D1, .D2 opcodes

src1 is added to *src2*. The result is placed in *dst*.

Execution for .D1, .D2 opcodes

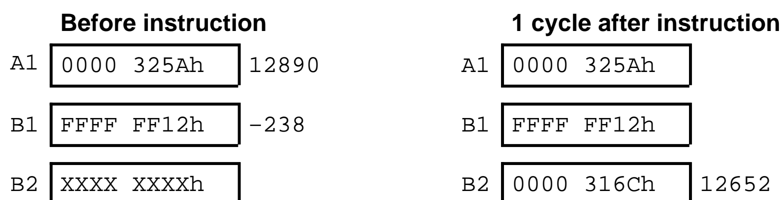
if (cond) *src2* + *src1* → *dst*
else nop

Instruction Type Single-cycle

Delay Slots 0

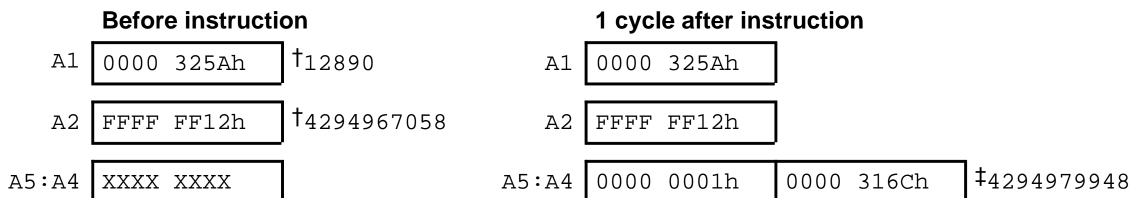
Example 1

```
ADD .L2    A1,B1,B2
```



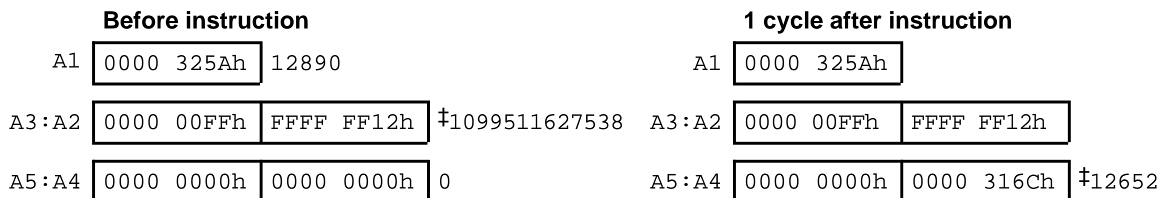
Example 2

```
ADDU .L1    A1,A2,A5:A4
```



Example 3

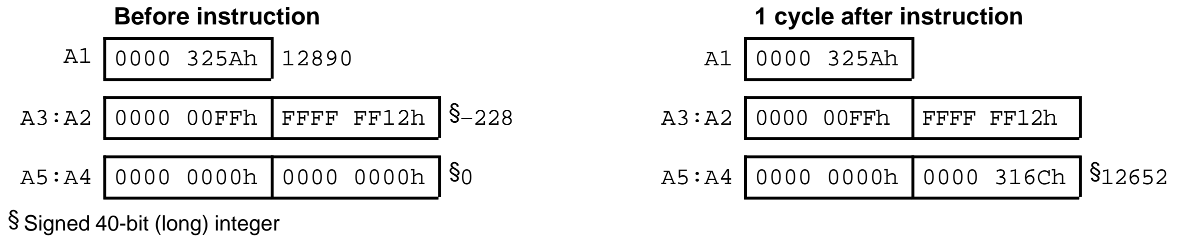
```
ADDU .L1    A1,A3:A2,A5:A4
```



† Unsigned 32-bit integer
‡ Unsigned 40-bit (long) integer

Example 4

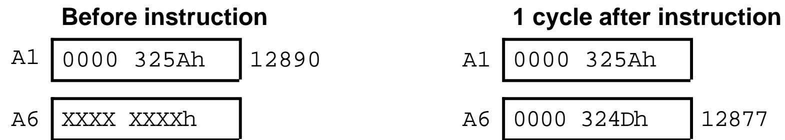
```
ADD .L1    A1,A3:A2,A5:A4
```



§ Signed 40-bit (long) integer

Example 5

```
ADD .L1    −13,A1,A6
```



Example 6

```
ADD .D1    26,A1,A6
```

Syntax

ADDAB (.unit) *src2*, *src1*, *dst*
or
ADDAH (.unit) *src2*, *src1*, *dst*
or
ADDAW (.unit) *src2*, *src1*, *dst*

.unit = .D1 or .D2

Opcode map field used...	For operand type...	Unit	Opfield
src2 *src1* *dst*	sint sint sint	.D1, .D2	byte: 110000 halfword: 110100 word: 111000
src2 *src1* *dst*	sint *ucst*5 sint	.D1, .D2	byte: 110010 halfword: 110110 word: 111010

Description

src1 is added to *src2* using the addressing mode specified for *src2*. The addition defaults to linear mode. However, if *src2* is one of A4–A7 or B4–B7, the mode can be changed to circular mode by writing the appropriate value to the AMR (see section 2.3). *src1* is left shifted by 1 or 2 for halfword and word data sizes respectively. Byte, halfword, and word mnemonics are **ADDAB**, **ADDAH**, and **ADDAW**, respectively. The result is placed in *dst*.

Execution

if (cond) *src2* +a *src1* → *dst*
else nop

Instruction Type Single-cycle

Delay Slots 0

Example 1

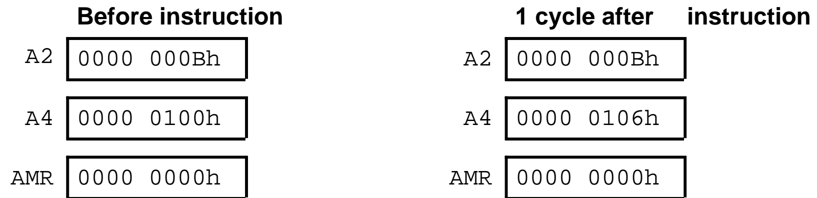
```
ADDAB .D1  A4,A2,A4
```

	Before instruction		1 cycle after instruction
A2	0000 000Bh	A2	0000 000Bh
A4	0000 0100h	A4	0000 0103h
AMR	0002 0001h	AMR	0002 0001h

```
BK0 = 2 → size = 8
A4 in circular addressing mode using BK0
```

Example 2

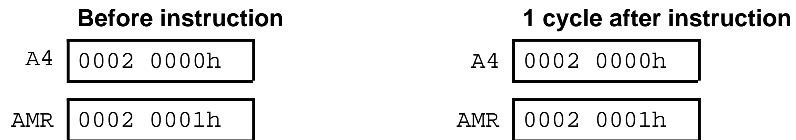
```
ADDAH .D1  A4,A2,A4
```



BK0 = 2 → size = 8
A4 in linear addressing mode

Example 3

```
ADDAW .D1  A4,2,A4
```

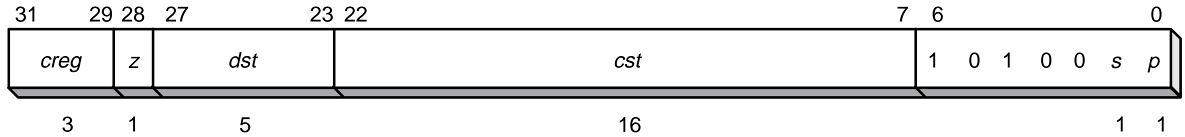


BK0 = 2 → size = 8
A4 in circular addressing mode using BK0

Syntax **ADDK** (.unit) *cst*, *dst*

.unit = .S1 or .S2

Opcode



Opcode map field used...	For operand type...	Unit
cst *dst*	*scst*16 uint	.S1, .S2

Description A 16-bit signed constant is added to the *dst* register specified. The result is placed in *dst*.

Execution

if (cond) *cst* + *dst* → *dst*
else nop

Instruction Type Single-cycle

Delay Slots 0

Example

```
ADDK .S1   15401,A1
```

	Before instruction			1 cycle after instruction	
A1	0021 37E1h	2176993	A4	0021 740Ah	2192394

Syntax

ADD2 (.unit) *src1*, *src2*, *dst*

.unit = .S1 or .S2

Opcode map field used...	For operand type...	Unit	Opfield
src1 *src2* *dst*	sint xsint sint	.S1, .S2	000001

Description

The upper and lower halves of the *src1* operand are added to the upper and lower halves of the *src2* operand. Any carry from the lower half add does not affect the upper half add.

Execution

if (cond) {
((lsb16(*src1*) + lsb16(*src2*)) and FFFFh) or
((msb16(*src1*) + msb16(*src2*)) << 16) → *dst*
}
else nop

Instruction Type

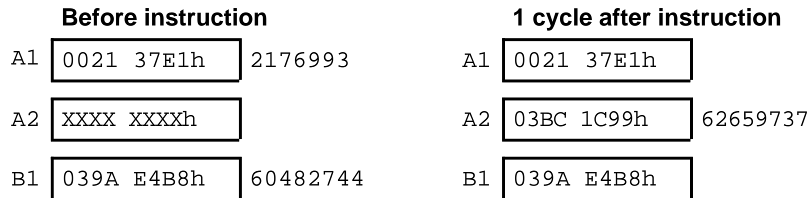
Single-cycle

Delay Slots

0

Example

```
ADD2 .S1   A1,B1,A2
```



Syntax

AND (.unit) *src1*, *src2*, *dst*

.unit = .L1 or .L2, .S1 or .S2

Opcode map field used...	For operand type...	Unit	Opfield
src1 *src2* *dst*	uint xuint uint	.L1, .L2	1111001
src1 *src2* *dst*	*scst*5 xuint uint	.L1, .L2	1111010
src1 *src2* *dst*	uint xuint uint	.S1, .S2	011111
src1 *src2* *dst*	*scst*5 xuint uint	.S1, .S2	011110

Description

A bitwise AND is performed between *src1* and *src2*. The result is placed in *dst*. The *scst*5 operands are sign extended to 32 bits.

Execution

if (cond) *src1* and *src2* → *dst*
else nop

Delay Slots

0

Example 1

```
AND .L1    A1,B1,A2
```

	Before instruction		1 cycle after instruction
A1	F7A1 302Ah	A1	F7A1 302Ah
A2	XXXX XXXXh	A2	02A0 2020h
B1	02B6 E724h	B1	02B6 E724h

Example 2

```
AND .L1    15,A1,A3
```

	Before instruction		1 cycle after instruction
A1	32E4 6936h	A1	32E4 6936h
A3	XXXX XXXXh	A3	0000 0006h

Syntax **B** (.unit) label

.unit = .S1 or .S2

Opcode

31–29	28	27–7	6	5	4	3	2	1	0
creg	*z*	*cst*	1	0	1	0	0	*s*	*p*
3	1	21						1	1

Opcode map field used...	For operand type...	Unit
cst	*scst21*	.S1, .S2

Description A 21-bit signed constant specified by *cst* is shifted left by 2 bits and is added to the address of the first instruction of the fetch packet that contains the branch instruction. The result is placed in the program fetch counter (PFC). The assembler/linker automatically computes the correct value for *cst* by the following formula:

cst = (label – PCE1) >> 2

If two branches are in the same execute packet and both are taken, behavior is undefined.

Two conditional branches can be in the same execute packet only if one of them is B using a register, B IRP, or B NRP. As long as only one branch has a true condition, code executes in a normal, well-defined way.

Execution

if (cond) *cst* << 2 + PCE1 → PFC
else nop

Notes:

1) PCE1 (program counter) represents the address of the first instruction in the fetch packet in the E1 stage of the pipeline. PFC is the program fetch counter.
2) An execute packet containing a branch and the execute packets in its delay slots cannot be interrupted. This is true regardless of whether the branch is taken.
3) See section 3.5.2 on page 3-12 for information on branching into the middle of an execute packet.

Instruction Type Branch

Delay Slots 5

Table 3–8 gives the program counter values and actions for the following code example.

Example

```
0000 0000          B      .S1    LOOP
0000 0004          ADD    .L1    A1, A2, A3
0000 0008       || ADD    .L2    B1, B2, B3
0000 000C    LOOP: MPY    .MIX   A3, B3, A4
0000 0010       || SUB    .D1    A5, A6, A6
0000 0014          MPY    .M1    A3, A6, A5
0000 0018          MPY    .M2    A6, A7, A8
0000 001C          SHR    .S1    A4, 15, A4
0000 0020          ADD    .D1    A4, A6, A4
```

Table 3–8. Program Counter Values for Branch Using a Displacement Example

Cycle	Program Counter Value	Action
Cycle 0	0000 0000h	Branch command executes (target code fetched)
Cycle 1	0000 0004h	
Cycle 2	0000 000Ch	
Cycle 3	0000 0014h	
Cycle 4	0000 0018h	
Cycle 5	0000 001Ch	
Cycle 6	0000 000Ch	Branch target code executes
Cycle 7	0000 0014h	

Syntax

B (.unit) *src2*

.unit = .S2

Opcode map field used...	For operand type...	Unit	Opfield
src2	xuint	.S2	001101

Description

src2 is placed in the PFC.

If two branches are in the same execute packet and are both taken, behavior is undefined.

Two conditional branches can be in the same execute packet only if one of them is B using a register, B IRP, or B NRP. As long as only one branch has a true condition, code executes in a normal, well-defined way.

Execution

if (cond) *src2* → PFC
else nop

Notes:

1) This instruction executes on .S2 only. PFC is program fetch counter.
2) An execute packet containing a branch and the execute packets in its delay slots cannot be interrupted. This is true regardless of whether the branch is taken.

Instruction Type

Branch

Delay Slots

5

Table 3–9 gives the program counter values and actions for the following code example. In this example, the B10 register holds the value 0000 0003.

Example

```
B10 | 0000 0003 |

0000 0000        B      .S2    B10
0000 0004        ADD    .L1    A1, A2, A3
0000 0008     || ADD    .L2    B1, B2, B3
0000 000C        MPY    .MIX   A3, B3, A4
0000 0010     || SUB    .D1    A5, A6, A6
0000 0014        MPY    .M1    A3, A6, A5
0000 0018        MPY    .M2    A6, A7, A8
0000 001C        SHR    .S1    A4, 15, A4
0000 0020        ADD    .D1    A4, A6, A4
```

Table 3–9. Program Counter Values for Branch Using a Register Example

Cycle	Program Counter Value	Action
Cycle 0	0000 0000h	Branch command executes (target code fetched)
Cycle 1	0000 0004h	
Cycle 2	0000 000Ch	
Cycle 3	0000 0014h	
Cycle 4	0000 0018h	
Cycle 5	0000 001Ch	
Cycle 6	0000 000Ch	Branch target code executes
Cycle 7	0000 0014h	

Syntax

B (.unit) IRP

.unit = .S2

Opcode map field used...	For operand type...	Unit	Opfield
src2	xsint	.S2	000011

Description

IRP is placed in the PFC. This instruction also moves PGIE to GIE. PGIE is unchanged.

If two branches are in the same execute packet and are both taken, behavior is undefined.

Two conditional branches can be in the same execute packet only if one of them is B using a register, B IRP, or B NRP. As long as only one branch has a true condition, code executes in a normal, well-defined way.

Execution

if (cond) IRP $\rightarrow$ PFC
else nop

Notes:

1) This instruction executes on .S2 only. PFC is the program fetch counter.
2) Refer to the chapter on interrupts for more information on IRP, PGIE, and GIE.
3) An execute packet containing a branch and the execute packets in its delay slots cannot be interrupted. This is true regardless of whether the branch is taken.

Instruction Type

Branch

Delay Slots

5

Table 3–10 gives the program counter values and actions for the following code example.

Example Given that an interrupt occurred at `PC = 0000 0100 → IRP = 0000 0100`:

```
0000 0000    B      .S2      IRP
0000 0004    ADD    .S1      A0, A2, A1
0000 0008    MPY    .M1      A1, A0, A1
0000 000C    NOP
0000 0010    SHR    .S1      A1, 15, A1
0000 0014    ADD    .L1      A1, A2, A1
0000 0018    ADD    .L2      B1, B2, B3
```

Table 3–10. Program Counter Values for B IRP Example

Cycle	Program Counter Value	Action
Cycle 0	0000 0000	Branch command executes (target code fetched)
Cycle 1	0000 0004	
Cycle 2	0000 0008	
Cycle 3	0000 000C	
Cycle 4	0000 0010	
Cycle 5	0000 0014	
Cycle 6	0000 0100	Branch target code executes

Syntax

B (.unit) NRP

.unit = .S2

Opcode map field used...	For operand type...	Unit	Opfield
src2	xsint	.S2	000011

Description

NRP is placed in the PFC. This instruction also sets NMIE. PGIE is unchanged.

If two branches are in the same execute packet and are both taken, behavior is undefined.

Two conditional branches can be in the same execute packet only if one of them is B using a register, B IRP, or B NRP. As long as only one branch has a true condition, code executes in a normal, well-defined way.

Execution

if (cond) NRP $\rightarrow$ PFC
else nop

Notes:

1) This instruction executes on .S2 only. PFC is program fetch counter.
2) Refer to the chapter on interrupts for more information on NRP and NMIE.
3) An execute packet containing a branch and the execute packets in its delay slots cannot be interrupted. This is true regardless of whether the branch is taken.

Instruction Type Branch

Delay Slots 5

Table 3–11 gives the program counter values and actions for the following code example.

Example

Given that an interrupt occurred at PC = 0000 0100 →NRP = 0000 0100:

```
0000 0000    B      .S2       NRP
0000 0004    ADD    .S1       A0, A2, A1
0000 0008    MPY    .M1       A1, A0, A1
0000 000C    NOP
0000 0010    SHR    .S1       A1, 15, A1
0000 0014    ADD    .L1       A1, A2, A1
0000 0018    ADD    .L2       B1, B2, B3
```

Table 3–11. Program Counter Values for B NRP Example

Cycle	Program Counter Value	Action
Cycle 0	0000 0000	Branch command executes (target code fetched)
Cycle 1	0000 0004	
Cycle 2	0000 0008	
Cycle 3	0000 000C	
Cycle 4	0000 0010	
Cycle 5	0000 0014	
Cycle 6	0000 0100	Branch target code executes

Syntax

CLR (.unit) *src2*, *csta*, *cstb*, *dst*
or
CLR (.unit) *src2*, *src1*, *dst*

.unit = .S1 or .S2

Opcode

Constant form:

31–29	28	27–23	22–18	17–13	12–8	7–6	5–0		
creg	*z*	*dst*	*src2*	*csta*	*cstb*	1 0	0 0 1 0	*s*	*p*
3	1	5	5	5	5	2		1	1

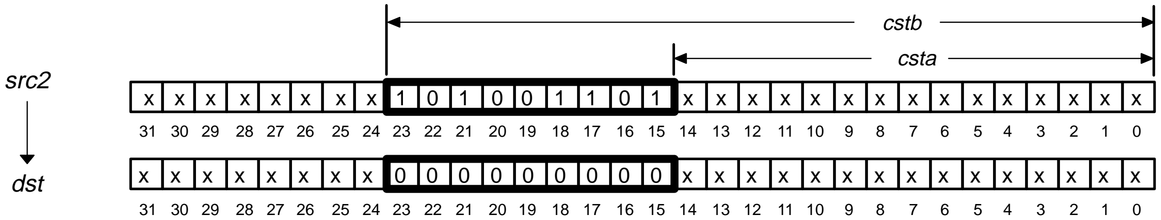
Register form:

31–29	28	27–23	22–18	17–13	12	11–6	5–0		
creg	*z*	*dst*	*src2*	*src1*	*x*	1 1 1 0 1 1	1 0 0 0	*s*	*p*
3	1	5	5	5	1	6		1	1

Opcode map field used...	For operand type...	Unit	Opfield
src2 *csta* *cstb* *dst*	uint *ucst*5 *ucst*5 uint	.S1, .S2	11
src2 *src1* *dst*	xuint uint uint	.S1, .S2	111111

Description

The field in *src2*, specified by *csta* and *cstb*, is cleared to zero. *csta* and *cstb* may be specified as constants or as the ten LSBs of the *src1* registers, with *cstb* being bits 0–4 and *csta* bits 5–9. *csta* signifies the bit location of the LSB in the field and *cstb* signifies the bit location of the MSB in the field. In other words, *csta* and *cstb* represent the beginning and ending bits, respectively, of the field to be cleared. The LSB location of *src2* is 0 and the MSB location of *src2* is 31. In the example below, *csta* is 15 and *cstb* is 23.



Execution

If the constant form is used:

if (cond)	*src2* clear *csta*, *cstb* $\rightarrow$ *dst*
else	nop

If the register form is used:

if (cond)	*src2* clear $src1_{9..5}$, $src1_{4..0}$ $\rightarrow$ *dst*
else	nop

Instruction Type Single-cycle

Delay Slots 0

Example 1

```
CLR .S1    A1,4,19,A2
```

	Before instruction	1 cycle after instruction
A1	07A4 3F2Ah	07A4 3F2Ah
A2	XXXX XXXXh	07A0 000Ah

Example 2

```
CLR .S2    B1,B3,B2
```

	Before instruction	1 cycle after instruction
B1	03B6 E7D5h	03B6 E7D5h
B2	XXXX XXXXh	03B0 0001h
B3	0000 0052h	0000 0052h

Syntax

CMPEQ (.unit) *src1*, *src2*, *dst*

.unit = .L1 or .L2

Opcode map field used...	For operand type...	Unit	Opfield
src1 *src2* *dst*	sint xsint uint	.L1, .L2	1010011
src1 *src2* *dst*	*scst*5 xsint uint	.L1, .L2	1010010
src1 *src2* *dst*	xsint slong uint	.L1, .L2	1010001
src1 *src2* *dst*	*scst*5 slong uint	.L1, .L2	1010000

Description

This instruction compares *src1* to *src2*. If *src1* equals *src2*, then 1 is written to *dst*. Otherwise, 0 is written to *dst*.

Execution

```
if (cond)    {
             if (src1 == src2) 1 → dst
             else 0 → dst
             }
else         nop
```

Instruction Type

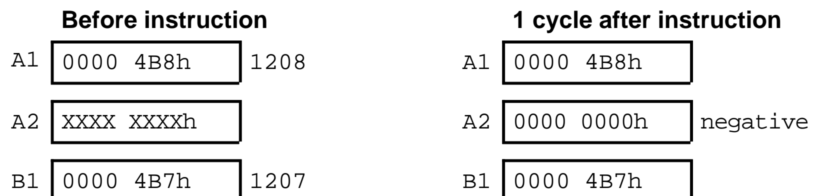
Single-cycle

Delay Slots

0

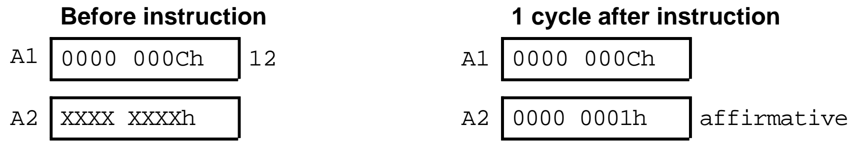
Example 1

```
CMPEQ .L1  A1,B1,B2
```



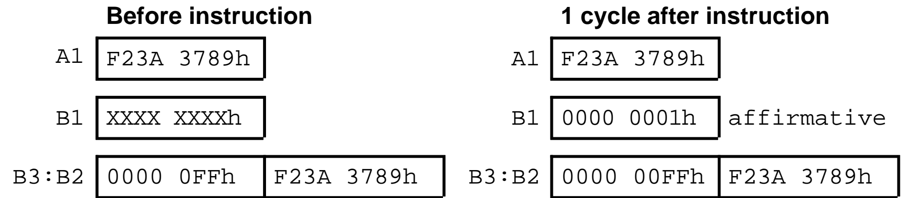
Example 2

```
CMPEQ .L1  3519,A1,A2
```



Example 3

```
CMPEQ .L1  A1,B3:B2,B1
```



Syntax

CMPGT (.unit) *src1*, *src2*, *dst*
or
CMPGTU (.unit) *src1*, *src2*, *dst*

.unit = .L1 or .L2

Opcode map field used...	For operand type...	Unit	Opfield	Mnemonic
src1 *src2* *dst*	sint xsint uint	.L1, .L2	1000111	CMPGT
src1 *src2* *dst*	*scst*5 xsint uint	.L1, .L2	1000110	CMPGT
src1 *src2* *dst*	xsint slong uint	.L1, .L2	1000101	CMPGT
src1 *src2* *dst*	*scst*5 slong uint	.L1, .L2	1000100	CMPGT
src1 *src2* *dst*	uint xuint uint	.L1, .L2	1001111	CMPGTU
src1 *src2* *dst*	*ucst*4 xuint uint	.L1, .L2	1001110	CMPGTU
src1 *src2* *dst*	xuint ulong uint	.L1, .L2	1001101	CMPGTU
src1 *src2* *dst*	*ucst*4 ulong uint	.L1, .L2	1001100	CMPGTU

Description

This instruction does a signed or unsigned comparison of *src1* to *src2*. If *src1* is greater than *src2*, then 1 is written to *dst*. Otherwise, 0 is written to *dst*.

Execution

if (cond) {
if (*src1* > *src2*) 1 → *dst*
else 0 → *dst*
}
else nop

Instruction Type

Single-cycle

Delay Slots

0

Example 1

```
CMPGT .L1  A1,B1,A2
```

	Before instruction			1 cycle after instruction	
A1	0000 01B6h	438	A1	0000 01B6h	
A2	XXXX XXXXh		A2	0000 0000h	negative
B1	0000 08BDh	2237	B1	0000 08BDh	

Example 2

```
CMPGT .L1  A1,B1,A2
```

	Before instruction			1 cycle after instruction	
A1	FFFF FE91h	−367	A1	FFFF FE91h	
A2	XXXX XXXXh		A2	0000 0001h	affirmative
B1	FFFF FDC4h	−572	B1	FFFF FDC4h	

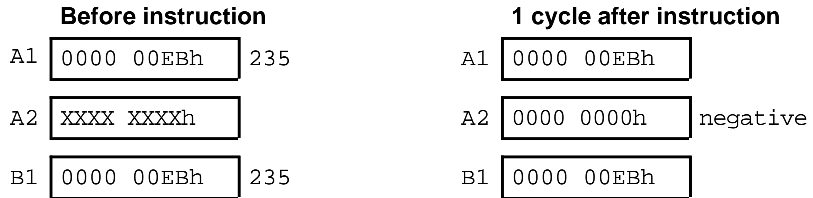
Example 3

```
CMPGT .L1  8,A1,A2
```

	Before instruction			1 cycle after instruction	
A1	0000 0023h	35	A1	0000 0023h	
A2	XXXX XXXXh		A2	0000 0000h	negative

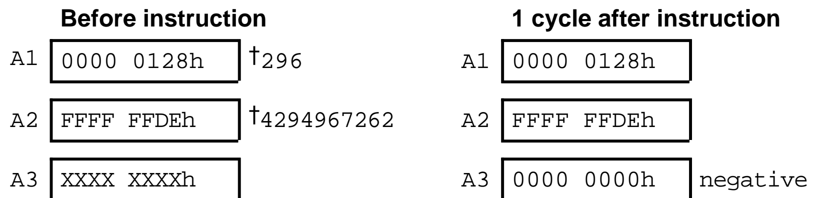
Example 4

```
CMPGT .L1  A1,B1,A2
```



Example 5

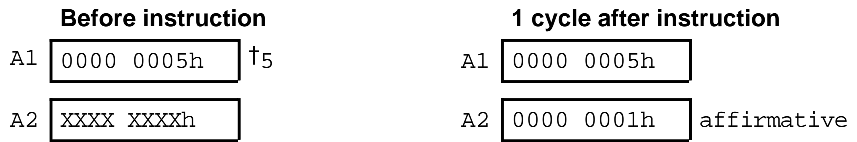
```
CMPGTU .L1 A1,A2,A3
```



† Unsigned 32-bit integer

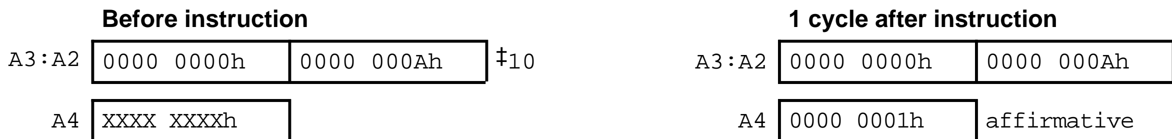
Example 6

```
CMPGTU .L1 0Ah,A1,A2
```



Example 7

```
CMPGTU .L1 0Eh,A3:A2,A4
```



† Unsigned 32-bit integer
‡ Unsigned 40-bit (long) integer

Syntax

CMPLT (.unit) *src1*, *src2*, *dst*
or
CMPLTU (.unit) *src1*, *src2*, *dst*

.unit = .L1 or .L2

Opcode map field used...	For operand type...	Unit	Opfield	Mnemonic
src2 *src1* *dst*	sint xsint uint	.L1, .L2	1010111	CMPLT
src2 *src1* *dst*	*scst*5 xsint uint	.L1, .L2	1010110	CMPLT
src2 *src1* *dst*	xsint slong uint	.L1, .L2	1010101	CMPLT
src2 *src1* *dst*	*scst*5 slong uint	.L1, .L2	1010100	CMPLT
src1 *src2* *dst*	uint xuint uint	.L1, .L2	1011111	CMPLTU
src1 *src2* *dst*	*ucst*4 xuint uint	.L1, .L2	1011110	CMPLTU
src1 *src2* *dst*	xuint ulong uint	.L1, .L2	1011101	CMPLTU
src1 *src2* *dst*	*ucst*4 ulong uint	.L1, .L2	1011100	CMPLTU

Description

This instruction does a signed or unsigned comparison of *src1* to *src2*. If *src1* is less than *src2*, then 1 is written to *dst*. Otherwise, 0 is written to *dst*.

Execution

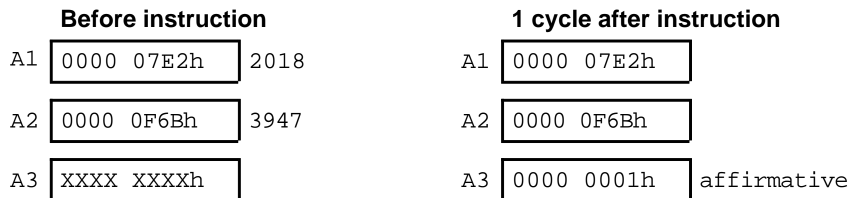
```
if (cond)    {
             if (src1 < src2) 1 → dst
             else 0 → dst
             }
else         nop
```

Instruction Type Single-cycle

Delay Slots 0

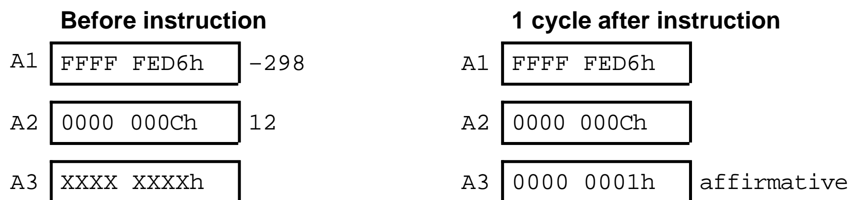
Example 1

```
CMPLT .L1  A1,A2,A3
```



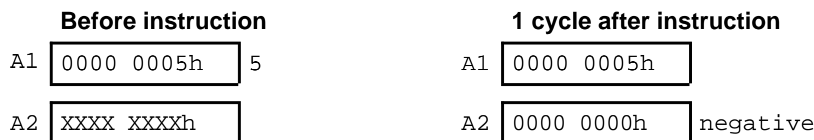
Example 2

```
CMPLT .L1  A1,A2,A3
```



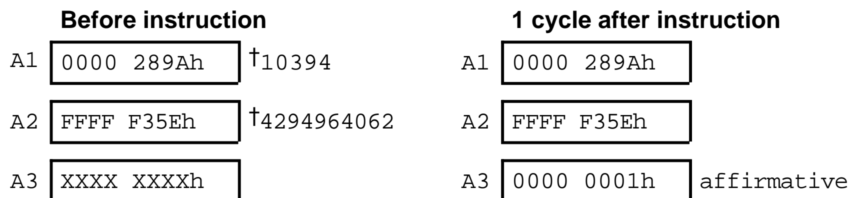
Example 3

```
CMPLT .L1  9,A1,A2
```



Example 4

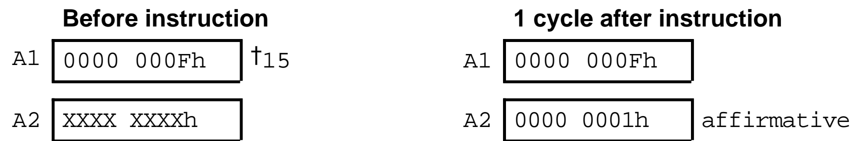
```
CMPLTU .L1 A1,A2,A3
```



† Unsigned 32-bit integer

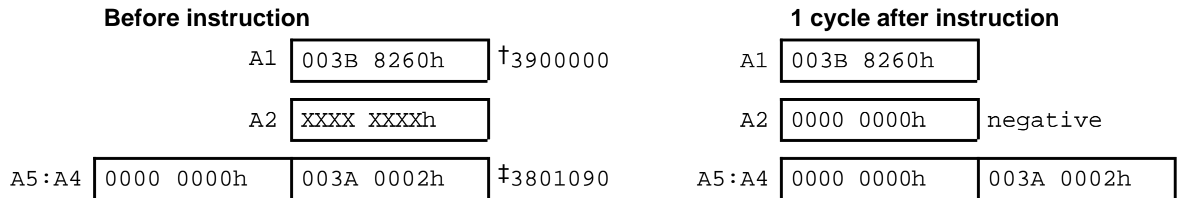
Example 5

```
CMPLTU .L1 14,A1,A2
```



Example 6

```
CMPLTU .L1 A1,A5:A4,A2
```



† Unsigned 32-bit integer
‡ Unsigned 40-bit (long) integer

Syntax

EXT (.unit) *src2*, *csta*, *cstb*, *dst*
or
EXT (.unit) *src2*, *src1*, *dst*

.unit = .S1 or .S2

Opcode

Constant form:

31–29	28	27–23	22–18	17–13	12–8	7–6	5–0		
creg	*z*	*dst*	*src2*	*csta*	*cstb*	0 1	0 0 1 0	*s*	*p*
3	1	5	5	5	5	2		1	1

Register form:

31–29	28	27–23	22–18	17–13	12	11–6	5–0		
creg	*z*	*dst*	*src2*	*src1*	*x*	1 0 1 1 1 1	1 0 0 0	*s*	*p*
3	1	5	5	5		6		1	1

Opcode map field used...	For operand type...	Unit	Opfield
src2 *csta* *cstb* *dst*	sint *ucst*5 *ucst*5 sint	.S1, .S2	01
src2 *src1* *dst*	xsint uint sint	.S1, .S2	101111

Description

The field in *src2*, specified by *csta* and *cstb*, is extracted and sign-extended to 32 bits. The extract is performed by a shift left followed by a signed shift right. *csta* and *cstb* are the shift left amount and shift right amount respectively. This can be thought of in terms of the LSB and MSB of the field to be extracted. Then *csta* = 31 – MSB of the field and cstb = LSB of the field + *csta*. The shift left and shift right amounts may also be specified as the ten LSBs of the *src1* register with *cstb* being bits 0–4 and *csta* bits 5–9. In the example below, *csta* is 8 and *cstb* is 15 + 8, or 23.

csta | cstb – csta

src2 1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
x	x	x	x	x	x	x	x	1	0	1	0	0	1	1	0	1	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x

Shifts left by 8 to produce:

2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	1	1	0	1	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	0	0	0	0	0	0	0	0

Then shifts right by 23 to produce:

dst 3)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	0	0	1	1	0	1

Execution

If the constant form is used:

if (cond) *src2* exts *csta*, *cstb* → *dst*
else nop

If the register form is used:

if (cond) *src2* exts $src1_{9..5}$, $src1_{4..0}$ → *dst*
else nop

Instruction Type Single-cycle

Delay Slots 0

Example 1

```
EXT .S1    A1,10,19,A2
```

	Before instruction		1 cycle after instruction
A1	07A4 3F2Ah	A1	07A4 3F2Ah
A2	XXXX XXXXh	A2	FFFF F21Fh

Example 2

```
EXT .S1    A1,A2,A3
```

	Before instruction		1 cycle after instruction
A1	03B6 E7D5h	A1	03B6 E7D5h
A2	0000 0073h	A2	0000 0073h
A3	XXXX XXXXh	A3	0000 03B6h

Syntax

EXTU (.unit) *src2*, *csta*, *cstb*, *dst*
or
EXTU (.unit) *src2*, *src1*, *dst*

.unit = .S1 or .S2

Opcode

Constant width and offset form:

31–29	28	27–23	22–18	17–13	12–8	7–6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	*cstb*	*op*	0	0	1	0	*s*	*p*
3	1	5	5	5	5	2					1	1

Register width and offset form:

31–29	28	27–23	22–18	17–13	12	11–6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	*x*	*op*	1	0	0	0	*s*	*p*
3	1	5	5	5		6					1	1

Opcode map field used...	For operand type...	Unit	Opfield
src2 *csta* *cstb* *dst*	uint *ucst*5 *ucst*5 uint	.S1, .S2	00
src2 *src1* *dst*	xuint uint uint	.S1, .S2	101011

Description

The field in *src2*, specified by *csta* and *cstb*, is extracted and zero extended to 32 bits. The extract is performed by a shift left followed by an unsigned shift right. *csta* and *cstb* are the amounts to shift left and shift right respectively. This can be thought of in terms of the LSB and MSB of the field to be extracted. Then *csta* = 31 – MSB of the field and cstb = LSB of the field + *csta*. The shift left and shift right amounts may also be specified as the ten LSBs of the *src1* register with *cstb* being bits 0–4 and *csta* bits 5–9. In the example below, *csta* is 8 and *cstb* is 15+8, or 23.

csta (bits 31–24) *cstb – cst a* (bits 14–0)

src2 1)

x	x	x	x	x	x	x	x	1	0	1	0	0	1	1	0	1	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Shifts left by 8 to produce:

2)

1	0	1	0	0	1	1	0	1	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	0	0	0	0	0	0	0	0
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Then shifts right by 23 to produce:

dst 3)

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	1	1	0	1
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Execution

If the constant form is used:

if (cond) *src2* extu *csta*, *cstb* → *dst*
else nop

If the register width and offset form is used:

if (cond) *src2* extu $src1_{9..5}$, $src1_{4..0}$ → *dst*
else nop

Instruction Type Single-cycle

Delay Slots 0

Example 1

```
EXTU .S1   A1,10,19,A2
```

	Before instruction		1 cycle after instruction
A1	07A4 3F2Ah	A1	07A4 3F2Ah
A2	XXXX XXXXh	A2	0000 121Fh

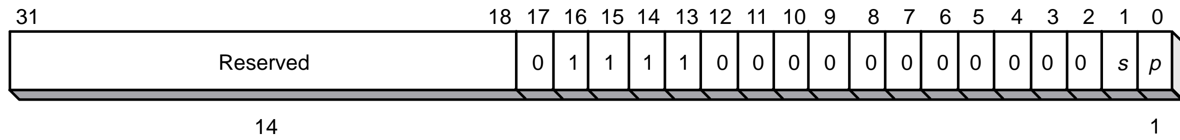
Example 2

```
EXTU .S1   A1,A2,A3
```

	Before instruction		1 cycle after instruction
A1	03B6 E7D5h	A1	03B6 E7D5h
A2	0000 0156h	A2	0000 0156h
A3	xxxx xxxxh	A3	0000 036Eh

Syntax **IDLE**

Opcode



Description This instruction performs a multicycle **NOP** that terminates only upon servicing an interrupt.

Instruction Type NOP

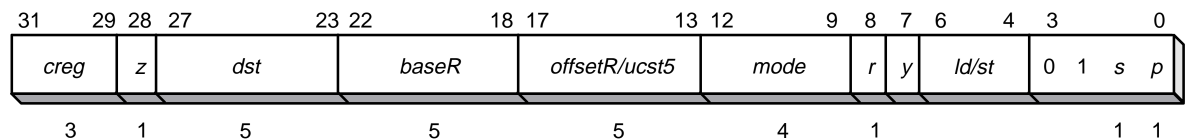
Delay Slots 0

Syntax

Register Offset	Unsigned Constant Offset
LDB (.unit) *+*baseR[offsetR]*, *dst*	**LDB** (.unit) *+*baseR[ucst5]*, *dst*
or	or
LDH (.unit) *+*baseR[offsetR]*, *dst*	**LDH** (.unit) *+*baseR[ucst5]*, *dst*
or	or
LDW (.unit) *+*baseR[offsetR]*, *dst*	**LDW** (.unit) *+*baseR[ucst5]*, *dst*
or	or
LDBU (.unit) *+*baseR[offsetR]*, *dst*	**LDBU** (.unit) *+*baseR[ucst5]*, *dst*
or	or
LDHU (.unit) *+*baseR[offsetR]*, *dst*	**LDHU** (.unit) *+*baseR[ucst5]*, *dst*

.unit = .D1 or .D2

Opcode



Description Each of these instructions performs a load from memory to a general-purpose register (*dst*). Table 3–12 summarizes the data types supported by loads. Table 3–13 describes the addressing generator options. The memory address is formed from a base address register (*baseR*) and an optional offset that is either a register (*offsetR*) or a 5-bit unsigned constant (*ucst5*). If an offset is not given, the assembler assigns an offset of zero.

offsetR and *baseR* must be in the same register file and on the same side as the .D unit used. The *y* bit in the opcode determines the .D unit and register file used: *y* = 0 selects the .D1 unit and *baseR* and *offsetR* from the A register file and *y* = 1 selects the .D2 unit and *baseR* and *offsetR* from the B register file.

offsetR/ucst5 is scaled by a left-shift of 0, 1, or 2 for **LDB(U)**, **LDH(U)**, and **LDW**, respectively. After scaling, *offsetR/ucst5* is added to or subtracted from *baseR*. For the preincrement, predecrement, positive offset, and negative offset address generator options, the result of the calculation is the address to be accessed in memory. For postincrement or postdecrement addressing, the value of *baseR* before the addition or subtraction is the address to be accessed in memory.

The addressing arithmetic that performs the additions and subtractions defaults to linear mode. However, for A4–A7 and for B4–B7, the mode can be changed to circular mode by writing the appropriate value to the AMR (see section 2.3).

For **LDH(U)** and **LDB(U)** the values are loaded into the 16 and 8 LSBs of *dst*, respectively. For **LDH** and **LDB**, the upper 16- and 24-bits, respectively, of *dst* values are sign-extended. For **LDHU** and **LDBU** loads, the upper 16- and 24-bits, respectively, of *dst* are zero-filled. For **LDW**, the entire 32 bits fills *dst*. *dst* can be in either register file, regardless of the .D unit or *baseR* or *offsetR* used. The *s* bit determines which file *dst* will be loaded into: *s* = 0 indicates *dst* will be in the A register file and *s* = 1 indicates *dst* will be loaded in the B register file.

Table 3–12. Data Types Supported by Loads

Mnemonic	*ld/st* Field	Load Data Type	SIze	Left Shift of Offset
LDB	0 1 0	Load byte	8	0 bits
LDBU	0 0 1	Load byte unsigned	8	0 bits
LDH	1 0 0	Load halfword	16	1 bit
LDHU	0 0 0	Load halfword unsigned	16	1 bit
LDW	1 1 0	Load word	32	2 bits

Table 3–13. Address Generator Options

Mode Field	Syntax	Modification Performed
0 1 0 1	*+R[*offsetR*]	Positive offset
0 1 0 0	*–R[*offsetR*]	Negative offset
1 1 0 1	*++R[*offsetR*]	Preincrement
1 1 0 0	*––R[*offsetR*]	Predecrement
1 1 1 1	*R++[*offsetR*]	Postincrement
1 1 1 0	*R––[*offsetR*]	Postdecrement
0 0 0 1	*+R[*ucst5*]	Positive offset
0 0 0 0	*–R[*ucst5*]	Negative offset
1 0 0 1	*++R[*ucst5*]	Preincrement
1 0 0 0	*– –R[*ucst5*]	Predecrement
1 0 1 1	*R++[*ucst5*]	Postincrement
1 0 1 0	*R– –[*ucst5*]	Postdecrement

Increments and decrements default to 1 and offsets default to zero when no bracketed register or constant is specified. Loads that do no modification to the *baseR* can use the syntax *R. Square brackets, [ ], indicate that the *ucst*5 offset is left-shifted by 2, 1, or 0 for word, halfword, and byte loads, respectively. Parentheses, ( ), can be used to set a nonscaled, constant offset. For example, **LDW** (.unit) *+*baseR* (12) *dst* represents an offset of 12 bytes, whereas **LDW** (.unit) *+*baseR* [12] *dst* represents an offset of 12 words, or 48 bytes.

Execution

if (cond)   mem → *dst*
else   nop

Instruction Type Load

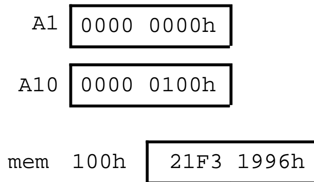
Delay Slots

4 for loaded value
0 for address modification from pre/post increment/decrement
For more information on delay slots for a load, see Chapter 4, *TMS320C62xx Pipeline*.

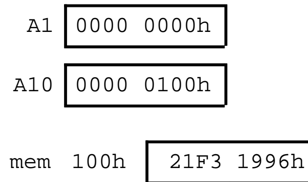
Example 1

```
LDW .D1    *A10,A1
```

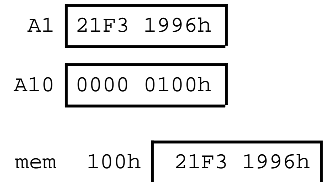
Before LDW



1 cycle after LDW



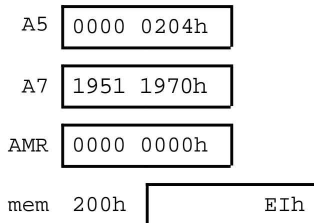
5 cycles after LDW



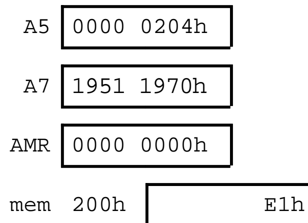
Example 2

```
LDB .D1    *-A5[4],A7
```

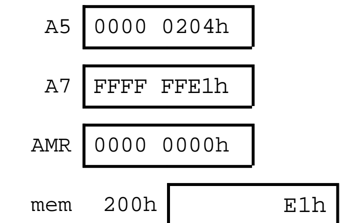
Before LDB



1 cycle after LDB



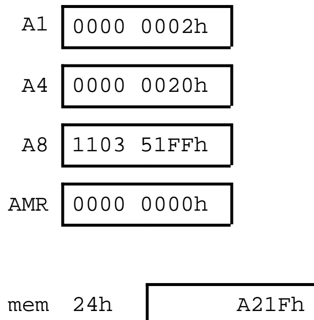
5 cycles after LDB



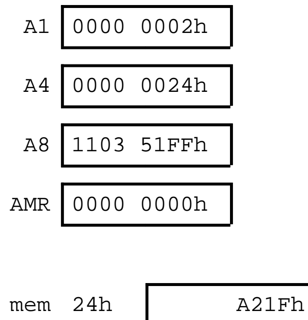
Example 3

```
LDH .D1    *++A4[A1],A8
```

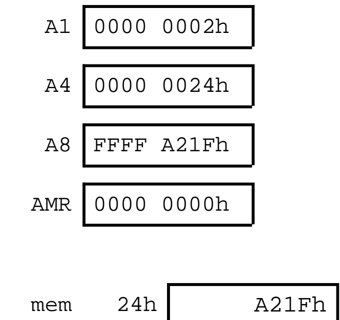
Before LDH



1 cycle after LDH



5 cycles after LDH



Example 4 `LDW .D1 *A4,++[1],A6`

	Before LDW	1 cycle after LDW	5 cycles after LDW
A4	0000 0100h	0000 0104h	0000 0104h
A6	1234 4321h	1234 4321h	0798 F25Ah
AMR	0000 0000h	0000 0000h	0000 0000h
mem 100h	0798 F25Ah	0798 F25Ah	0798 F25Ah
mem 104h	1970 19F3h	1970 19F3h	1970 19F3

Example 5 `LDW .D1 *++A4(4),A6`

	Before LDW	1 cycle after LDW	5 cycles after LDW
A4	0000 0100h	0000 0104h	0000 0104h
A6	1234 5678h	1234 5678h	0217 6991h
AMR	0000 0000h	0000 0000h	0000 0000h
mem 104h	0217 6991h	0217 6991h	0217 6991h

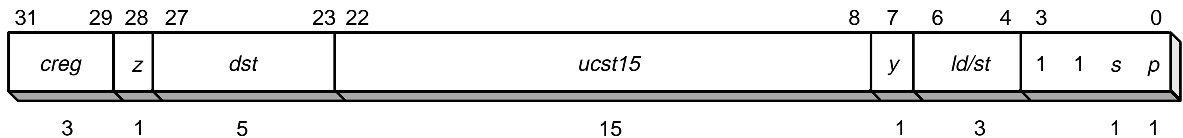
Syntax

LDB (.unit) *+B14/B15[*ucst15*], *dst*
or
LDH (.unit) *+B14/B15[*ucst15*], *dst*
or
LDW (.unit) *+B14/B15[*ucst15*], *dst*

LDBU (.unit) *+B14/B15[*ucst15*], *dst*
or
LDHU (.unit) *+B14/B15[*ucst15*], *dst*

.unit = .D2

Opcode



Description

Each of these instructions performs a load from memory to a general-purpose register (*dst*). Table 3–14 summarizes the data types supported by loads. The memory address is formed from a base address register (*baseR*) B14 (*y* = 0) or B15 (*y* = 1) and an offset, which is a 15-bit unsigned constant (*ucst15*). The assembler selects this format only when the constant is larger than five bits in magnitude. This instruction operates only on the .D2 unit.

ucst15 is scaled by a left shift of 0, 1, or 2 for **LDB(U)**, **LDH(U)**, and **LDW**, respectively. After scaling, *ucst15* is added to or subtracted from *baseR*. The result of the calculation is the address sent to memory. The addressing arithmetic is always performed in linear mode.

For **LDH(U)** and **LDB(U)**, the values are loaded into the 16 and 8 LSBs of *dst*, respectively. For **LDH** and **LDB**, the upper 16 and 24 bits of *dst* values are sign-extended, respectively. For **LDHU** and **LDBU** loads, the upper 16 and 24 bits of *dst* are zero-filled, respectively. For **LDW**, the entire 32 bits fills *dst*. *dst* can be in either register file. The *s* bit determines which file *dst* will be loaded into: *s* = 0 indicates *dst* is loaded in the A register file, and *s* = 1 indicates *dst* is loaded into the B register file.

Square brackets, [ ], indicate that the *ucst*15 offset is left-shifted by 2, 1, or 0 for word, halfword, and byte loads, respectively. Parentheses, ( ), can be used to set a nonscaled, constant offset. For example, **LDW** (.unit) *+B14/B15(60) *dst* represents an offset of 60 bytes, whereas **LDW** (.unit) *+B14/B15[60] *dst* represents an offset of 60 words, or 240 bytes.

Table 3–14. Data Types Supported by Loads

Mnemonic	*ld/st* Field	Load Data Type	SIze	Left Shift of Offset
LDB	0 1 0	Load byte	8	0 bits
LDBU	0 0 1	Load byte unsigned	8	0 bits
LDH	1 0 0	Load halfword	16	1 bit
LDHU	0 0 0	Load halfword unsigned	16	1 bit
LDW	1 1 0	Load word	32	2 bits

Execution

if (cond) mem → *dst*
else nop

Note:

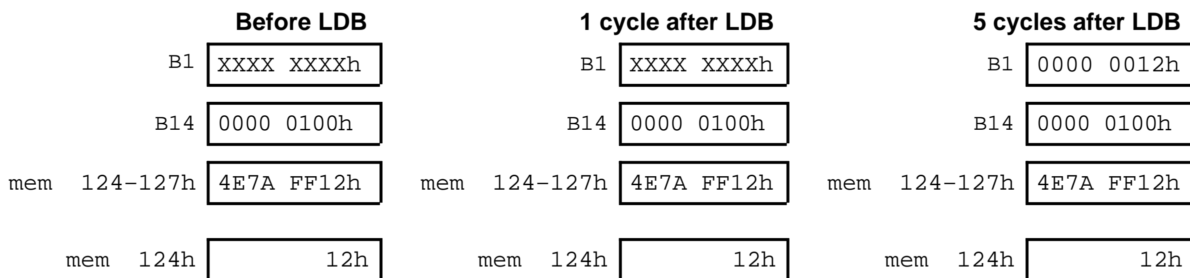
This instruction executes only on the B side (.D2).

Instruction Type Load

Delay Slots 4

Example

```
LDB .D2    *+B14[36],B1
```



Syntax

LMBD (.unit) *src1*, *src2*, *dst*

.unit = .L1 or .L2

Opcode map field used...	For operand type...	Unit	Opfield
src1 *src2* *dst*	uint xuint uint	.L1, .L2	1101011
src1 *src2* *dst*	*cst*5 xuint uint	.L1, .L2	1101010

Description

The LSB of the *src1* operand determines whether to search for a leftmost 1 or 0 in *src2*. The number of bits to the left of the first 1 or 0 when searching for a 1 or 0, respectively, is placed in *dst*.

The following diagram illustrates the operation of **LMBD** for several cases.

When searching for 0 in *src2*, **LMBD** returns 0:

0	1	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

When searching for 1 in *src2*, **LMBD** returns 4:

0	0	0	0	1	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

When searching for 0 in *src2*, **LMBD** returns 32:

1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Execution

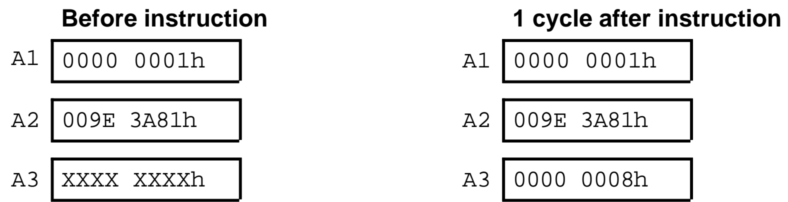
if (cond) {
  if ($src1_0$ == 0) lmb0(*src2*) → *dst*
  if ($src1_0$ == 1) lmb1(*src2*) → *dst*
}
else nop

Instruction Type Single-cycle

Delay Slots 0

Example

```
LMBD    A1,A2,A3
```



Syntax

MPY(U/US/SU) (.unit) *src1*, *src2*, *dst*

.unit = .M1 or .M2

Opcode map field used...	For operand type...	Unit	Opfield	Mnemonic
src1 *src2* *dst*	slsb16 xslsb16 sint	.M1, .M2	11001	MPY
src1 *src2* *dst*	ulsb16 xulsb16 uint	.M1, .M2	11111	MPYU
src1 *src2* *dst*	ulsb16 xslsb16 sint	.M1, .M2	11101	MPYUS
src1 *src2* *dst*	slsb16 xulsb16 sint	.M1, .M2	11011	MPYSU
src1 *src2* *dst*	*scst*5 xslsb16 sint	.M1, .M2	11000	MPY
src1 *src2* *dst*	*scst*5 xulsb16 sint	.M1, .M2	11110	MPYSU

Description

The *src1* operand is multiplied by the *src2* operand. The result is placed in *dst*. The source operands are signed by default. The S is needed in the mnemonic to specify a signed operand when both signed and unsigned operands are used.

Execution

if (cond) lsb16(*src1*) $\times$ lsb16(*src2*) $\rightarrow$ *dst*
else nop

Instruction Type

Multiply

Delay Slots

1

Example 1

```
MPY .M1    A1,A2,A3
```

	Before instruction			2 cycles after instruction	
A1	0000 0123h	†291	A1	0000 0123h	
A2	01E0 FA81h	†−1407	A2	01E0 FA81h	
A3	XXXX XXXXh		A3	FFF9 C0A3	−409437

Example 2

```
MPYU .M1   A1,A2,A3
```

	Before instruction			2 cycles after instruction	
A1	0000 0123h	‡291	A1	0000 0123h	
A2	0F12 FA81h	‡64129	A2	0F12 FA81h	
A3	XXXX XXXXh		A3	011C C0A3	§18661539

Example 3

```
MPYUS .M1  A1,A2,A3
```

	Before instruction			2 cycles after instruction	
A1	1234 FFA1h	‡65441	A1	1234 FFA1h	
A2	1234 FFA1h	†−95	A2	1234 FFA1h	
A3	XXXX XXXXh		A3	FFA1 2341h	−6216895

Example 4

```
MPY .M1    13,A1,A2
```

	Before instruction			2 cycles after instruction	
A1	3497 FFF3h	†−13	A1	3497 FFF3h	
A2	XXXX XXXXh		A2	FFFF FF57h	−163

Example 5

```
MPYSU .M1  13,A1,A2
```

	Before instruction			2 cycles after instruction	
A1	3497 FFF3h	‡65523	A1	3497 FFF3h	
A2	XXXX XXXXh		A2	000C FF57h	851779

† Signed 16-LSB integer
‡ Unsigned 16-LSB integer
§ Unsigned 32-bit integer

Syntax

MPYH(U/US/SU) (.unit) *src1*, *src2*, *dst*

.unit = .M1 or .M2

Opcode map field used...	For operand type...	Unit	Opfield	Mnemonic
src1 *src2* *dst*	smsb16 xsmsb16 sint	.M1, .M2	00001	MPYH
src1 *src2* *dst*	umsb16 xumsb16 uint	.M1, .M2	00111	MPYHU
src1 *src2* *dst*	umsb16 xsmsb16 sint	.M1, .M2	00101	MPYHUS
src1 *src2* *dst*	smsb16 xumsb16 sint	.M1, .M2	00011	MPYHSU

Description

The *src1* operand is multiplied by the *src2* operand. The result is placed in *dst*. The source operands are signed by default. The S is needed in the mnemonic to specify a signed operand when both signed and unsigned operands are used.

Execution

if (cond) msb16(*src1*) $\times$ msb16(*src2*) $\rightarrow$ *dst*
else nop

Instruction Type

Multiply

Delay Slots

1

Example 1

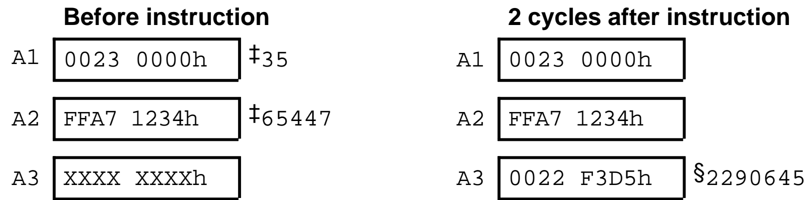
```
MPYH .M1   A1,A2,A3
```

	Before instruction			2 cycles after instruction	
A1	0023 0000h	†35	A1	0023 0000h	
A2	FFA7 1234h	†−89	A2	FFA7 1234h	
A3	XXXX XXXXh		A3	FFFF F3D5h	−3115

† Signed integer, 16 MSBs

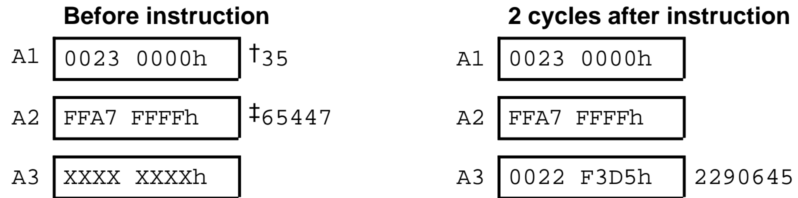
Example 2

```
MPYHU .M1  A1,A2,A3
```



Example 3

```
MPYHSU .M1 A1,A2,A3
```



† Signed integer, 16 MSBs
‡ Unsigned integer, 16 MSBs
§ Unsigned 32-bit integer

Syntax **MPYH(U/S)L(U/S)** (.unit) *src1*, *src2*, *dst*

.unit = .M1 or .M2

Opcode map field used...	For operand type...	Unit	Opfield	Mnemonic
src1 *src2* *dst*	smsb16 xslsb16 sint	.M1, .M2	01001	MPYHL
src1 *src2* *dst*	umsb16 xulsb16 uint	.M1, .M2	01111	MPYHLU
src1 *src2* *dst*	umsb16 xslsb16 sint	.M1, .M2	01101	MPYHULS
src1 *src2* *dst*	smsb16 xulsb16 sint	.M1, .M2	01011	MPYHSLU

Description The *src1* operand is multiplied by the *src2* operand. The result is placed in *dst*. The source operands are signed by default. The S is needed in the mnemonic to specify a signed operand when both signed and unsigned operands are used.

Execution

if (cond) msb16(*src1*) $\times$ lsb16(*src2*) $\rightarrow$ *dst*
else nop

Instruction Type Multiply

Delay Slots 1

Example

```
MPYHL .M1  A1,A2,A3
```

	Before instruction			2 cycles after instruction	
A1	008A 003Eh	†138	A1	008A 003Eh	
A2	21FF 00A7h	‡167	A2	21FF 00A7h	
A3	XXXX XXXXh		A3	0000 5A06h	23046

† Signed integer, 16 MSBs
‡ Signed integer, 16 LSBs

Syntax **MPYL(U/S)H(U/S)** (.unit) *src1*, *src2*, *dst*

.unit = .M1 or .M2

Opcode map field used...	For operand type...	Unit	Opfield	Mnemonic
src1 *src2* *dst*	slsb16 xsmsb16 sint	.M1, .M2	10001	MPYLH
src1 *src2* *dst*	ulsb16 xumsb16 uint	.M1, .M2	10111	MPYLHU
src1 *src2* *dst*	ulsb16 xsmsb16 sint	.M1, .M2	10101	MPYLUHS
src1 *src2* *dst*	slsb16 xumsb16 sint	.M1, .M2	10011	MPYLSHU

Description The *src1* operand is multiplied by the *src2* operand. The result is placed in *dst*. The source operands are signed by default. The S is needed in the mnemonic to specify a signed operand when both signed and unsigned operands are used.

Execution

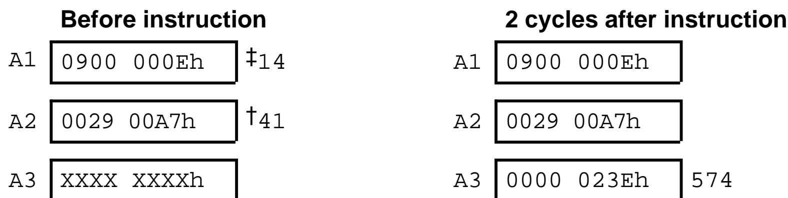
if (cond) lsb16(*src1*) $\times$ msb16(*src2*) $\rightarrow$ *dst*
else nop

Instruction Type Multiply

Delay Slots 1

Example

```
MPYLH .M1  A1,A2,A3
```



	Before instruction			2 cycles after instruction	
A1	0900 000Eh	‡14	A1	0900 000Eh	
A2	0029 00A7h	†41	A2	0029 00A7h	
A3	XXXX XXXXh		A3	0000 023Eh	574

† Signed integer, 16 MSBs
‡ Signed integer, 16 LSBs

Syntax **MV** (.unit) *src, dst*

.unit = .L1, .L2, .S1, .S2, .D1, .D2

Description This is a pseudo operation that moves a value from one register to another. The assembler uses the operation **ADD** (.unit) 0, *src*, *dst* to perform this task.

Execution if (cond) 0 + *src* → dst
else nop

Instruction Type Single-cycle

Delay Slots 0

Syntax **MVC** (.unit) *src2*, *dst*

.unit = .S2

Operands when moving from the control file to the register file:

Opcode map field used...	For operand type...	Unit	Opfield
src2 *dst*	uint uint	.S2	001111

Description The *src2* register is moved from the control register file to the register file. Valid values for *src2* are any register listed in the control register file.

Operands when moving from the register file to the control file:

Opcode map field used...	For operand type...	Unit	Opfield
src2 *dst*	xuint uint	.S2	001110

Description The *src2* register is moved from the register file to the control register file. Valid values for *src2* are any register listed in the control register file.

Register addresses for accessing the control registers are in Table 3–15.

Table 3–15. Register Addresses for Accessing the Control Registers

Register Abbreviation	Name	Register Address	Read/ Write
AMR	Addressing mode register	00000	R, W
CSR	Control status register	00001	R, W
IFR	Interrupt flag register	00010	R
ISR	Interrupt set register	00010	W
ICR	Interrupt clear register	00011	W
IER	Interrupt enable register	00100	R, W
ISTP	Interrupt service table pointer	00101	R, W
IRP	Interrupt return pointer	00110	R, W
NRP	Nonmaskable interrupt return pointer	00111	R, W
PCE1	Program counter, E1 phase	10000	R

Note: R = Readable by the **MVC** instruction
W = Writeable by the **MVC** instruction

Execution

if (cond) *src* → *dst*
else nop

Note:

The **MVC** instruction executes only on the B side (.S2).

Instruction Type

Single-cycle

Any write to the ISR or ICR (by the **MVC** instruction) effectively has one delay slot because the results cannot be read (by the **MVC** instruction) in the IFR until two cycles after the write to the ISR or ICR.

Delay Slots

0

Example

```
MVC .S2    B1,AMR
```

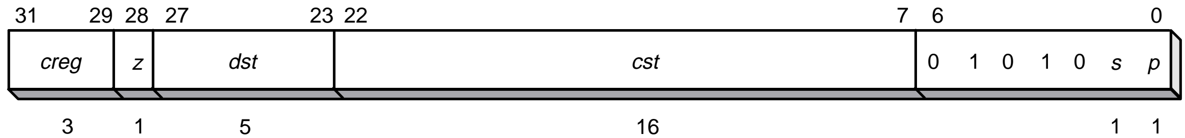
Note:

The six MSBs of the AMR register are reserved and therefore are not written to.

Syntax **MVK** (.unit) *cst*, *dst*

.unit = .S1 or .S2

Opcode



Opcode map field used...	For operand type...	Unit
cst *dst*	*scst*16 sint	.S1, .S2

Description The 16-bit constant is sign extended and placed in *dst*.

Execution

if (cond) *scst*16 → *dst*
else nop

Instruction Type Single-cycle

Delay Slots 0

Note:

To load 32-bit constants, such as 0x12345678, use the following pair of instructions:

```
MVK    0x5678
MVKH   0x1234
```

You could also use:

```
MVK    0x12345678
MVKH   0x12345678
```

If you are loading the address of a label, use:

```
MVK    label
MVKH   label
```

Example 1

```
MVK .S1    293,A1
```

	Before instruction		1 cycle after instruction	
A1	XXXX XXXXh	A1	0000 0125h	293

Example 2

```
MVK .S2    125h,B1
```

	Before instruction		1 cycle after instruction	
B1	XXXX XXXXh	B1	0000 0125h	293

Example 3

```
MVK .S1    0FF12h,A1
```

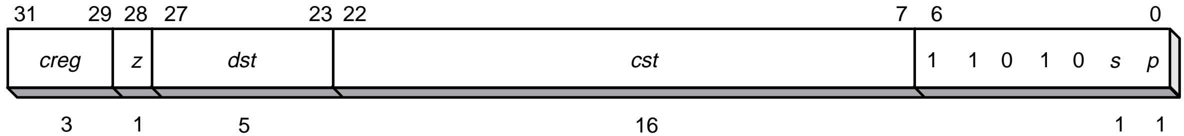
	Before instruction		1 cycle after instruction	
A1	XXXX XXXXh	A1	FFFF FF12h	−238

Syntax

MVKH (.unit) *cst*, *dst*
or
MVKLH (.unit) *cst*, *dst*

.unit = .S1 or .S2

Opcode



Opcode map field used...	For operand type...	Unit
cst	u*scst*16	.S1, .S2
dst	sint	

Description

The 16-bit constant *cst* is loaded into the upper 16 bits of *dst*. The 16 LSBs of *dst* are unchanged. The assembler encodes the 16 MSBs of a 32-bit constant into the *cst* field of the opcode for the **MVKH** instruction. The assembler encodes the 16 LSBs of a constant into the *cst* field of the opcode for the **MVKLH** instruction.

Execution

`MVKLH` if (cond)(($cst_{15..0}$) << 16) or ($dst_{15..0}$) $\rightarrow$ *dst*
else nop

`MVKH` if (cond)(($cst_{31..16}$) << 16) or ($dst_{15..0}$) $\rightarrow$ *dst*
else nop

Instruction Type Single-cycle

Delay Slots 0

Note:

To load 32-bit constants, such as 0x12345678, use the following pair of instructions:

```
MVK    0x5678
MVKH   0x1234
```

You could also use:

```
MVK    0x12345678
MVKH   0x12345678
```

If you are loading the address of a label, use:

```
MVK    label
MVKH   label
```

Example 1

```
MVKH .S1   0A329123h,A1
```

	Before instruction	1 cycle after instruction
A1	0000 7634h	0A32 7634h

Example 2

```
MVKLH .S1  7A8h,A1
```

	Before instruction	1 cycle after instruction
A1	FFFF F25Ah	07A8 F25Ah

Syntax **NEG** (.unit) *src, dst*

.unit = .L1, .L2, .S1, .S2

Description This is a pseudo operation used to negate *src* and place in *dst*. The assembler uses the operation **SUB** 0, *src, dst* to perform this task.

Execution if (cond) 0 –*s src* → *dst*
else nop

Instruction Type Single-cycle

Delay Slots 0

Syntax **NOP** [*count*]

Opcode

31–18	17	16–13	12–1	0
reserved	0	*src*	0 0 0 0 0 0 0 0 0 0 0 0	*p*
14		4		1

Opcode map field used...	For operand type...	Unit
src	*ucst*4	none

Description *src* is encoded as *count*–1. For *src* + 1 cycles, no operation is performed. The maximum value for *count* is 9. **NOP** with no operand is treated like **NOP** 1 with *src* encoded as 0000.

A multicycle **NOP** will not finish if a branch is completed first. For example, if a branch is initiated on cycle n and a **NOP** 5 instruction is initiated on cycle n + 3, the branch is complete on cycle n + 6 and the **NOP** is executed only from cycle n + 3 to cycle n + 5. A single-cycle **NOP** in parallel with other instructions does not affect operation.

Execution No operation for *count* cycles

Instruction Type **NOP**

Delay Slots 0

Example 1

```
NOP
MVK .S1            125h,A1
```

	Before NOP	1 cycle after NOP (No operation executes)	1 cycle after MVK
A1	1234 5678h	1234 5678h	0000 0125h

Example 2

```
MVK    .S1   1,A1
MVKLH  .S1   0,A1
NOP    5
ADD    .L1   A1,A2,A1
```

	Before NOP 5	1 cycle after ADD instruction (6 cycles after NOP 5)
A1	0000 0001h	0000 0004h
A2	0000 0003h	0000 0003h

Syntax

NORM (.unit) *src2*, *dst*

.unit = .L1 or .L2

Opcode map field used...	For operand type...	Unit	Opfield
src2 *dst*	xsint uint	.L1, .L2	1100011
src2 *dst*	slong uint	.L1, .L2	1100000

Description

The number of bits of the first nonredundant sign bit from the MSB of the *src2* operand is placed in *dst*. Several examples are shown in the following diagram.

In this case, **NORM** returns 0:

src2

0	1	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

In this case, **NORM** returns 3:

src2

0	0	0	0	1	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

In this case, **NORM** returns 30:

src2

1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

In this case, **NORM** returns 31:

src2

1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Execution

if (cond) norm(*src*) → *dst*
else nop

Instruction Type Single-cycle

Delay Slots 0

Example 1

```
NORM .L1   A1,A2
```

	Before instruction		1 cycle after instruction	
A1	02A3 469Fh	A1	02A3 469Fh	
A2	XXXX XXXXh	A2	0000 0005h	5

Example 2

```
NORM .L1   A1,A2
```

	Before instruction		1 cycle after instruction	
A1	FFFF F25Ah	A1	FFFF F25Ah	
A2	XXXX XXXXh	A2	0000 0013h	19

Syntax **NOT** (.unit) *src, dst*

(.unit) = .L1, .L2, .S1, or .S2

Description This is a pseudo operation used to bitwise **NOT** the *src* operand and place the result in *dst*. The assembler uses the operation **XOR** (.unit) –1, *src*, *dst* to perform this task.

Execution if (cond) –1 xor *src* → *dst*
else nop

Instruction Type Single-cycle

Delay Slots 0

Syntax

OR (.unit) *src1*, *src2*, *dst*

.unit = .L1, .L2, .S1, .S2

Opcode map field used...	For operand type...	Unit	Opfield
src1 *src2* *dst*	uint xuint uint	.L1, .L2	1111111
src1 *src2* *dst*	*scst5* xuint uint	.L1, .L2	1111110
src1 *src2* *dst*	uint xuint uint	.S1, .S2	011011
src1 *src2* *dst*	*scst5* xuint uint	.S1, .S2	011010

Description

A bitwise OR is performed between *src1* and *src2*. The result is placed in *dst*. The *scst5* operands are sign extended to 32 bits.

Execution

if (cond) *src1* or *src2* → *dst*
else nop

Instruction Type

Single-cycle

Delay Slots

0

Example 1

```
OR .L1    A1,B1,A2
```

	Before instruction	1 cycle after instruction
A1	08A3 A49Fh	08A3 A49Fh
A2	XXXX XXXXh	08FF B7DFh
B1	00FF 375Ah	00FF 375Ah

Example 2

```
OR .L2    -12,B1,B2
```

	Before instruction	1 cycle after instruction
B1	0000 3A41h	0000 3A41h
B2	XXXX XXXXh	FFFF FFF5h

Syntax

SADD (.unit) *src1*, *src2*, *dst*

.unit = .L1 or .L2

Opcode map field used...	For operand type...	Unit	Opfield
src1 *src2* *dst*	sint xsint sint	.L1, .L2	0010011
src1 *src2* *dst*	xsint slong slong	.L1, .L2	0110001
src1 *src2* *dst*	*scst*5 xsint sint	.L1, .L2	0010010
src1 *src2* *dst*	*scst*5 slong slong	.L1, .L2	0110000

Description

src1 is added to *src2* and saturated if an overflow occurs according to the following rules:

1) If the *dst* is an int and $src1 + src2 > 2^{31} - 1$, then the result is $2^{31} - 1$.
2) If the *dst* is an int and $src1 + src2 < -2^{31}$, then the result is -2^{31}.
3) If the *dst* is a long and $src1 + src2 > 2^{39} - 1$, then the result is $2^{39} - 1$.
4) If the *dst* is a long and $src1 + src2 < -2^{39}$, then the result is -2^{39}.

The result is placed in *dst*. If a saturate occurs, the SAT bit in the control status register (CSR) is set one cycle after *dst* is written.

Execution

if (cond) *src1 +s src2* → *dst*
else nop

Instruction Type

Single-cycle

Delay Slots

0

Example 1 `SADD .L1 A1,A2,A3`

	Before instruction			1 cycle after instruction			2 cycles after instruction	
A1	5A2E 51A3h	1512984995	A1	5A2E 51A3h		A1	5A2E 51A3h	
A2	012A 3FA2h	19546018	A2	012A 3FA2h		A2	012A 3FA2h	
A3	XXXX XXXXh		A3	5B58 9145h	1532531013	A3	5B58 9145h	
CSR	0001 0100h		CSR	0001 0100h		CSR	0001 0100h	Not saturated

Example 2 `SADD .L1 A1,A2,A3`

	Before instruction			1 cycle after instruction			2 cycles after instruction	
A1	4367 71F2h	1130852850	A1	4367 71F2h		A1	4367 71F2h	
A2	5A2E 51A3h	1512984995	A2	5A2E 51A3h		A2	5A2E 51A3h	
A3	XXXX XXXXh		A3	7FFF FFFFh	2147483647	A3	7FFF FFFFh	
CSR	0001 0100h		CSR	0001 0100h		CSR	0001 0300h	Saturated

Example 3 `SADD .L1 B2,A5:A4,A7:A6`

	Before instruction				1 cycle after instruction		
A5:A4	0000 0000h	7C83 39B1h	†1922644401	A5:A4	0000 0000h	7C83 39B1h	
A7:A6	XXXX XXXXh	XXXX XXXXh		A7:A6	0000 0000h	83C3 7953h	†2210625875
B2	112A 3FA2h	287981474		B2	112A 3FA2h		
CSR	0001 0100h			CSR	0001 0100h	CSR	

	2 cycles after instruction	
A5:A4	0000 0000h	7C83 39B1h
A7:A6	0000 0000h	83C3 7953h
B2	112A 3FA2h	
CSR	0001 0100h	Not saturated

† Signed long integer (40-bit)

Syntax

SAT (.unit) *src2*, *dst*

.unit = .L1 or .L2

Opcode map field used...	For operand type...	Unit	Opfield
src2 *dst*	slong sint	.L1, .L2	1000000

Description

A 40-bit *src2* value is converted to a 32-bit value. If the value in *src2* is greater than what can be represented in 32-bits, *src2* is saturated. The result is placed in *dst*. If a saturate occurs, the SAT bit in the control status register (CSR) is set one cycle after *dst* is written.

Execution

if (cond) {
  if ($src2 > (2^{31} - 1)$)
    $(2^{31} - 1) \rightarrow dst$
  else if ($src2 < -2^{31}$)
    $-2^{31} \rightarrow dst$
  else $src2_{31..0} \rightarrow dst$
  }
else nop

Instruction Type

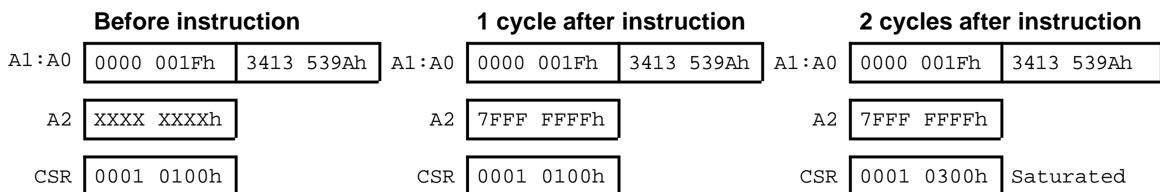
Single-cycle

Delay Slots

0

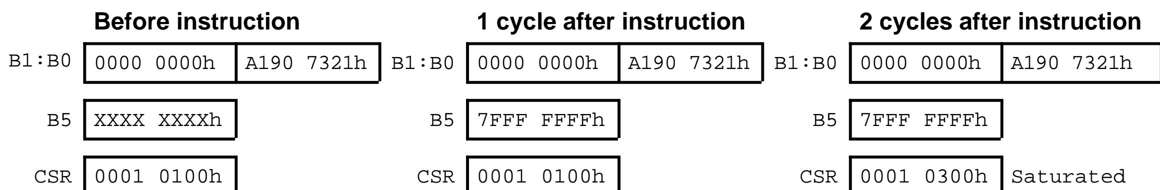
Example 1

```
SAT .L1    A1:A0,A2
```



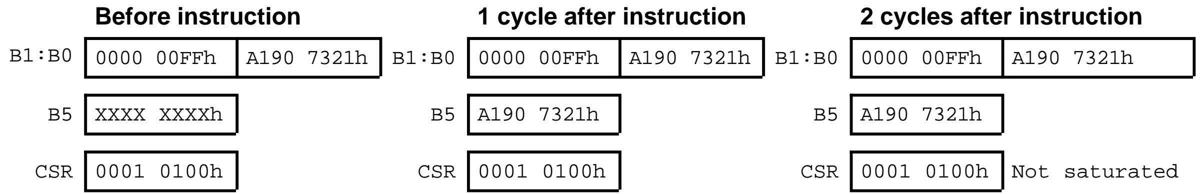
Example 2

```
SAT .L2    B1:B0,B5
```



Example 3

```
SAT .L2    B1:B0,B5
```



Syntax

SET (.unit) *src2*, *csta*, *cstb*, *dst*
or
SET (.unit) *src2*, *src1*, *dst*

.unit = .S1 or .S2

Opcode

Constant form:

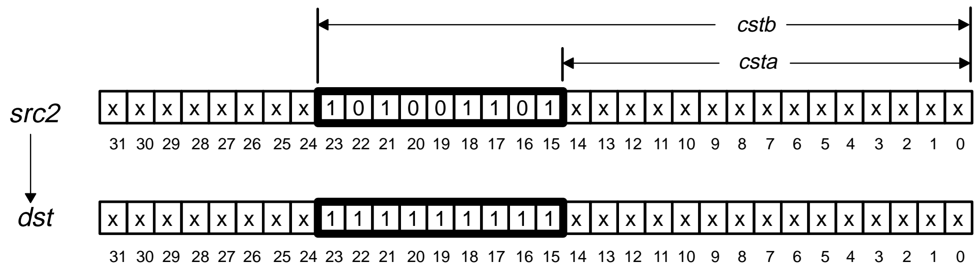
31–29	28	27–23	22–18	17–13	12–8	7–6	5–2	1	0
creg	*z*	*dst*	*src2*	*csta*	*cstb*	*op*	0 0 1 0	*s*	*p*
3	1	5	5	5	5	2		1	1

Register form:

31–29	28	27–23	22–18	17–13	12	11–6	5–2	1	0
creg	*z*	*dst*	*src2*	*src1*	*x*	*op*	1 0 0 0	*s*	*p*
3	1	5	5	5		6		1	1

Opcode map field used...	For operand type...	Unit	Opfield
src2 *csta* *cstb* *dst*	uint *ucst*5 *ucst*5 uint	.S1, .S2	10
src2 *src1* *dst*	xuint uint uint	.S1, .S2	111011

Description The field in *src2*, specified by *csta* and *cstb*, is set to all 1s. The *csta* and *cstb* operands may be specified as constants or in the ten LSBs of the *src1* register, with *cstb* being bits 0–4 and *csta* bits 5–9. *csta* signifies the bit location of the LSB of the field and *cstb* signifies the bit location of the MSB of the field. In other words, *csta* and *cstb* represent the beginning and ending bits, respectively, of the field to be set to all 1s. The LSB location of *src2* is 0 and the MSB location of *src2* is 31. In the example below, *csta* is 15 and *cstb* is 23.



Execution If the constant form is used:

if (cond) *src2* set *csta*, *cstb* $\rightarrow$ *dst*
else nop

If the register form is used:

if (cond) *src2* set $src1_{9..5}$, $src1_{4..0}$ $\rightarrow$ *dst*
else nop

Instruction Type Single-cycle

Delay Slots 0

Example 1

```
SET .S1    A0,7,21,A1
```

	Before instruction		1 cycle after instruction
A0	4B13 4A1Eh	A0	4B13 4A1Eh
A1	XXXX XXXXh	A1	4B3F FF9Eh

Example 2

```
SET .S1    B0,B1,B2
```

	Before instruction		1 cycle after instruction
B0	9ED3 1A31h	B0	9ED3 1A31h
B1	0000 C197h	B1	0000 C197h
B2	XXXX XXXXh	B2	9EFF FA31h

Syntax

SHL (.unit) *src2*, *src1*, *dst*

.unit = .S1 or .S2

Opcode map field used...	For operand type...	Unit	Opfield
src2 *src1* *dst*	xsint uint sint	.S1, .S2	110011
src2 *src1* *dst*	slong uint slong	.S1, .S2	110001
src2 *src1* *dst*	xsint uint slong	.S1, .S2	010011
src2 *src1* *dst*	xsint *ucst*5 sint	.S1, .S2	110010
src2 *src1* *dst*	slong *ucst*5 slong	.S1, .S2	110000
src2 *src1* *dst*	xsint *ucst*5 slong	.S1, .S2	010010

Description

The *src2* operand is shifted to the left by the *src1* operand. The result is placed in *dst*. When a register is used, the six LSBs specify the shift amount and valid values are 0–40. When an immediate is used, valid shift amounts are 0–31.

If 39 < *src1* < 64, *src2* is shifted to the left by 40. Only the six LSBs are used by the shifter so any bits set above bit 5 do not affect execution.

Execution

if (cond) *src2* << *src1* → *dst*
else nop

Instruction Type

Single-cycle

Delay Slots

0

Example 1

```
SHL .S1    A0,4,A1
```

	Before instruction		1 cycle after instruction
A0	29E3 D31Ch	A0	29E3 D31Ch
A1	XXXX XXXXh	A1	9E3D 31C0h

Example 2

```
SHL .S2    B0,B1,B2
```

	Before instruction		1 cycle after instruction
B0	4197 51A5h	B0	4197 51A5h
B1	0000 0009h	B1	0000 0009h
B2	XXXX XXXXh	B2	2EA3 4A00h

Example 3

```
SHL .S2    B1:B0,B2,B3:B2
```

	Before instruction			1 cycle after instruction	
B1:B0	0000 0009h	4197 51A5h	B1:B0	0000 0009h	4197 51A5h
B2	0000 0022h		B2	0000 0000h	
B3:B2	XXXX XXXXh	XXXX XXXXh	B3:B2	0000 0094h	0000 0000h

Syntax

SHR (.unit) *src2*, *src1*, *dst*

.unit = .S1 or .S2

Opcode map field used...	For operand type...	Unit	Opfield
src2 *src1* *dst*	xsint uint sint	.S1, .S2	110111
src2 *src1* *dst*	slong uint slong	.S1, .S2	110101
src2 *src1* *dst*	xsint *ucst*5 sint	.S1, .S2	110110
src2 *src1* *dst*	slong *ucst*5 slong	.S1, .S2	110100

Description

The *src2* operand is shifted to the right by the *src1* operand. The sign-extended result is placed in *dst*. When a register is used, the six LSBs specify the shift amount and valid values are 0–40. When an immediate is used, valid shift amounts are 0–31.

If 39 < *src1* < 64, *src2* is shifted to the right by 40. Only the six LSBs are used by the shifter so any bits set above bit 5 do not affect execution.

Execution

if (cond) *src2* >>s *src1* → *dst*
else nop

Instruction Type

Single-cycle

Delay Slots

0

Example 1

```
SHR .S1    A0,8,A1
```

Example 2

```
SHR .S2    B0,B1,B2
```

	Before instruction	1 cycle after instruction
B0	1492 5A41h	1492 5A41h
B1	0000 0012h	0000 0012h
B2	XXXX XXXXh	0000 0524h

Example 3

```
SHR .S2    B1:B0,B2,B3:B2
```

	Before instruction		1 cycle after instruction	
B1:B0	0000 0012h	1492 5A41h	0000 0012h	1492 5A41h
B2	0000 0019h		0000 090Ah	
B3:B2	XXXX XXXXh	XXXX XXXXh	0000 0000h	0000 090Ah

Syntax

SHRU (.unit) *src2*, *src1*, *dst*

.unit = .S1 or .S2

Opcode map field used...	For operand type...	Unit	Opfield
src2 *src1* *dst*	xuint uint uint	.S1, .S2	100111
src2 *src1* *dst*	ulong uint ulong	.S1, .S2	100101
src2 *src1* *dst*	xuint *ucst*5 uint	.S1, .S2	100110
src2 *src1* *dst*	ulong *ucst*5 ulong	.S1, .S2	100100

Description

The *src2* operand is shifted to the right by the *src1* operand. The zero-extended result is placed in *dst*. When a register is used, the six LSBs specify the shift amount and valid values are 0–40. When an immediate is used, valid shift amounts are 0–31.

If 39 < *src1* < 64, *src2* is shifted to the right by 40. Only the six LSBs are used by the shifter, so any bits set above bit 5 do not affect execution.

Execution

if (cond) *src2* >>z *src1* → *dst*
else nop

Instruction Type

Single-cycle

Delay Slots

0

Example

```
SHRU .S1   A0,8,A1
```

Syntax

SMPY(L)(H) (.unit) *src1*, *src2*, *dst*

.unit = .M1 or .M2

Opcode map field used...	For operand type...	Unit	Opfield	Mnemonic
src1 *src2* *dst*	slsb16 xslsb15 sint	.M1, .M2	11010	SMPY
src1 *src2* *dst*	smsb16 xslsb16 sint	.M1, .M2	01010	SMPYHL
src1 *src2* *dst*	slsb16 xsmsb16 sint	.M1, .M2	10010	SMPYLH
src1 *src2* *dst*	smsb16 xsmsb16 sint	.M1, .M2	00010	SMPYH

Description

The *src1* operand is multiplied by the *src2* operand. The result is left shifted by 1 and placed in *dst*. If the left-shifted result is 0x8000 0000, then the result is saturated to 0x7FFF FFFF. If a saturate occurs, the SAT bit in the CSR is set one cycle after *dst* is written.

Execution

```
if (cond)    {
             if (((src1 × src2) << 1) × != 0x8000 0000 )
                  ((src1 × src2) << 1) → dst
             else
                  0x7FFF FFFF → dst
             }
else         nop
```

Instruction Type

Single-cycle

Delay Slots

1

Example 1

```
SMPY .M1   A1,A2,A3
```

	Before instruction			1 cycle after instruction	
A1	0000 0123h	‡291	A1	0000 0123h	
A2	01E0 FA81h	‡–1407	A2	01E0 FA81h	
A3	XXXX XXXXh		A3	FFF3 8146h	–818874
CSR	0001 0100h		CSR	0001 0100h	Not saturated

‡ Signed 16 LSBs

Example 2

```
SMPYHL .M1 A1,A2,A3
```

	Before instruction			2 cycles after instruction	
A1	008A 0000h	†138	A1	008A 0000h	
A2	0000 00A7h	‡167	A2	0000 00A7h	
A3	XXXX XXXXh		A3	0000 B40Ch	46092
CSR	0001 0100h		CSR	0001 0100h	Not saturated

Example 3

```
SMPYLH .M1 A1,A2,A3
```

	Before instruction			2 cycles after instruction	
A1	0000 8000h	‡−32768	A1	0000 8000h	
A2	8000 0000h	†−32768	A2	8000 0000h	
A3	XXXX XXXXh		A3	7FFF FFFFh	2147483647
CSR	0001 0100h		CSR	0001 0300h	Saturated

† Signed integer, 16 MSBs
‡ Signed integer, 16 LSBs

Syntax **SSHL** (.unit) *src2*, *src1*, *dst*

.unit = .S1 or .S2

Opcode map field used...	For operand type...	Unit	Opfield
src2 *src1* *dst*	xsint uint sint	.S1, .S2	100011
src2 *src1* *dst*	xsint *ucst*5 sint	.S1, .S2	100010

Description The *src2* operand is shifted to the left by the *src1* operand. The result is placed in *dst*. When a register is used, the five least significant bits specify the shift amount and valid values are 0 through 31. The result of the shift is saturated to 32 bits. If a saturate occurs, the SAT bit in the CSR is set one cycle after *dst* is written.

Execution

```
if (cond) {
            if ( bit(31) through bit(31–src1) of src2 are all 1s or all 0s)
                dst = src2 << src1;
            else if (src2 > 0)
                saturate dst to 0x7FFFFFFF;
            else if (src2 < 0)
              saturate dst to 0x80000000;
            }
else        nop
```

Instruction Type Single-cycle

Delay Slots 0

Example 1

```
SSHL .S1   A0,2,A1
```

	Before instruction	1 cycle after instruction	2 cycles after instruction	
A0	02E3 031Ch	02E3 031Ch	02E3 031Ch	
A1	XXXX XXXXh	0B8C 0C70h	0B8C 0C70h	
CSR	0001 0100h	0001 0100h	0001 0100h	Not saturated

Example 2

```
SSHL .S1   A0,A1,A2
```

	Before instruction	1 cycle after instruction	2 cycles after instruction	
A0	4719 1925h	4719 1925h	4719 1925h	
A1	0000 0006h	0000 0006h	0000 0006h	
A2	XXXX XXXXh	7FFF FFFFh	7FFF FFFFh	
CSR	0001 0100h	0001 0100h	0001 0300h	Saturated

Syntax

SSUB (.unit) *src1*, *src2*, *dst*

.unit = .L1 or .L2

Opcode map field used...	For operand type...	Unit	Opfield
src1 *src2* *dst*	sint xsint sint	.L1, .L2	0001111
src1 *src2* *dst*	xsint sint sint	.L1, .L2	0011111
src1 *src2* *dst*	*scst*5 xsint sint	.L1, .L2	0001110
src1 *src2* *dst*	*scst*5 slong slong	.L1, .L2	0101100

Description

src2 is subtracted from *src1* and is saturated to the result size according to the following rules:

1) If the result is an int and *src1* – *src2* > 2^{31} – 1, then the result is 2^{31} – 1.
2) If the result is an int and *src1* – *src2* < –2^{31}, then the result is –2^{31}.
3) If the result is a long and *src1* – *src2* > 2^{39} – 1, then the result is 2^{39} – 1.
4) If the result is a long and *src1* – *src2* < –2^{39}, then the result is –2^{39}.

The result is placed in *dst*. If a saturate occurs, the SAT bit in the CSR is set one cycle after *dst* is written.

Execution

if (cond) *src1* –s *src2* → *dst*
else nop

Instruction Type Single-cycle

Delay Slots 0

Example 1

```
SSUB .L2   B1,B2,B3
```

	Before instruction			1 cycle after instruction			2 cycles after instruction	
B1	5A2E 51A3h	1512984995	B1	5A2E 51A3h		B1	5A2E 51A3h	
B2	802A 3FA2h	−2144714846	B2	802A 3FA2h		B2	802A 3FA2h	
B3	XXXX XXXXh		B3	7FFF FFFFh	2147483647	B3	7FFF FFFFh	
CSR	0001 0100h		CSR	0001 0100h		CSR	0001 0300h	Saturated

Example 2

```
SSUB .L2   A0,A1,A2
```

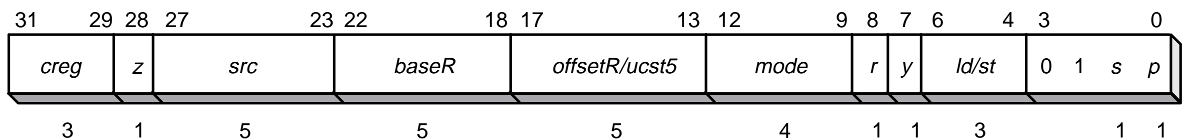
	Before instruction			1 cycle after instruction			2 cycles after instruction	
A0	4367 71F2h	1130852850	A0	4367 71F2h		A0	4367 71F2h	
A1	5A2E 51A3h	1512984995	A1	5A2E 51A3h		A1	5A2E 51A3h	
A2	XXXX XXXXh		A2	E939 204Fh	−382132145	A2	E939 204Fh	
CSR	0001 0100h		CSR	0001 0100h		CSR	0001 0100h	Not saturated

Syntax

Register Offset	Unsigned Constant Offset
STB (.unit) *src*,*+*baseR[offsetR]*,	**STBU** (.unit) *src*, *+*baseR[ucst5]*,
or	or
STH (.unit) *src*, *+*baseR[offsetR]*,	**STHU** (.unit) *src*, *+*baseR[ucst5]*,
or	or
STW (.unit) *src*, *+*baseR[offsetR]*,	**STW** (.unit) *src*, *+*baseR[ucst5]*,

.unit = .D1 or .D2

Opcode



Description

Each of these instructions performs a store to memory from a general-purpose register (*src*). Table 3–16 summarizes the data types supported by stores. Table 3–17 describes the addressing generator options. The memory address is formed from a base address register (*baseR*) and an optional offset that is either a register (*offsetR*) or a 5-bit unsigned constant (*ucst5*).

offsetR and *baseR* must be in the same register file and on the same side as the .D unit used. The *y* bit in the opcode determines the .D unit and register file used: *y* = 0 selects the .D1 unit and *baseR* and *offsetR* from the A register file, and *y* = 1 selects the .D2 unit and *baseR* and *offsetR* from the B register file.

offsetR/ucst5 is scaled by a left-shift of 0, 1, or 2 for **STB**, **STH**, and **STW**, respectively. After scaling, *offsetR*/*ucst*5 is added to or subtracted from *baseR*. For the preincrement, predecrement, positive offset, and negative offset address generator options, the result of the calculation is the address to be accessed in memory. For postincrement or postdecrement addressing, the value of *baseR* before the addition or subtraction is sent to memory.

The addressing arithmetic that performs the additions and subtractions defaults to linear mode. However, for A4–A7 and for B4–B7, the mode can be changed to circular mode by writing the appropriate value to the AMR (see section 2.3).

For **STB** and **STH** the 8 and 16 LSBs of the *src* register are stored. For **STW** the entire 32-bit value is stored. *src* can be in either register file, regardless of the .D unit or *baseR* or *offsetR* used. The *s* bit determines which file the source is read from: *s* = 0 indicates *src* will be in the A register file, and *s* = 1 indicates *src* will be in the B register file.

Table 3–16. Data Types Supported by Stores

Mnemonic	ld/st Field	Store Data Type	SIze	Left Shift of Offset
STB	0 1 1	Store byte	8	0 bits
STH	1 0 1	Store halfword	16	1 bit
STW	1 1 1	Store word	32	2 bits

Table 3–17. Address Generator Options

Mode Field	Syntax	Modification Performed
0 1 0 1	*+R[*offsetR*]	Positive offset
0 1 0 0	*–R[*offsetR*]	Negative offset
1 1 0 1	*++R[*offsetR*]	Preincrement
1 1 0 0	*––R[*offsetR*]	Predecrement
1 1 1 1	*R++[*offsetR*]	Postincrement
1 1 1 0	*R– –[*offsetR*]	Postdecrement
0 0 0 1	*+R[*ucst5*]	Positive offset
0 0 0 0	*–R[*ucst5*]	Negative offset
1 0 0 1	*++R[*ucst5*]	Preincrement
1 0 0 0	*– –R[*ucst5*]	Predecrement
1 0 1 1	*R++[*ucst5*]	Postincrement
1 0 1 0	*R– –[*ucst5*]	Postdecrement

Increments and decrements default to 1 and offsets default to zero when no bracketed register or constant is specified. Stores that do no modification to the *baseR* can use the syntax *R. Square brackets, [ ], indicate that the *ucst*5 offset is left-shifted by 2, 1, or 0 for word, halfword, and byte loads, respectively. Parentheses, ( ), can be used to set a nonscaled, constant offset. For example, **STW** (.unit) *+*baseR*(12) *dst* represents an offset of 12 bytes whereas **STW** (.unit) *+*baseR*[12] *dst* represents an offset of 12 words, or 48 bytes.

Execution

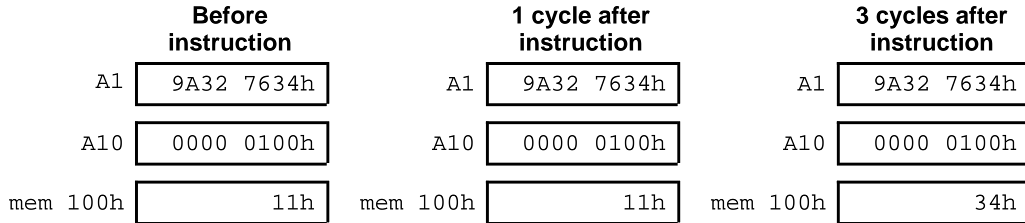
if (cond) *src* → mem
else nop

Instruction Type Store

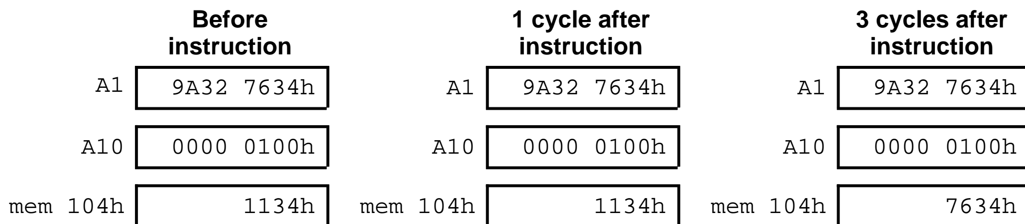
Delay Slots 0

For more information on delay slots for a store, see Chapter 4, *TMS320C62xx Pipeline*.

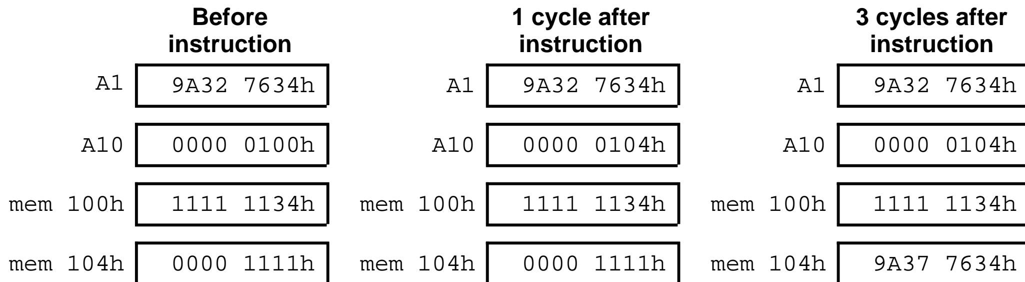
Example 1 `STB .D1 A1,*A10`



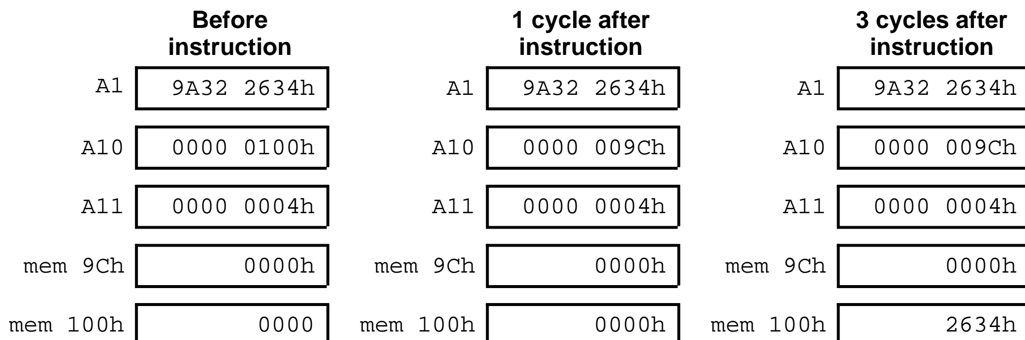
Example 2 `STH .D1 A1,*+A10(4)`



Example 3 `STW .D1 A1,*++A10[1]`



Example 4 `STH .D1 A1,*A10--[A11]`

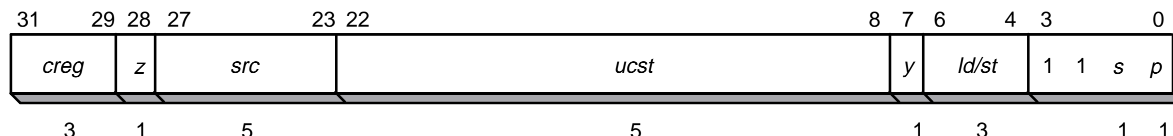


Syntax

STB (.unit) *src*, *+B14/B15[*ucst15*]
or
STH (.unit) *src*, *+B14/B15[*ucst15*]
or
STW (.unit) *src*, *+B14/B15[*ucst15*]

.unit = .D2

Opcode



Description

These instructions perform stores to memory from a general-purpose register (*src*). Table 3–18 summarizes the data types supported by stores. The memory address is formed from a base address register B14 (*y* = 0) or B15 (*y* = 1) and an optional offset that is a 15-bit unsigned constant (*ucst15*). The assembler selects this format only when the constant is larger than five bits in magnitude. This instruction executes only on the .D2 unit.

The offset, *ucst15*, is scaled by a left-shift of 0, 1, or 2 for **STB**, **STH**, and **STW**, respectively. After scaling, *ucst15* is added to or subtracted from *baseR*. The result of the calculation is the address that is sent to memory. The addressing arithmetic is always performed in linear mode.

For **STB** and **STH** the 8 and 16 LSBs of the *src* register are stored. For **STW** the entire 32-bit value is stored. *src* can be in either register file. The *s* bit determines which file the source is read from: *s* = 0 indicates *src* is in the A register file, and *s* = 1 indicates *src* is in the B register file.

Square brackets, [ ], indicate that the *ucst15* offset is left-shifted by 2, 1, or 0 for word, halfword, and byte loads, respectively. Parentheses, ( ), can be used to set a nonscaled, constant offset. For example, **STW** (.unit) *+B14/B15(60) *dst* represents an offset of 12 bytes, whereas **STW** (.unit) *+B14/B15[60] *dst* represents an offset of 60 words, or 240 bytes.

Table 3–18. Data Types Supported by Stores

Mnemonic	*ld/st* Field	Store Data Type	SIze	Left Shift of Offset
STB	0 1 1	Store byte	8	0 bits
STH	1 0 1	Store halfword	16	1 bit
STW	1 1 1	Store word	32	2 bits

Execution

if (cond) *src* → mem
else nop

Instruction Type Store

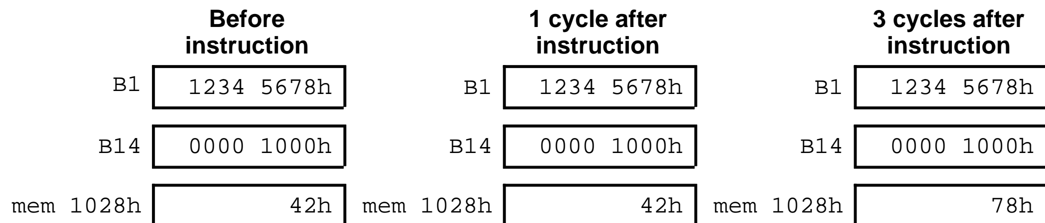
Delay Slots 0

Note:

This instruction executes only on the .D2 unit.

Example 1

```
STB .D2    B1,*+B14[40]
```



Syntax

SUB (.unit) *src1*, *src2*, *dst*
or
SUBU (.unit) *src1*, *src2*, *dst*
or
SUB (.D1 or .D2) *src2*, *src1*, *dst*

.unit = .L1, .L2, .S1, .S2

Opcode map field used...	For operand type...	Unit	Opfield	Mnemonic
src1 *src2* *dst*	sint xsint sint	.L1, .L2	0000111	SUB
src1 *src2* *dst*	xsint sint sint	.L1, .L2	0010111	SUB
src1 *src2* *dst*	sint xsint slong	.L1, .L2	0100111	SUB
src1 *src2* *dst*	xsint sint slong	.L1, .L2	0110111	SUB
src1 *src2* *dst*	uint xuint ulong	.L1, .L2	0101111	SUBU
src1 *src2* *dst*	xuint uint ulong	.L1, .L2	0111111	SUBU
src1 *src2* *dst*	*scst*5 xsint sint	.L1, .L2	0000110	SUB
src1 *src2* *dst*	*scst*5 slong slong	.L1, .L2	010010 0	SUB
src1 *src2* *dst*	sint xsint sint	.S1, .S2	010111	SUB
src1 *src2* *dst*	*scst*5 xsint sint	.S1, .S2	010110	SUB

Opcode map field used...	For operand type...	Unit	Opfield	Mnemonic
src2 *src1* *dst*	sint sint sint	.D1, .D2	010001	SUB
src2 *src1* *dst*	sint *ucst5* sint	.D1, .D2	010011	SUB

Description for .L1, .L2 and .S1, .S2 opcodes

src2 is subtracted from *src1*. The result is placed in *dst*.

Execution for .L1, .L2 and .S1, .S2 opcodes

if (cond) *src1* – *src2* → *dst*
else nop

Description for .D1, .D2 opcodes

src1 is subtracted from *src2*. The result is placed in *dst*.

Execution for .D1, .D2 opcodes

if (cond) *src2* – *src1* → *dst*
else nop

Note:

Subtraction with a signed constant on the .L and .S units allows either the first or the second operand to be the signed 5-bit constant.

SUB scst5, *src2*, *dst*
or
SUB *src1*, scst5, *dst* (Encoded as ADD –scst5, *src2*, *dst* where the *src1* register is now *src2* and scst5 is now –scst5)

However, the .D unit provides only the second operand as a constant since it is an unsigned 5-bit constant. *ucst5* allows a greater offset for addressing with the .D unit.

Instruction Type Single-cycle

Delay Slots 0

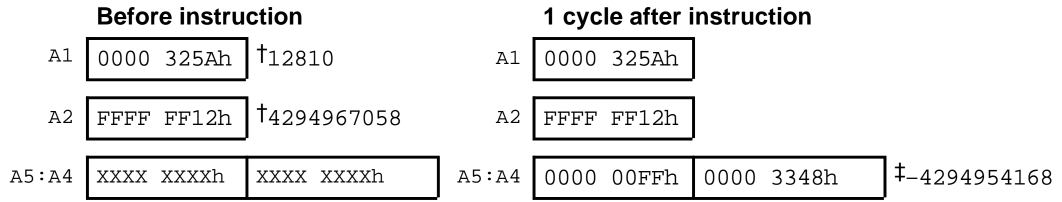
Example 1

```
SUB .L1    A1,A2,A3
```

	Before instruction			1 cycle after instruction	
A1	0000 325Ah	12810	A1	0000 325Ah	
A2	FFFF FF12h	−238	A2	FFFF FF12h	
A3	XXXX XXXXh		A3	0000 3348h	13128

Example 2

```
SUBU .L1   A1,A2,A5:A4
```



† Unsigned 32-bit integer
‡ Signed 40-bit (long) integer

Syntax

SUBAB (.unit) *src2*, *src1*, *dst*
or
SUBAH (.unit) *src2*, *src1*, *dst*
or
SUBAW (.unit) *src2*, *src1*, *dst*

.unit = .D1 or .D2

Opcode map field used...	For operand type...	Unit	Opfield
src2 *src1* *dst*	sint sint sint	.D1, .D2	Byte: 110001 Halfword: 110101 Word: 111001
src2 *src1* *dst*	sint *ucst*5 sint	.D1, .D2	Byte: 110011 Halfword: 110111 Word: 111011

Description

src1 is subtracted from *src2*. The subtraction defaults to linear mode. However, if *src2* is one of A4–A7 or B4–B7, the mode can be changed to circular mode by writing the appropriate value to the AMR (see section 2.3). *src1* is left shifted by 1 or 2 for halfword and word data sizes, respectively. **SUBAB**, **SUBAH**, and **SUBAW** are byte, halfword, and word mnemonics, respectively. The result is placed in *dst*.

Execution

if (cond) *src2* –a *src1* → *dst*
else nop

Instruction Type Single-cycle

Delay Slots 0

Example 1

```
SUBAB .D1  A5,A0,A5
```

	Before instruction		1 cycle after instruction
A0	0000 0004h	A0	0000 0004h
A5	0000 4000h	A5	0000 400Ch
AMR	0003 0004h	AMR	0003 0004h

```
BK0 = 3 → size = 16
A5 in circular addressing mode using BK0
```

Example 2

```
SUBAW .D1  A5,2,A3
```

	Before instruction		1 cycle after instruction
A3	XXXX XXXXh	A3	0000 0108h
A5	0000 0100h	A5	0000 0100h
AMR	0003 0004h	AMR	0003 0004h

```
BK0 = 3 → size = 16
A5 in circular addressing mode using BK0
```

Syntax

SUBC (.unit) *src1*, *src2*, *dst*

.unit = .L1 or .L2

Opcode map field used...	For operand type...	Unit	Opfield
src1 *src2* *dst*	uint xuint uint	.L1, .L2	1001011

Description

Subtract *src2* from *src1*. If result is greater than or equal to zero, left shift result and add 1 to it. Place the result in *dst*. This step is commonly used in division.

Execution

if (cond) {
  if (*src1* – *src2* $\geq$ 0)
    ((*src1*–*src2*) << 1) + 1 $\rightarrow$ *dst*
  else *src1* << 1 $\rightarrow$ *dst*
  }
else nop

Instruction Type

Single-cycle

Delay Slots

0

Example 1

```
SUBC .L1   A0,A1,A0
```

	Before instruction			1 cycle after instruction	
A0	0000 125Ah	4698	A0	0000 024B4h	9396
A1	0000 1F12h	7954	A1	0000 1F12h	

Example 2

```
SUBC .L1   A0,A1,A0
```

	Before instruction			1 cycle after instruction	
A0	0002 1A31h	137777	A0	0000 2464h	128575
A1	0001 F63Fh	73490	A1	0001 F63Fh	

Syntax

SUB2 (.unit) *src1*, *src2*, *dst*

.unit = .S1 or .S2

Opcode map field used...	For operand type...	Unit	Opfield
src1 *src2* *dst*	sint xsint sint	.S1, .S2	010001

Description

The upper and lower halves of *src2* are subtracted from the upper and lower halves of *src1*. Any borrow from the lower-half subtraction does not affect the upper-half subtraction.

Execution

```
if (cond)   {
            ((lsb16(src1) – lsb16(src2)) and FFFFh) or
            ((msb16(src1) – msb16(src2)) << 16) → dst
            }
else        nop
```

Instruction Type

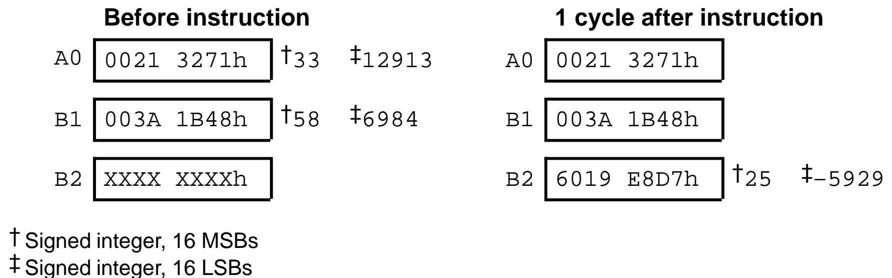
Single-cycle

Delay Slots

0

Example

```
SUB2 .S2   B1,A0,B2
```



	Before instruction				1 cycle after instruction		
A0	0021 3271h	†33	‡12913	A0	0021 3271h		
B1	003A 1B48h	†58	‡6984	B1	003A 1B48h		
B2	XXXX XXXXh			B2	6019 E8D7h	†25	‡–5929

† Signed integer, 16 MSBs
‡ Signed integer, 16 LSBs

Syntax

XOR (.unit) *src2*, *src1*, *dst*

.unit = .L1 or .L2, .S1 or .S2

Opcode map field used...	For operand type...	Unit	Opfield
src1 *src2* *dst*	uint xuint uint	.L1, .L2	1101111
src1 *src2* *dst*	*scst*5 xuint uint	.L1, .L2	1101110
src1 *src2* *dst*	uint xuint uint	.S1, .S2	001011
src1 *src2* *dst*	*scst*5 xuint uint	.S1, .S2	001010

Description

A bitwise exclusive-OR is performed between *src1* and *src2*. The result is placed in *dst*. The *scst*5 operands are sign extended to 32 bits.

Execution

if (cond) *src1* xor *src2* → *dst*
else nop

Instruction Type

Single-cycle

Delay Slots

0

Example 1

```
XOR .L1    A1,A2,A3
```

	Before instruction		1 cycle after instruction
A1	0721 325Ah	A1	0721 325Ah
A2	0019 0F12h	A2	0019 0F12h
A3	XXXX XXXXh	A3	0738 3D48h

Example 2

```
XOR .L2    B1,dh,B2
```

	Before instruction		1 cycle after instruction
B1	0000 1023h	B1	0000 1023h
B2	XXXX XXXXh	B2	0000 102Eh

Syntax **ZERO** (.unit) *dst*

.unit = .L1, .L2, .D1, .D2, .S1, or .S2

Description This is a pseudo operation used to zero out the *dst* register by subtracting the *dst* from itself and placing the result in the *dst*. The assembler uses the operation **SUB** (.unit) *src1, src2, dst* to perform this task where *src1* and *src2* both equal *dst*.

Execution

if (cond)	*dst* – *dst* → *dst*
else	nop

Instruction Type Single-cycle

Delay Slots 0

Chapter 4

TMS320C62xx Pipeline

The 'C62xx pipeline provides flexibility to simplify programming and improve performance. Two factors provide this flexibility:

- Control of the pipeline is simplified by eliminating pipeline interlocks.
- Increased pipelining eliminates traditional architectural bottlenecks in program fetch, data access, and multiply operations. This provides single-cycle throughput.

This chapter starts with a description of the pipeline flow. Highlights are:

- The pipeline can dispatch eight parallel instructions every cycle.
- Parallel instructions proceed simultaneously through each pipeline phase.
- Serial instructions proceed through the pipeline with a fixed relative phase difference between instructions.
- Load and store addresses appear on the CPU boundary during the same pipeline phase, eliminating read-after-write memory conflicts.

All instructions require the same number of pipeline phases for fetch and decode, but require a varying number of execute phases. This chapter contains a description of the number of execution phases for each type of instruction.

Finally, the chapter contains performance considerations for the pipeline. These considerations include the occurrence of fetch packets that contain multiple execute packets, execute packets that contain multicycle **NOP**s, and memory considerations for the pipeline. For more information about fully optimizing a program and taking full advantage of the pipeline, see the *TMS320C62xx Programmer's Guide*.

Topic **Page**

4.1 Pipeline Operation Overview 4-2
4.2 Pipeline Execution of Instruction Types 4-10
4.3 Performance Considerations 4-17

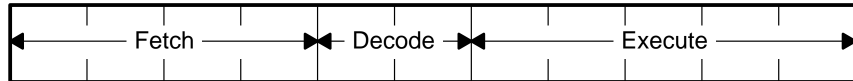
4.1 Pipeline Operation Overview

The pipeline phases are divided into three stages:

- ❑ Fetch
- ❑ Decode
- ❑ Execute

All instructions in the ’C62xx instruction set flow through the fetch, decode and execute stages of the pipeline. The fetch stage of the pipeline has four phases for all instructions, and the decode stage has two phases for all instructions. The execute stage of the pipeline requires a varying number of phases, depending on the type of instruction. The stages of the ’C62xx pipeline are shown in Figure 4–1.

Figure 4–1. Pipeline Stages



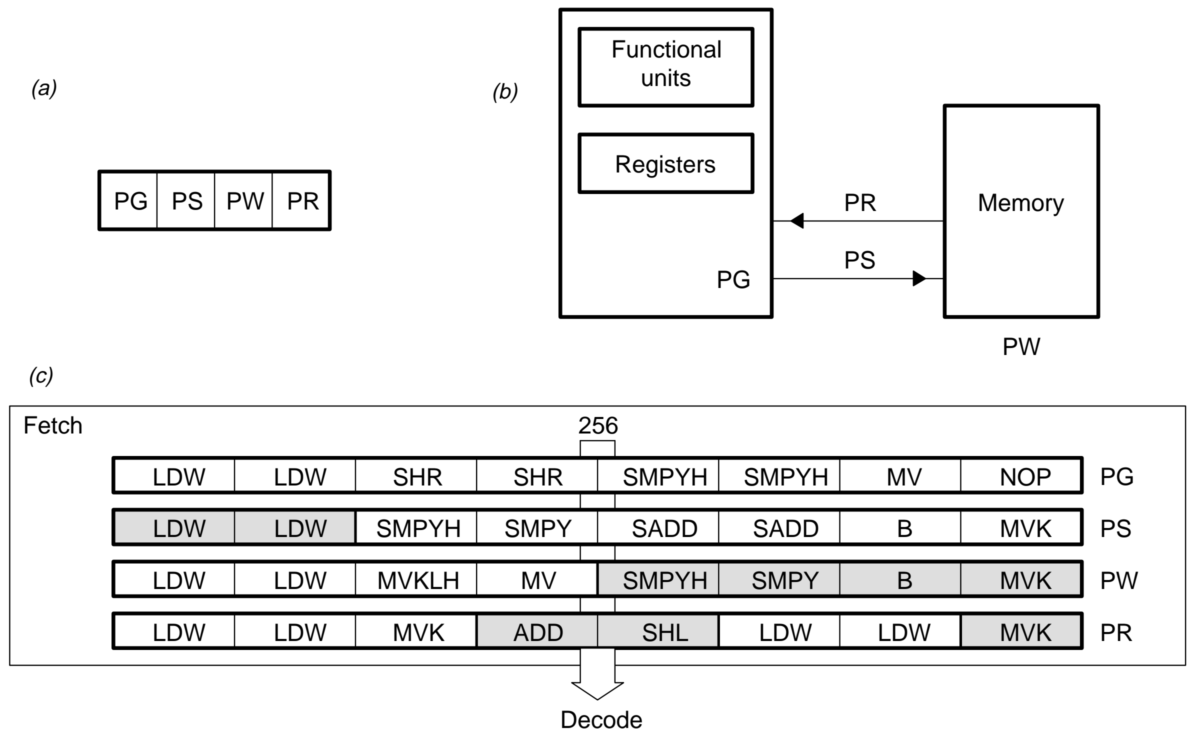
4.1.1 Fetch

The fetch phases of the pipeline are:

- **PG:** Program address generate
- **PS:** Program address send
- **PW:** Program access ready wait
- **PR:** Program fetch packet receive

The ’C62xx uses a fetch packet (FP) of eight instructions. All eight of the instructions proceed through fetch processing together, through the PG, PS, PW, and PR phases. Figure 4–2(a) shows the fetch phases in sequential order from left to right. Figure 4–2(b) shows a functional diagram of the flow of instructions through the fetch phases. During the PG phase, the program address is generated in the CPU. In the PS phase, the program address is sent to memory. In the PW phase, a memory read occurs. Finally, in the PR phase, the fetch packet is received at the CPU. Figure 4–2(c) shows fetch packets flowing through the phases of the fetch stage of the pipeline. In Figure 4–2(c), the first fetch packet (in PR) is made up of 4 execute packets, and the second and third fetch packets (in PS and PW) contain 2 execute packets each. The last fetch packet (in PG) contains a single-cycle execute packet of eight instructions.

Figure 4–2. Fetch Phases of the Pipeline



4.1.2 Decode

The decode phases of the pipeline are:

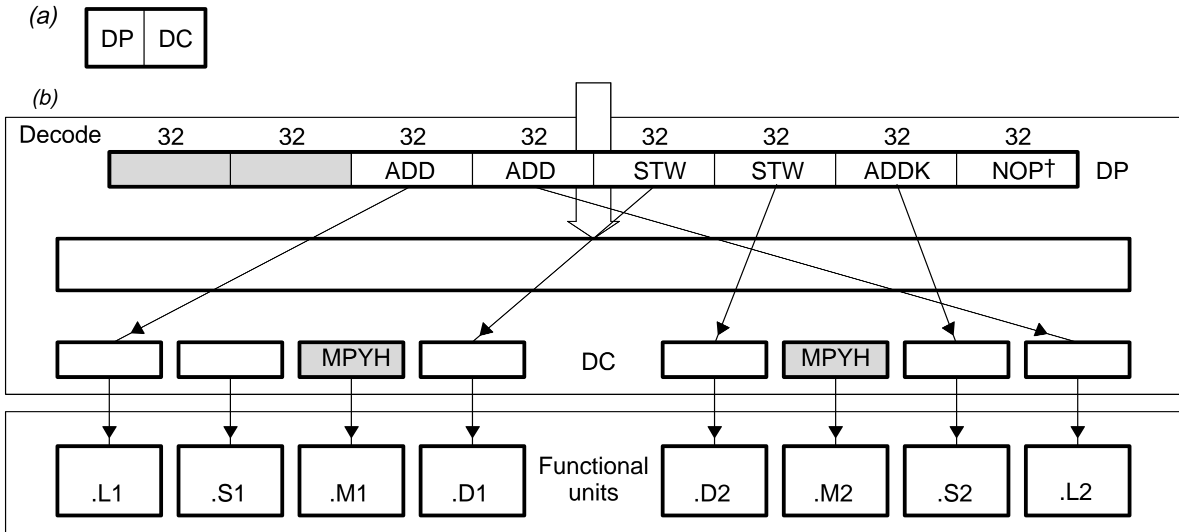
- **DP:** Instruction dispatch
- **DC:** Instruction decode

In the DP phase of the pipeline, the fetch packets are split into execute packets. Execute packets consist of one instruction or from two to eight parallel instructions. During the DP phase, the instructions in an execute packet are assigned to the appropriate functional units. In the DC phase, the the source registers, destination registers, and associated paths are decoded for the execution of the instructions in the functional units.

Figure 4–3(a) shows the decode phases in sequential order from left to right. Figure 4–3(b) shows a fetch packet that contains two execute packets as they are processed through the decode stage of the pipeline. The last six instructions of the fetch packet (FP) are parallel and form an execute packet (EP). This EP is in the dispatch phase (DP) of the decode stage. The arrows indicate each instruction's assigned functional unit for execution during the same cycle. The **NOP** instruction in the eighth slot of the FP is not dispatched to a functional unit because there is no execution associated with it.

The first two slots of the fetch packet (shaded below) represent an execute packet of two parallel instructions that were dispatched (DP) on the previous cycle. This execute packet contains two MPY instructions that are now in decode (DC) one cycle before execution. There are no instructions decoded for the .L, .S, and .D functional units for the situation illustrated.

Figure 4–3. Decode Phases of the Pipeline



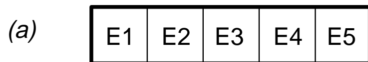
† NOP is not dispatched to a functional unit.

4.1.3 Execute

The execute portion of the pipeline is subdivided into five phases (E1–E5). Different types of instructions require different numbers of these phases to complete their execution. These phases of the pipeline play an important role in your understanding the device state at CPU cycle boundaries. The execution of different types of instructions in the pipeline is described in section 4.2, *Pipeline Execution of Instruction Types*. Figure 4–4(a) shows the execute phases of the pipeline in sequential order from left to right. Figure 4–4(b) shows the portion of the functional block diagram in which execution occurs.

Figure 4–4. Execute Phases of the Pipeline and Functional Block Diagram

(a)



(b)

E1
SADD .L1
B .S1
SMPY .M1
STH .D1
STH .D2
SMPYH .M2
SUB .S2
SADD .L2
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
Register file A
32
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
Register file B
Data 1
32
32
Data 2
Data memory interface control
32
16
16
16
16
32
0 1 2 3 4 5 6 7
8 9
Data address 1
Data address 2
Internal data memory
(byte addressable)
Execute

4.1.4 Summary of Pipeline Operation

Figure 4–5 shows all the phases in each stage of the ’C62xx pipeline in sequential order, from left to right.

Figure 4–5. Pipeline Phases

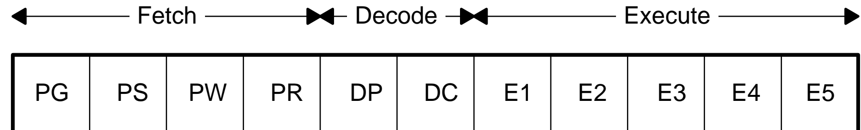
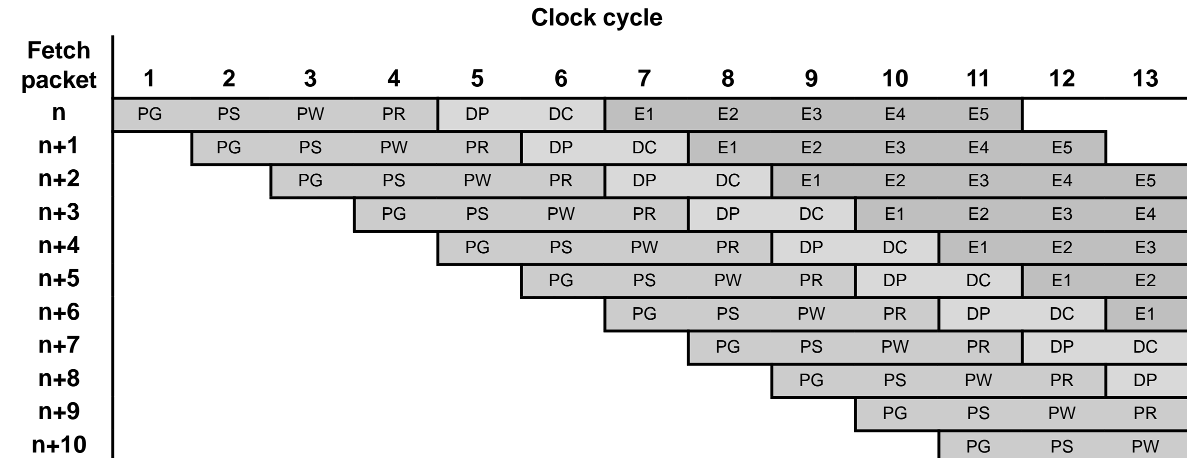


Figure 4–6 shows an example of the pipeline flow of consecutive fetch packets that contain eight parallel instructions. In this case, where the pipeline is full, all instructions in a fetch packet are in parallel and split into one execute packet per fetch packet. The fetch packets flow in lockstep fashion through each phase of the pipeline.

For example, observe cycle 7 in Figure 4–6. When the instructions from FPn reach E1, the instructions in the execute packet from FPn +1 are being decoded. FPn + 2 is in dispatch while FPs n + 3, n + 4, n + 5, and n + 6 are each in one of four phases of program fetch. See section 4.3, *Performance Considerations* on page 4-17, for additional detail on code flowing through the pipeline.

Figure 4–6. Pipeline Operation: One Execute Packet per Fetch Packet



Clock cycle

Fetch packet	1	2	3	4	5	6	7	8	9	10	11	12	13
n	PG	PS	PW	PR	DP	DC	E1	E2	E3	E4	E5		
n+1		PG	PS	PW	PR	DP	DC	E1	E2	E3	E4	E5	
n+2			PG	PS	PW	PR	DP	DC	E1	E2	E3	E4	E5
n+3				PG	PS	PW	PR	DP	DC	E1	E2	E3	E4
n+4					PG	PS	PW	PR	DP	DC	E1	E2	E3
n+5						PG	PS	PW	PR	DP	DC	E1	E2
n+6							PG	PS	PW	PR	DP	DC	E1
n+7								PG	PS	PW	PR	DP	DC
n+8									PG	PS	PW	PR	DP
n+9										PG	PS	PW	PR
n+10											PG	PS	PW

Table 4–1 summarizes the pipeline phases and what happens in each.

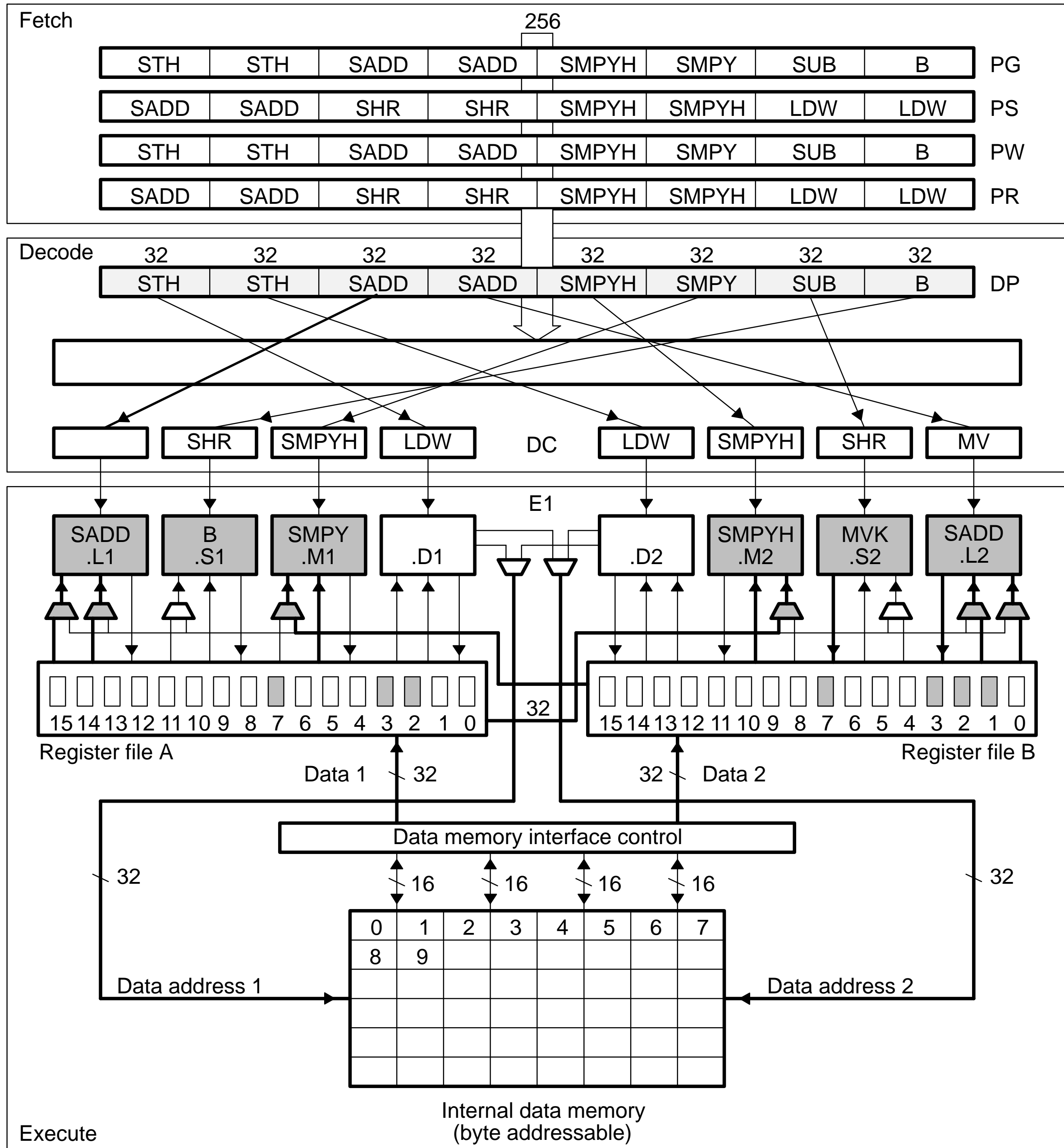
Table 4–1. Operations Occurring During Pipeline Phases

Stage	Phase	Symbol	During This Phase	Instruction Types Completed
Program fetch	Program address generate	PG	The address of the fetch packet is determined.	
	Program address send	PS	The address of the fetch packet is sent to memory.	
	Program wait	PW	A program memory access is performed.	
	Program data receive	PR	The fetch packet is at the CPU boundary.	
Program decode	Dispatch	DP	The next execute packet in the fetch packet is determined and sent to the appropriate functional units to be decoded.	
	Decode	DC	Instructions are decoded in functional units.	
Execute	Execute 1	E1	For all instruction types, the conditions for the instructions are evaluated and operands are read. For load and store instructions, address generation is performed and address modifications are written to a register file.† For branch instructions, branch fetch packet in PG phase is affected.† For single-cycle instructions, results are written to a register file.†	Single cycle
	Execute 2	E2	For load instructions, the address is sent to memory. For store instructions, the address and data are sent to memory.† Single-cycle instructions that saturate results set the SAT bit in the control status register (CSR) if saturation occurs.† For multiply instructions, results are written to a register file.†	Multiply
	Execute 3	E3	Data memory accesses are performed. Any multiply instruction that saturates results sets the SAT bit in the control status register (CSR) if saturation occurs.†	Store
	Execute 4	E4	For load instructions, data is brought to the CPU boundary.†	
	Execute 5	E5	For load instructions, data is written into a register.†	Load

† This assumes that the conditions for the instructions are evaluated as true. If the condition is evaluated as false, the instruction does not write any results or have any pipeline operation after E1.

Figure 4–7 shows a ’C62xx functional block diagram laid out vertically by stages of the pipeline.

Figure 4–7. Functional Block Diagram Based on Pipeline Phases



The pipeline operation is based on CPU cycles. A CPU cycle is the period during which a particular execute packet is in a particular pipeline phase. CPU cycle boundaries always occur at clock cycle boundaries.

As code flows through the pipeline phases, it is processed by different parts of the 'C62xx. Figure 4–7 shows a full pipeline with a fetch packet in every phase of fetch. One execute packet of eight instructions is being dispatched at the same time as a 7-instruction execute packet is in decode. The arrows between DP and DC correspond to the functional units identified in the code in Example 4–1.

Example 4–1. Execute Packet in Dispatch (DP) Phase in Figure 4–7

```
LOOP1:
      STH    .D1    A5,*A8++[2]
||    STH    .D2    B5,*B8++[2]
||    SADD   .L1    A2,A7,A2
||    SADD   .L2    B2,B7,B2
||    SMPYH  .M2X   B3,A3,B2
||    SMPY   .M1X   B3,A3,A2
||[B1]SUB    .S2    B1,1,B1
||[B1]B      .S1    LOOP1
```

In the DC phase portion of Figure 4–7, one box is empty because a NOP was the eighth instruction in the fetch packet in DC. Finally, the figure shows six functional units processing code during the same cycle of the pipeline. The instructions processed in E1 are shown in the code in Example 4–2.

Example 4–2. Execute Packet in E1 Phase of Execution in Figure 4–7

```
      SMPYH  .M2X   B3,A3,B2
||    SMPY   .M1X   B3,A3,A2
||    SADD   .L1    A2,A7,A2
||    SADD   .L2    B2,B7,B2
||    B      .S1    LOOP1
||    MVK    .S2    117,B1
```

Registers used by the instructions in E1 are shaded in Figure 4–7. The multiplexers used for the input operands to the functional units are also shaded in the figure. Note the bold crosspaths used by the MPY instructions.

Most 'C62xx instructions are single-cycle instructions, which means they have only one execution phase (E1). A small number of instructions require more than one execute phase. The types of instructions, each of which require different numbers of execute phases, are described in section 4.2, *Pipeline Execution of Instruction Types*.

4.2 Pipeline Execution of Instruction Types

The pipeline operation of the ’C62xx instructions can be categorized into six instruction types. Five of these are shown in Table 4–2 (**NOP** is not included in the table), which is a mapping of operations occurring in each execution phase for the different instruction types. The delay slots associated with each instruction type are listed in the bottom row.

Table 4–2. Execution Stage Length Description for Each Instruction Type

		Instruction Type				
		Single Cycle	**Multiply**	**Store**	**Load**	**Branch**
Execution phases	E1	Compute result and write to register	Read operands and start computations	Compute address	Compute address	Target code in PG‡
	E2		Compute result and write to register	Send address and data to memory	Send address to memory	
	E3			Access memory	Access memory	
	E4				Send data back to CPU	
	E5				Write data into register	
Delay slots		0	1	0†	4†	5‡

Notes: 1) This table assumes that the condition for each instruction is evaluated as true. If the condition is evaluated as false, the instruction does not write any results or have any pipeline operation after E1.

2) NOP is not shown and has no operation in any of the execution phases.

† See section 4.2.3 and 4.2.4 for more information on execution and delay slots for stores and loads.

‡ See section 4.2.5 for more information on branches.

The execution of instructions can be defined in terms of delay slots. A delay slot is a CPU cycle that occurs after the first execution phase (E1) of an instruction. Results from instructions with delay slots are not available until the end of the last delay slot. For example, a multiply instruction has one delay slot, which means that one CPU cycle elapses before the results of the multiply are available for use by a subsequent instruction. However, results are available from other instructions finishing execution during the same CPU cycle in which the multiply is in a delay slot.

4.2.1 Single-Cycle Instructions

Single-cycle instructions complete execution during the E1 phase of the pipeline. Figure 4–8 shows the fetch, decode, and execute phases of the pipeline that single-cycle instructions use.

Figure 4–8. Single-Cycle Instruction Phases

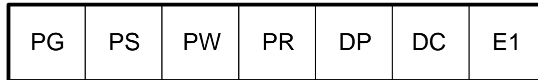
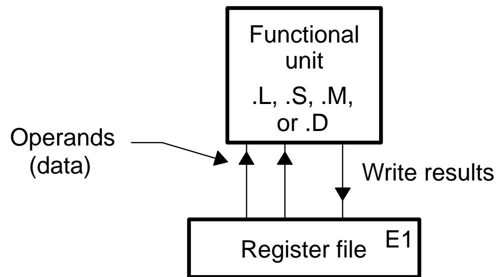


Figure 4–9 shows the single-cycle execution diagram. The operands are read, the operation is performed, and the results are written to a register, all during E1. Single-cycle instructions have no delay slots.

Figure 4–9. Single-Cycle Execution Block Diagram



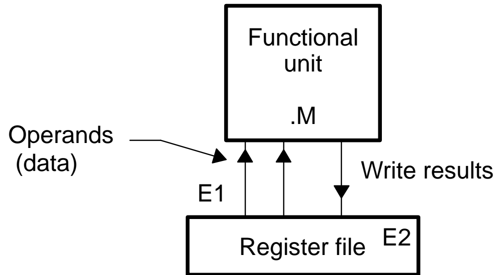
4.2.2 Multiply Instructions

Multiply instructions use both the E1 and E2 phases of the pipeline to complete their operations. Figure 4–10 shows the pipeline phases the multiply instructions use.

Figure 4–10. Multiply Instruction Phases

Figure 4–11 shows the operations occurring in the pipeline for a multiply. In the E1 phase, the operands are read and the multiply begins. In the E2 phase, the multiply finishes, and the result is written to the destination register. Multiply instructions have one delay slot.

Figure 4–11. Multiply Execution Block Diagram



4.2.3 Store Instructions

Store instructions require phases E1 through E3 to complete their operations. Figure 4–12 shows the pipeline phases the store instructions use.

Figure 4–12. Store Instruction Phases

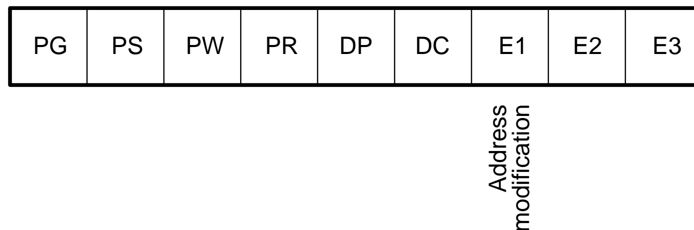
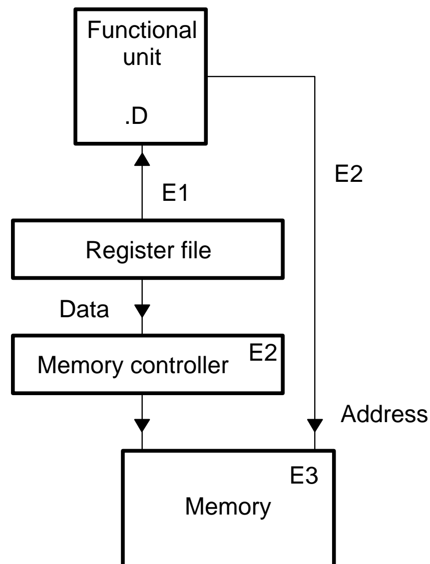


Figure 4–13 shows the operations occurring in the pipeline phases for a store. In the E1 phase, the address of the data to be stored is computed. In the E2 phase, the data and destination addresses are sent to data memory. In the E3 phase, a memory write is performed. The address modification is performed in the E1 stage of the pipeline. Even though stores finish their execution in the E3 phase of the pipeline, they have no delay slots.

Figure 4–13. Store Execution Block Diagram



When you perform a load and a store to the same memory location, these rules apply (*i* = cycle):

- When a load is executed before a store, the old value is loaded and the new value is stored.

i		LDW
i + 1		STW

- When a store is executed before a load, the new value is stored and the new value is loaded.

i		STW
i + 1		LDW

- When the instructions are executed in parallel, the old value is loaded first and then the new value is stored, but both occur in the same phase.

i		STW
i	||	LDW

There is additional explanation of why stores have zero delay slots in section 4.2.4.

4.2.4 Load Instructions

Data loads require all five of the pipeline execute phases to complete their operations. Figure 4–14 shows the pipeline phases the load instructions use.

Figure 4–14. Load Instruction Phases

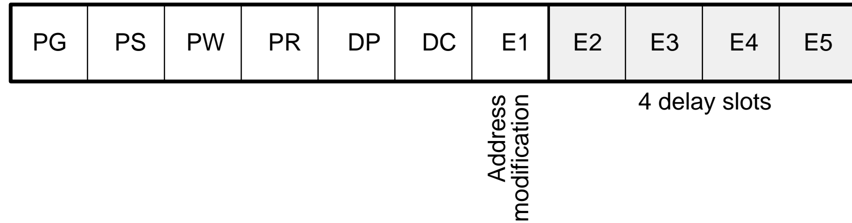
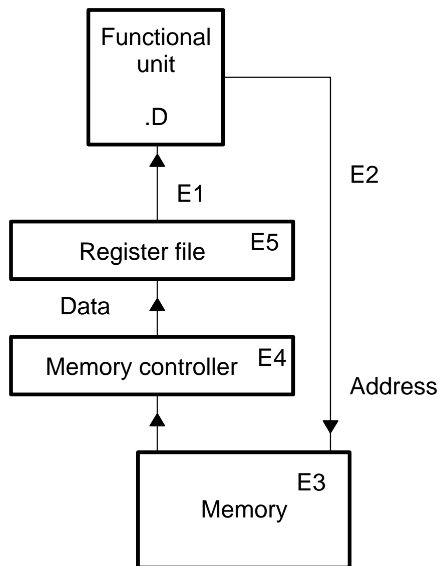


Figure 4–15 shows the operations occurring in the pipeline phases for a load. In the E1 phase, the data address pointer is modified in its register. In the E2 phase, the data address is sent to data memory. In the E3 phase, a memory read at that address is performed.

Figure 4–15. Load Execution Block Diagram



In the E4 stage of a load, the data is received at the CPU core boundary. Finally, in the E5 phase, the data is loaded into a register. Because data is not written to the register until E5, load instructions have four delay slots. Because pointer results are written to the register in E1, there are no delay slots associated with the address modification.

In the following code, pointer results are written to the A4 register in the first execute phase of the pipeline and data is written to the A3 register in the fifth execute phase.

```
LDW  .D1  *A4++,A3
```

Because a store takes three execute phases to write a value to memory and a load takes three execute phases to read from memory, a load following a store accesses the value placed in memory by that store in the cycle after the store is completed. This is why the store is considered to have zero delay slots.

4.2.5 Branch Instructions

Although branch takes one execute phase, there are five delay slots between the execution of the branch and execution of the target code. Figure 4–16 shows the pipeline phases used by the branch instruction and branch target code. The delay slots are shaded.

Figure 4–16. Branch Instruction Phases

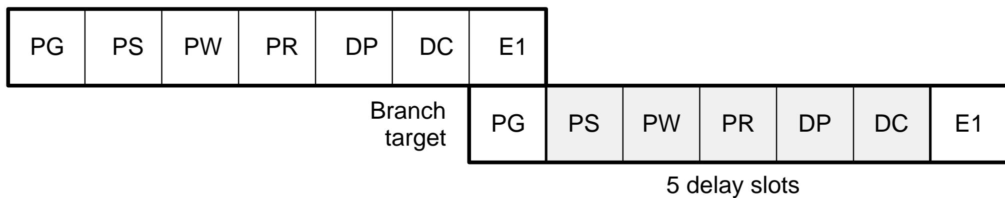
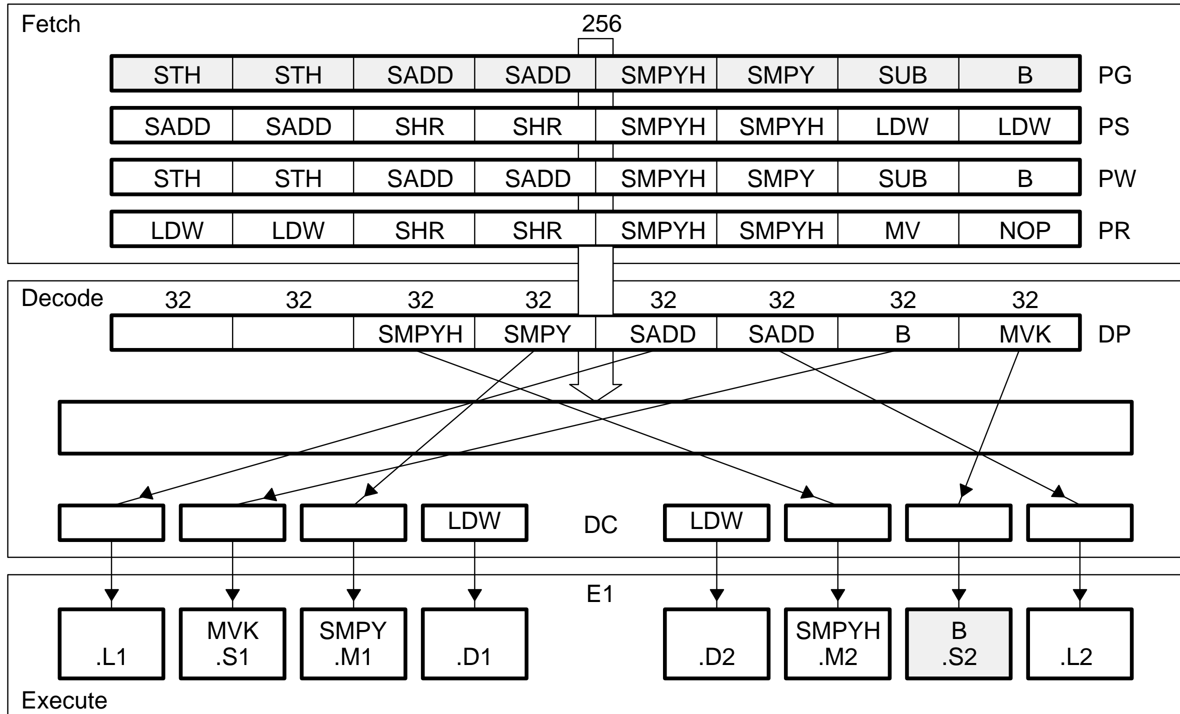


Figure 4–17 shows a branch execution block diagram. If a branch is in the E1 phase of the pipeline (in the .S2 unit in the figure), its branch target is in the fetch packet that is in PG during that same cycle (shaded in the figure). Because the branch target has to wait until it reaches the E1 phase to begin execution, the branch takes five delay slots before the branch target code executes.

Figure 4–17. Branch Execution Diagram



4.3 Performance Considerations

The 'C62xx pipeline is most effective when it is kept as full as the algorithms in the program allow it to be. It is useful to consider some situations that can affect pipeline performance.

A fetch packet (FP) is a grouping of eight instructions. Each FP can be split into from one to eight execute packets (EPs). Each EP contains instructions that execute in parallel. Each instruction executes in an independent functional unit. The effect on the pipeline of combinations of EPs that include varying numbers of parallel instructions, or just a single instruction that executes serially with other code, is considered here.

In general, the number of execute packets in a single FP defines the flow of instructions through the pipeline. Another defining factor is the instruction types in the EP. Each type of instruction has a fixed number of execute cycles that determines when this instruction's operations are complete. Section 4.3.2 covers the effect of including a multicycle **NOP** in an individual EP.

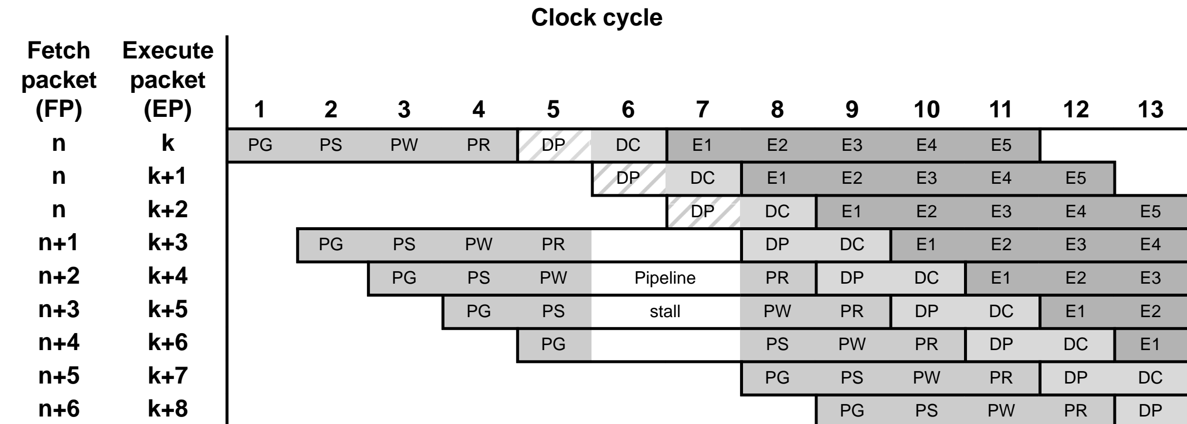
Finally, the effect of the memory system on the operation of the pipeline is considered. The access of program and data memory is discussed, along with memory stalls.

4.3.1 Pipeline Operation With Multiple Execute Packets in a Fetch Packet

Again referring to Figure 4–6 on page 4-6, pipeline operation is shown with eight instructions in every fetch packet. Figure 4–18, however, shows the pipeline operation with a fetch packet that contains multiple execute packets. Code for Figure 4–18 might have this layout:

```
   instruction A ; EP k          FP n
|| instruction B ;

   instruction C ; EP k + 1      FP n
|| instruction D
|| instruction E

   instruction F ; EP k + 2      FP n
|| instruction G
|| instruction H

   instruction I ; EP k + 3      FP n + 1
|| instruction J
|| instruction K
|| instruction L
|| instruction M
|| instruction N
|| instruction O
|| instruction P

... continuing with EPs k + 4 through k + 8, which have
eight instructions in parallel, like k + 3.
```

Figure 4–18. Pipeline Operation: Fetch Packets With Different Numbers of Execute Packets



In Figure 4–18, fetch packet n, which contains three execute packets, is shown followed by six fetch packets (n + 1 through n + 6), each with one execute packet (containing eight parallel instructions). The first fetch packet (n) goes through the program fetch phases during cycles 1–4. During these cycles, a program fetch phase is started for each of the fetch packets that follow.

In cycle 5, the program dispatch (DP) phase, the CPU scans the *p*-bits and detects that there are three execute packets (k through k + 2) in fetch packet n. This forces the pipeline to stall, which allows the DP phase to start for execute packets k + 1 and k + 2 in cycles 6 and 7. Once execute packet k + 2 is ready to move on to the DC phase (cycle 8), the pipeline stall is released.

The fetch packets n + 1 through n + 4 were all stalled so the CPU could have time to perform the DP phase for each of the three execute packets (k through k + 2) in fetch packet n. Fetch packet n + 5 was also stalled in cycles 6 and 7: it was not allowed to enter the PG phase until after the pipeline stall was released in cycle 8. The pipeline continues operation as shown with fetch packets n + 5 and n + 6 until another fetch packet containing multiple execution packets enters the DP phase, or an interrupt occurs.

4.3.2 Multicycle NOPs

The **NOP** instruction has an optional operand, *count*, that allows you to issue a single instruction for multicycle **NOP**s. A **NOP** 2, for example, fills in extra delay slots for the instructions in its execute packet and for all previous execute packets. If a **NOP** 2 is in parallel with an **MPY** instruction, the **MPY**'s results will be available for use by instructions in the next execute packet.

Figure 4–19 shows how a multicycle **NOP** can drive the execution of other instructions in the same execute packet. Figure 4–19(a) shows a **NOP** in an execute packet (in parallel) with other code. The results of the **LD**, **ADD**, and **MPY** will all be available during the proper cycle for each instruction. Hence **NOP** has no effect on the execute packet.

Figure 4–19(b) shows a multicycle **NOP** (**NOP** 5) in the same execute packet in place of the single-cycle **NOP**. The **NOP** 5 will cause no operation to perform other than the operations from the instructions inside its execute packet. The results of the **LD**, **ADD**, and **MPY** cannot be used by any other instructions until the **NOP** 5 period has completed.

Figure 4–19. Multicycle NOP in an Execute Packet

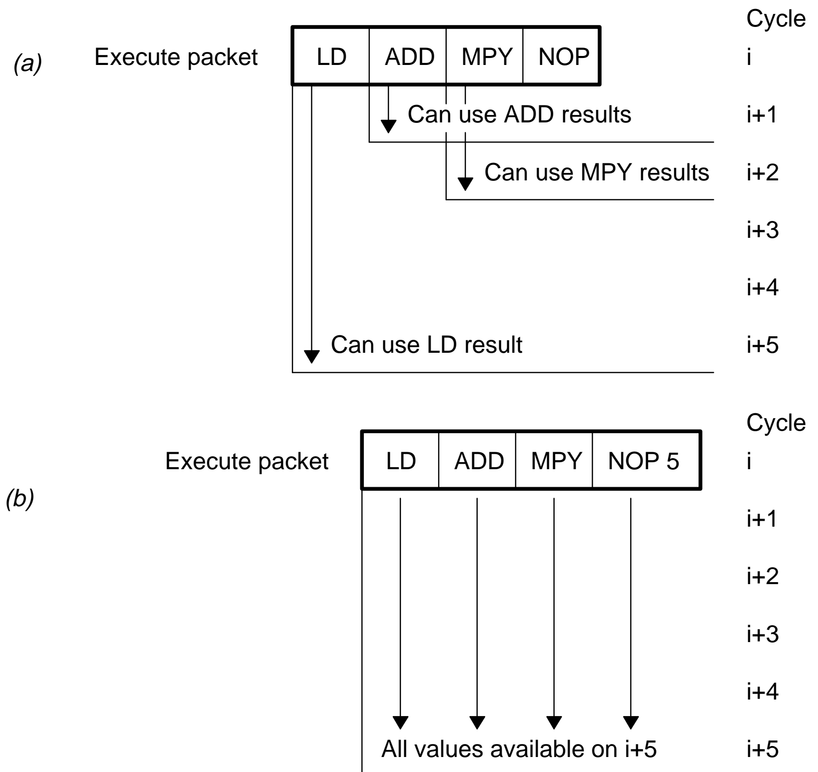
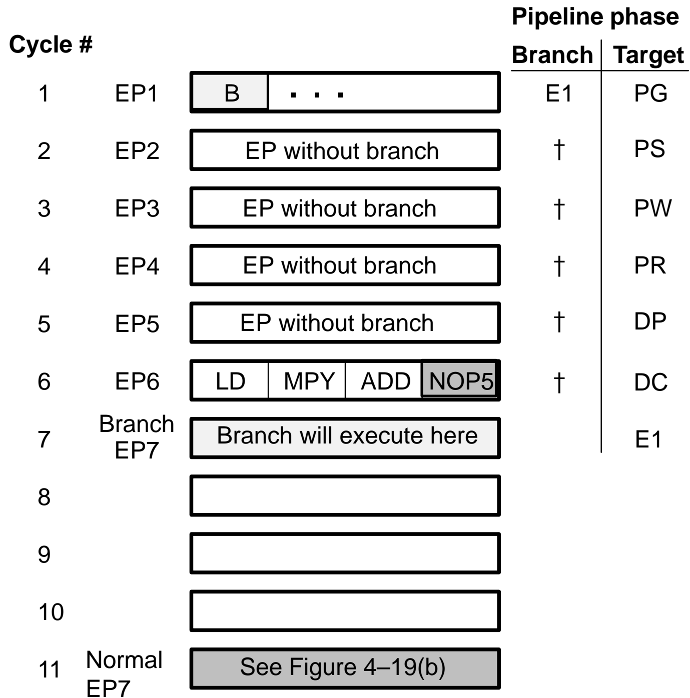


Figure 4–20 shows how a multicycle **NOP** can be affected by a branch. If the delay slots of a branch finish while a multicycle **NOP** is still dispatching **NOP**s into the pipeline, the branch overrides the multicycle **NOP** and the branch target begins execution five delay slots after the branch was issued.

*Figure 4–20. Branching and Multicycle **NOP**s*



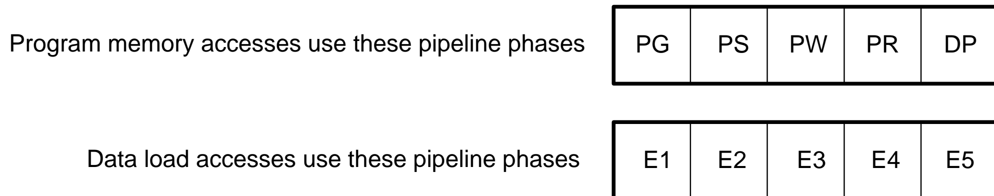
In one case, execute packet 1 (EP1) does not have a branch. The **NOP** 5 in EP6 will force the CPU to wait until cycle 11 to execute EP7.

In the other case, EP1 does have a branch. The delay slots of the branch coincide with cycles 2 through 6. Once the target code reaches E1 in cycle 7, it executes.

4.3.3 Memory Considerations

The ’C62xx has a memory configuration typical of a DSP, with program memory in one physical space and data memory in another physical space. Data loads and program fetches have the same operation in the pipeline, they just use different phases to complete their operations. With both data loads and program fetches, memory accesses are broken up into multiple phases. This enables the ’C62xx to access memory at a high speed. These phases are shown in Figure 4–21.

Figure 4–21. Pipeline Phases Used During Memory Accesses



To understand the memory accesses, compare data loads and instruction fetches/dispatches. The comparison is valid because data loads and program fetches operate on internal memories of the same speed on the ’C62xx and perform the same types of operations (listed in Table 4–3) to accommodate those memories. Table 4–3 shows the operation of program fetches pipeline versus the operation of a data load.

Table 4–3. Program Memory Accesses Versus Data Load Accesses

Operation	Program Memory Access Phase	Data Load Access Phase
Compute address	PG	E1
Send address to memory	PS	E2
Memory read/write	PW	E3
Program memory: receive fetch packet at CPU boundary Data load: receive data at CPU boundary	PR	E4
Program memory: send instruction to functional units Data load: send data to register	DP	E5

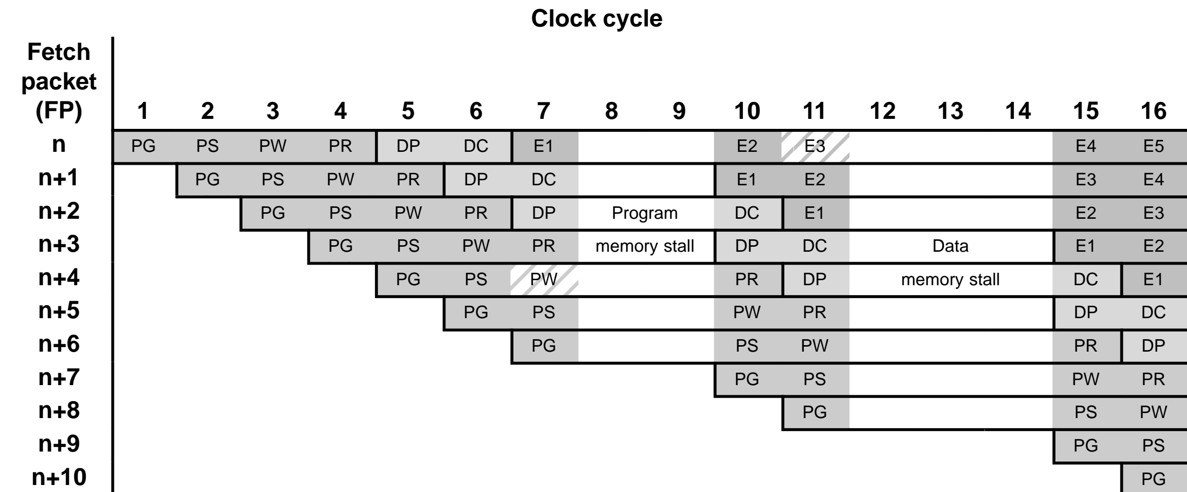
Depending on the type of memory and the time required to complete an access, the pipeline may stall to ensure proper coordination of data and instructions. This is discussed in section 4.3.3.1, *Memory Stalls.*

In the instance where multiple accesses are made to a single ported memory, the pipeline will stall to allow the extra access to occur. This is called a memory bank hit and is discussed in section 4.3.3.2, *Memory Bank Hits*.

4.3.3.1 Memory Stalls

A memory stall occurs when memory is not ready to respond to an access from the CPU. This access occurs during the PW phase for a program memory access and during the E3 phase for a data memory access. The memory stall causes all of the pipeline phases to lengthen beyond a single clock cycle, causing execution to take additional clock cycles to finish. The results of the program execution are identical whether a stall occurs or not. Figure 4–22 illustrates this point.

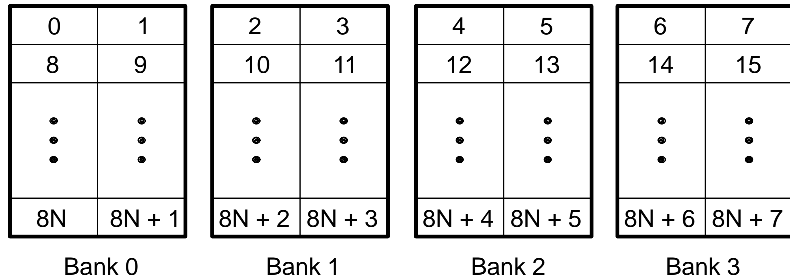
Figure 4–22. Program and Data Memory Stalls



4.3.3.2 Memory Bank Hits

Most ’C62xx devices use an interleaved memory bank scheme, as shown in Figure 4–23. Each number in the diagram represents a byte address. A load byte (**LDB**) instruction from address 0 loads byte 0 in bank 0. A load halfword (**LDH**) from address 0 loads the halfword value in bytes 0 and 1, which are also in bank 0. An **LDW** from address 0 loads bytes 0 through 3 in banks 0 and 1.

Figure 4–23. 4-Bank Interleaved Memory



Because each of these banks is single-ported memory, only one access to each bank is allowed per cycle. Two accesses to a single bank in a given cycle result in a memory stall that halts all pipeline operation for one cycle, while the second value is read from memory. Two memory operations per cycle are allowed without any stall, as long as they do not access the same bank.

Consider the code in Example 4–3. Because both loads are trying to access the same bank at the same time, one load must wait. The first **LDW** accesses bank 0 on cycle *i* + 2 (in the E3 phase) and the second **LDW** accesses bank 0 on cycle *i* + 3 (in the E3 phase). See Table 4–4 for identification of cycles and phases. The E4 phase for both LDW instructions is in cycle *i* + 4. To eliminate this extra phase, the loads must access data from different banks (B4 address would need to be in bank 1). For more information on programming topics, see the *TMS320C62xx Programmer’s Guide.*

Example 4–3. Load From Memory Banks

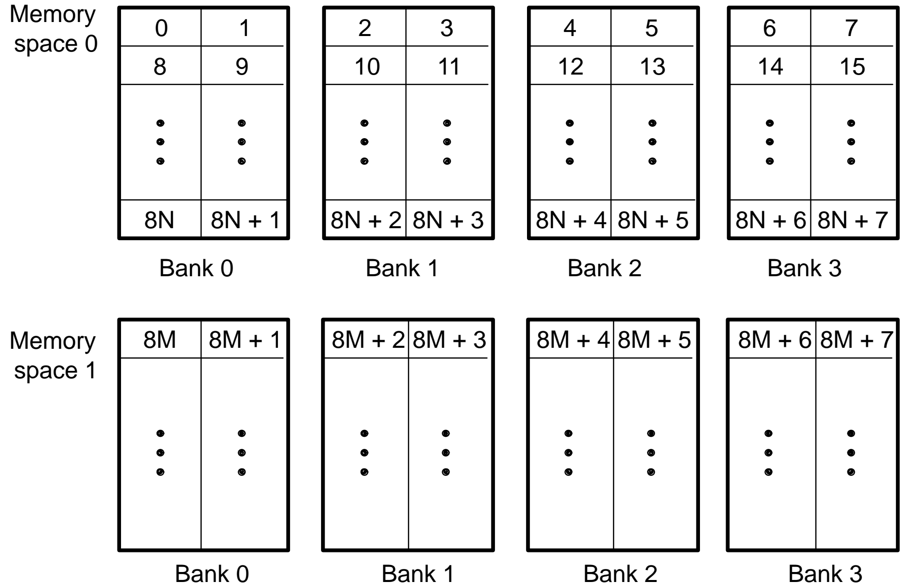
```
   LDW    .D1    *A4++,A5  ; load 1, A4 address is in bank 0
|| LDW    .D2    *B4++,B5  ; load 2, B4 address is in bank 0
```

Table 4–4. Loads in Pipeline From Example 4–3

	i	***i* +1**	***i* +2**	***i* +3**	***i* +4**	***i* +5**
LDW .D1 Bank 0	E1	E2	E3	*	E4	E5
LDW .D2 Bank 0	E1	E2	*	E3	E4	E5

For devices that have more than one memory space (see Figure 4–24), an access to bank 0 in one space does not interfere with an access to bank 0 in another memory space, and no pipeline stall occurs.

Figure 4–24. 4-Bank Interleaved Memory With Two Memory Spaces



The internal memory of the ’C62xx family varies from device to device. See the *TMS320C62xx Peripherals Reference Guide* to determine the memory spaces in your particular device.

Chapter 5

Interrupts

This chapter describes CPU interrupts, including reset and the nonmaskable interrupt (NMI). It details the related CPU control registers and their functions in controlling interrupts. It also describes interrupt processing, the method the CPU uses to detect automatically the presence of interrupts and divert program execution flow to your interrupt service code. Finally, the chapter describes the programming implications of interrupts.

Topic **Page**

5.1 Overview of Interrupts 5-2

5.2 Globally Enabling and Disabling Interrupts (Control Status Register–CSR) 5-11

5.3 Individual Interrupt Control 5-13

5.4 Interrupt Detection and Processing 5-18

5.5 Performance Considerations 5-23

5.6 Programming Considerations 5-24

5.1 Overview of Interrupts

Typically, DSPs work in an environment that contains multiple external asynchronous events. These events require tasks to be performed by the DSP when they occur. An interrupt is an event that stops the current process in the CPU so that the CPU can attend to the task needing completion because of the event. These interrupt sources can be on chip or off chip, such as timers, analog-to-digital converters, or other peripherals.

Servicing an interrupt involves saving the context of the current process, completing the interrupt task, restoring the registers and the process context, and resuming the original process. There are eight registers that control servicing interrupts.

An appropriate transition on an interrupt pin sets the pending status of the interrupt within the interrupt flag register (IFR). If the interrupt is properly enabled, the CPU begins processing the interrupt and redirecting program flow to the interrupt service routine.

5.1.1 Types of Interrupts and Signals Used

There are three types of interrupts on the ’C62xx CPU. These three types are differentiated by their priorities, as shown in Table 5–1. The reset interrupt has the highest priority and corresponds to the $\overline{\text{RESET}}$ signal. The nonmaskable interrupt is the interrupt of second highest priority and corresponds to the NMI signal. The lowest priority interrupts are interrupts 4–15. They correspond to the INT4–INT15 signals. $\overline{\text{RESET}}$, NMI, and some of the INT4–INT15 signals are mapped to pins on ’C62xx devices. Some of the INT4–INT15 interrupt signals are used by internal peripherals and some may be unavailable or can be used under software control. Check your data sheet to see your device’s interrupt specifications.

Table 5–1. Interrupt Priorities

Priority	Interrupt Name
Highest	Reset
	NMI
	INT4
	INT5
	INT6
	INT7
	INT8
	INT9
	INT10
	INT11
	INT12
	INT13
	INT14
Lowest	INT15

5.1.1.1 Reset ($\overline{\text{RESET}}$)

Reset is the highest priority interrupt and is used to halt the CPU and return it to a known state. The reset interrupt is unique in a number of ways:

- $\overline{\text{RESET}}$ is an active-low signal. All other interrupts are active-high signals.
- $\overline{\text{RESET}}$ must be held low for 10 clock cycles before it goes high again to reinitialize the CPU properly.
- The instruction execution in progress is aborted and all registers are returned to their default states.
- The reset interrupt service fetch packet must be located at address 0.
- $\overline{\text{RESET}}$ is not affected by branches.

5.1.1.2 Nonmaskable Interrupt (NMI)

NMI is the second-highest priority interrupt and is generally used to alert the CPU of a serious hardware problem such as imminent power failure.

For NMI processing to occur, the nonmaskable interrupt enable (NMIE) bit in the interrupt enable register must be set to 1. If NMIE is set to 1, the only condition that can prevent NMI processing is if the NMI occurs during the delay slots of a branch (whether the branch is taken or not).

NMIE is cleared to 0 at reset to prevent interruption of the reset. It is cleared at the occurrence of an NMI to prevent another NMI from being processed. You

cannot manually clear NMIE, but you can set NMIE to allow nested NMIs. While NMI is cleared, all maskable interrupts (INT4–INT15) are disabled.

5.1.1.3 Maskable Interrupts (INT4–INT15)

The ’C62xx CPU has twelve interrupts that are maskable. These have lower priority than the NMI and reset interrupts. These interrupts can be associated with external devices, on-chip peripherals, software control, or not be available.

Assuming that a maskable interrupt does not occur during the delay slots of a branch (this includes conditional branches that do not complete execution due to a false condition), the following conditions must be met to process a maskable interrupt:

- ❑ The global interrupt enable bit (GIE) bit in the control status register (CSR) is set to1.
- ❑ The NMIE bit in the interrupt enable register (IER) is set to1.
- ❑ The corresponding interrupt enable (IE) bit in the IER is set to1.
- ❑ The corresponding interrupt occurs, which sets the corresponding bit in the IFR to 1 and there are no higher priority interrupt flag (IF) bits set in the IFR.

5.1.1.4 Interrupt Acknowledgment (IACK and INUMx)

The IACK and INUMx signals alert hardware external to the ’C62xx that an interrupt has occurred and is being processed. The IACK signal indicates that the CPU has begun processing an interrupt. The INUMx signals (INUM3–INUM0) indicate the number of the interrupt (bit position in the IFR) that is being processed.

For example:

INUM3 = 0 (MSB)
INUM2 = 1
INUM1 = 1
INUM0 = 1 (LSB)

Together, these signals provide the 4-bit value 0111, indicating INT7 is being processed.

5.1.2 Interrupt Service Table (IST)

When the CPU begins processing an interrupt, it references the interrupt service table (IST). The IST is a table of fetch packets that contain code for servicing the interrupts. The IST consists of 16 consecutive fetch packets. Each interrupt service fetch packet contains eight instructions. A simple interrupt service routine may fit in an individual fetch packet.

The addresses and contents of the IST are shown in Figure 5–1. Because each fetch packet contains eight 32-bit instruction words (or 32 bytes), each address in the table is incremented by 32 bytes (20h) from the one adjacent to it.

Figure 5–1. Interrupt Service Table

Address	Interrupt service table (IST)
000h	$\overline{\text{RESET}}$ ISFP
020h	NMI ISFP
040h	Reserved
060h	Reserved
080h	INT4 ISFP
0A0h	INT5 ISFP
0C0h	INT6 ISFP
0E0h	INT7 ISFP
100h	INT8 ISFP
120h	INT9 ISFP
140h	INT10 ISFP
160h	INT11 ISFP
180h	INT12 ISFP
1A0h	INT13 ISFP
1C0h	INT14 ISFP
1E0h	INT15 ISFP

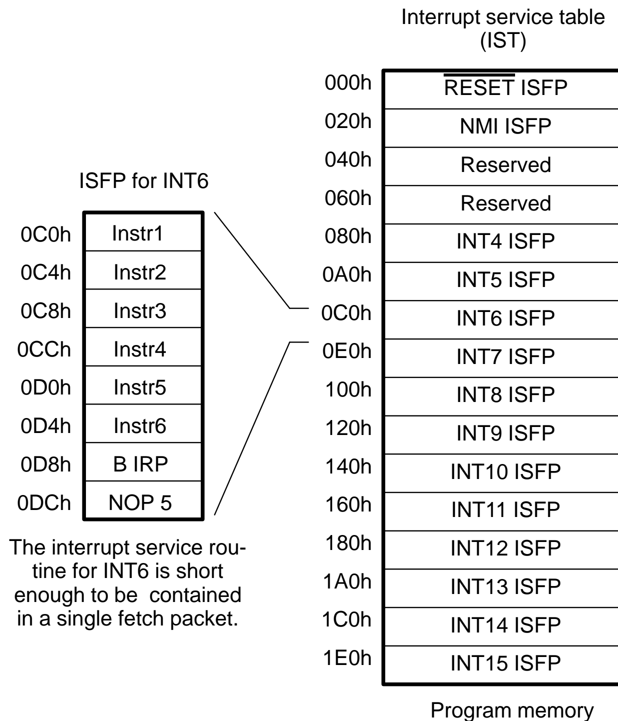
Program memory

5.1.2.1 Interrupt Service Fetch Packet (ISFP)

An ISFP is a fetch packet used to service an interrupt. Figure 5–2 shows an ISFP that contains an interrupt service routine small enough to fit in a single fetch packet (FP). To branch back to the main program, the FP contains a branch to the interrupt return pointer instruction (**B IRP**). This is followed by a **NOP** 5 instruction to allow the branch target to reach the execution stage of the pipeline.

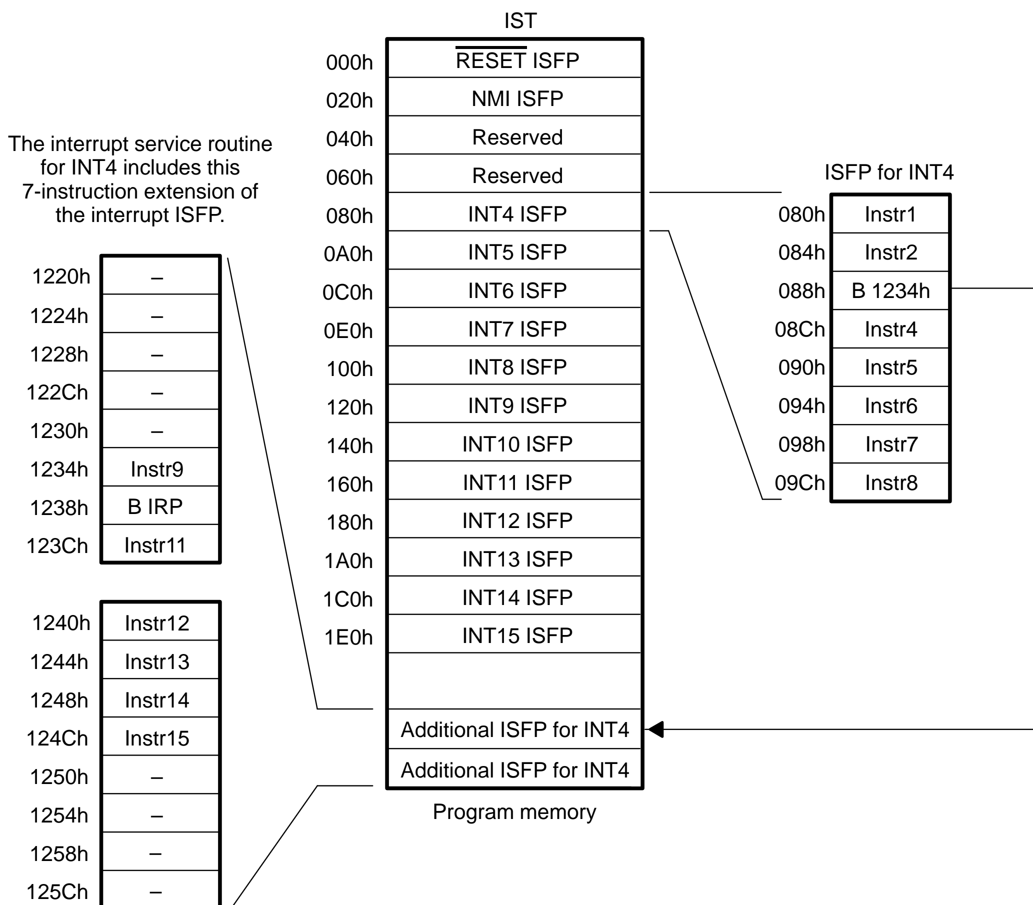
> **Note:**
>
> If the **NOP** 5 was not in the routine, the CPU would execute the next five execute packets that are associated with the next ISFP.

Figure 5–2. Interrupt Service Fetch Packet



If the interrupt service routine for an interrupt is too large to fit in a single FP, a branch to the location of additional interrupt service routine code is required. Figure 5–3 shows that the interrupt service routine for INT4 was too large for a single FP, and a branch to memory location 1234h is required to complete the interrupt service routine.

Figure 5–3. IST With Branch to Additional Interrupt Service Code Located Outside the IST



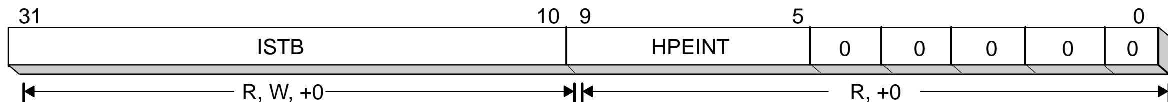
Note:

The instruction **B** 1234h branches into the middle of a fetch packet (at 1220h) and processes code starting at address 1234h. The CPU ignores code from address 1220–1230h, even if it is in parallel to code at address 1234h.

5.1.2.2 Interrupt Service Table Pointer Register (ISTP)

The interrupt service table pointer (ISTP) register is used to locate the interrupt service routine. One field, ISTB identifies the base portion of the address of the IST; another field, HPEINT, identifies the specific interrupt and locates the specific fetch packet within the IST. Figure 5–4 shows the fields of the ISTP. Table 5–2 describes the fields and how they are used.

Figure 5–4. Interrupt Service Table Pointer (ISTP)



Legend: R — Readable by the **MVC** instruction
W — Writeable by the **MVC** instruction
+0 — Value is cleared at reset

Table 5–2. ISTP Fields

Bits	Field Name	Description
0–4		Set to 0 (fetch packets must be aligned on 8-word (32-byte) boundaries).
5–9	HPEINT	Highest priority enabled interrupt. This field gives the number (related bit position in the IFR) of the highest priority interrupt (as defined in Table 5–1) that is enabled by its bit in the IER. Thus, the ISTP can be used for manual branches to the highest priority enabled interrupt. If no interrupt is pending and enabled, HPEINT contains the value 00000b. The corresponding interrupt need not be enabled by NMIE (unless it is NMI) or by GIE.
10–31	ISTB	Interrupt service table base portion of the IST address. This field is set to 0 on reset. Thus, upon startup the IST must reside at address 0. After reset, you can relocate the IST by writing a new value to ISTB. If relocated, the first ISFP (corresponding to $\overline{\text{RESET}}$) is never executed via interrupt processing, because reset sets the ISTB to 0. See Example 5–1.

The reset fetch packet must be located at address 0, but the rest of the IST can be at any program memory location that is on a 256-word boundary. The location of the IST is determined by the interrupt service table base (ISTB). Example 5–1 shows the relationship of the ISTB to the table location.

Example 5–1. Relocation of Interrupt Service Table

(a) Relocating the IST to 800h

1) Copy the IST, located between 0h and 200h, to the memory location between 800h and A00h.

2) Write 800h to the ISTP register:

```
MVK 800h, A2
MVC A2, ISTP
```

ISTP = 800h = 1000 0000 0000b

(b) How the ISTP directs the CPU to the appropriate ISFP in the relocated IST

Assume: IFR = BBC0h = 1011 1011 1100 0000b
IER = 1230h = 0001 0010 0011 0001b
2 enabled interrupts pending: INT9 and INT12

The 1s in the IFR indicate pending interrupts; the 1s in the IER indicate the interrupts that are enabled. INT9 has a higher priority than INT12, so HPEINT is encoded with the value for INT9, 01001b.

HPEINT corresponds to bits 9–5 of the ISTP:
ISTP = 1001 0010 0000b = 920h = address of INT9

IST

Address	Content
0	$\overline{\text{RESET}}$ ISFP
800h	$\overline{\text{RESET}}$ ISFP
820h	NMI ISFP
840h	Reserved
860h	Reserved
880h	INT4 ISFP
8A0h	INT5 ISFP
8C0h	INT6 ISFP
8E0h	INT7 ISFP
900h	INT8 ISFP
920h	INT9 ISFP
940h	INT10 ISFP
96h0	INT11 ISFP
980h	INT12 ISFP
9A0h	INT13 ISFP
9C0h	INT14 ISFP
9E0h	INT15 ISFP

Program memory

5.1.3 Summary of Interrupt Control Registers

Table 5–3 lists the eight interrupt control registers on the ’C62xx devices. The control status register (CSR) and the interrupt enable register (IER) enable or disable interrupt processing. The interrupt flag register (IFR) identifies pending interrupts. The interrupt set (ISR) and interrupt clear (ICR) registers can be used in manual interrupt processing.

There are three pointer registers. ISTP points to the interrupt service table. NRP and IRP are the return pointers used when returning from a nonmaskable or a maskable interrupt, respectively. More information on all the registers can be found at the locations listed in the table.

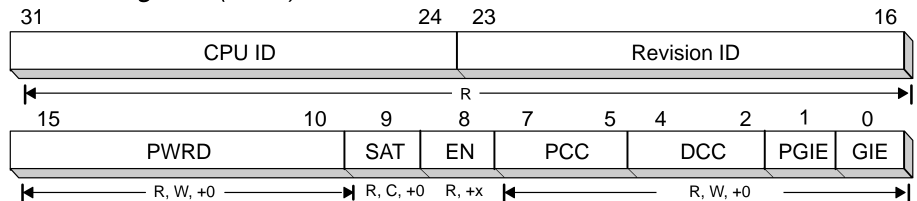
Table 5–3. Interrupt Control Registers

Abbreviation	Name	Description	Page Number
CSR	Control status register	Allows you to globally set or disable interrupts	5-11
IER	Interrupt enable register	Allows you to enable interrupts	5-13
IFR	Interrupt flag register	Shows the status of interrupts	5-14
ISR	Interrupt set register	Allows you to set flags in the IFR manually	5-14
ICR	Interrupt clear register	Allows you to clear flags in the IFR manually	5-14
ISTP	Interrupt service table pointer	Pointer to the beginning of the interrupt service table	5-8
NRP	Nonmaskable interrupt return pointer	Contains the return address used on return from a nonmaskable interrupt. This return is accomplished via the B NRP instruction.	5-16
IRP	Interrupt return pointer	Contains the return address used on return from a maskable interrupt. This return is accomplished via the B IRP instruction.	5-17

5.2 Globally Enabling and Disabling Interrupts (Control Status Register–CSR)

The control status register (CSR) contains two fields that control interrupts: GIE and PGIE, as shown in Figure 5–5. The other fields of the registers serve other purposes and are discussed in section 2.4 on page 2-9.

Figure 5–5. Control Status Register (CSR)



Legend: R Readable by the **MVC** instruction
W Writeable by the **MVC** instruction
+x Value undefined after reset
+0 Value is zero after reset
C Clearable using the **MVC** instruction

Table 5–4. CSR Interrupt Control Field Descriptions

Bit	Field Name	Description
0	GIE	Global interrupt enable; globally enables or disables all maskable interrupts. GIE = 1 maskable interrupts globally enabled GIE = 0, maskable interrupts globally disabled
1	PGIE	Previous GIE; saves the value of GIE when an interrupt is taken. This value is used on return from an interrupt.

The global interrupt enable (GIE) allows you to enable or disable all maskable interrupts by controlling the value of a single bit. GIE is bit 0 of the control status register (CSR).

- GIE = 1 enables the maskable interrupts so that they are processed.
- GIE = 0 disables the maskable interrupts so that they are not processed.

Bit 1 of the CSR is PGIE and contains the previous value of GIE. During processing of a maskable interrupt, PGIE is loaded with GIE and GIE is cleared. GIE is cleared during a maskable interrupt to keep another maskable interrupt from occurring before the device state has been saved. Upon return from an interrupt, by way of the **B IRP** instruction, the PGIE value is copied back to GIE and remains unchanged. The purpose of PGIE is to allow proper clearing of GIE when an interrupt has already been detected for processing.

Suppose the CPU begins processing an interrupt. Just as the interrupt processing begins, GIE is being cleared by you writing a 0 to bit 0 of the CSR with the MVC instruction. GIE is cleared by the MVC instruction prior to being copied to PGIE. Upon returning from the interrupt, PGIE is copied back to GIE, resulting in GIE being cleared as directed by your code.

Example 5–2 shows how to disable maskable interrupts globally and Example 5–3 shows how to enable maskable interrupts globally.

Example 5–2. Code Sequence to Disable Maskable Interrupts Globally

```
MVC .S2    CSR,B0    ; get CSR
AND .S2    -2,B0,B0  ; get ready to clear GIE
MVC .S2    B0,CSR    ; clear GIE
```

Example 5–3. Code Sequence to Enable Maskable Interrupts Globally

```
MVC .S2    CSR,B0    ; get CSR
OR  .S2    1,B0,B0   ; get ready to set GIE
MVC .S2    B0,CSR    ; set GIE
```

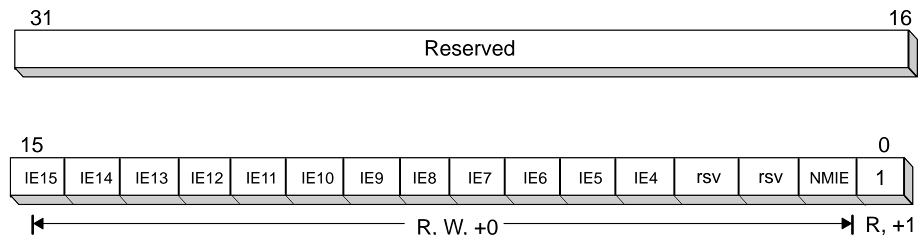
5.3 Individual Interrupt Control

Servicing interrupts effectively requires individual control of all three types of interrupts: reset, nonmaskable, and maskable. Enabling and disabling individual interrupts is done with the interrupt enable register (IER). The status of pending interrupts is stored in the interrupt flag register (IFR). Manual interrupt processing can be accomplished through the use of the interrupt set register (ISR) and interrupt clear register (ICR). The interrupt return pointers restore context after servicing nonmaskable and maskable interrupts.

5.3.1 Enabling and Disabling Interrupts (Interrupt Enable Register–IER)

You can enable and disable individual interrupts by setting and clearing bits in the IER that correspond to the individual interrupts. An interrupt can trigger interrupt processing only if the corresponding bit in the IER is set. Bit 0, corresponding to reset, is not writeable and is always read as 1, so the reset interrupt is always enabled. You cannot disable the reset interrupt. Bits IE4–IE15 can be written as 1 or 0, enabling or disabling the associated interrupt, respectively. The IER is shown in Figure 5–6.

Figure 5–6. Interrupt Enable Register (IER)



Legend: R = Readable by the **MVC** instruction
W = Writeable by the **MVC** instruction
rsv = Reserved
+1 = Value after reset
+0 = Value after reset

When NMIE = 0, all nonreset interrupts are disabled, preventing interruption of an NMI. NMIE is cleared at reset to prevent any interruption of processor initialization until you enable NMI. After reset, you must set NMIE to enable the NMI and to allow INT15–INT4 to be enabled by GIE and the appropriate IER bit. You cannot manually clear the NMIE; the bit is unaffected by a write of 0. NMIE is also cleared by the occurrence of an NMI. If cleared, NMIE is set only by completing a **B NRP** instruction or by a write of 1 to NMIE. Example 5–4 and Example 5–5 show code for enabling and disabling individual interrupts, respectively.

Example 5–4. Code Sequence to Enable an Individual Interrupt–INT9

```
MVK.S2    200h,B1 ; set bit 9
MVC.S2    IER,B0 ; get IER
OR .S2    B1,B0,B0 ; get ready to set IE9
MVC.S2    B0,IER ; set bit 9 in IER
```

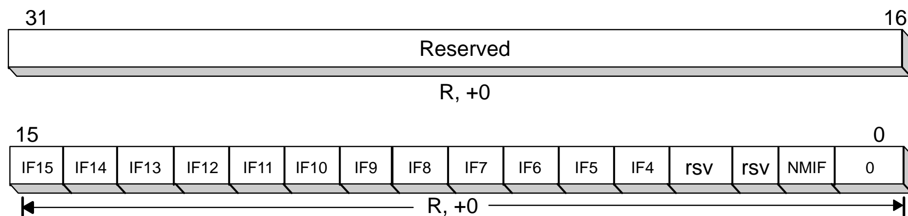
Example 5–5. Code Sequence to Disable an Individual Interrupt–INT9

```
MVK.S2    FDFFh,B1 ; clear bit 9
MVC.S2    IER,B0
AND.S2    B1,B0,B0 ; get ready to clear IE9
MVC.S2    B0,IER ; clear bit 9 in IER
```

5.3.2 Status of, Setting, and Clearing Interrupts (Interrupt Flag, Set, and Clear Registers – IFR, ISR, ICR)

The interrupt flag register (IFR) contains the status of INT4–INT15 and NMI. Each interrupt's corresponding bit in the IFR is set to 1 when that interrupt occurs; otherwise, the bits have a value of 0. If you want to check the status of interrupts, use the MVC instruction to read the IFR. Figure 5–7 shows the IFR.

Figure 5–7. Interrupt Flag Register (IFR)



Legend: R = Readable by the **MVC** instruction
+0 = Cleared at reset
rsv = Reserved

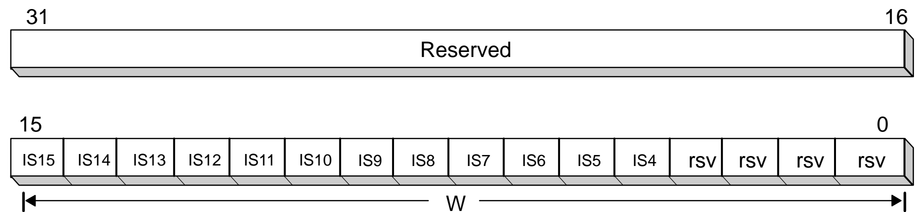
The interrupt set register (ISR) and interrupt clear register (ICR) allow you to set or clear maskable interrupts manually in the IFR. Writing a 1 to IS4–IS15 of the ISR causes the corresponding interrupt flag to be set in the IFR. Similarly, writing a 1 to a bit of the ICR causes the corresponding interrupt flag to be cleared. Writing a 0 to any bit of either the ISR or the ICR has no effect. Incoming interrupts have priority and override any write to the ICR. You cannot set or clear any bit in the ISR or ICR to affect NMI or reset.

Note:

Any write to the ISR or ICR (by the **MVC** instruction) effectively has one delay slot because the results cannot be read (by the **MVC** instruction) in the IFR until two cycles after the write to the ISR or ICR.

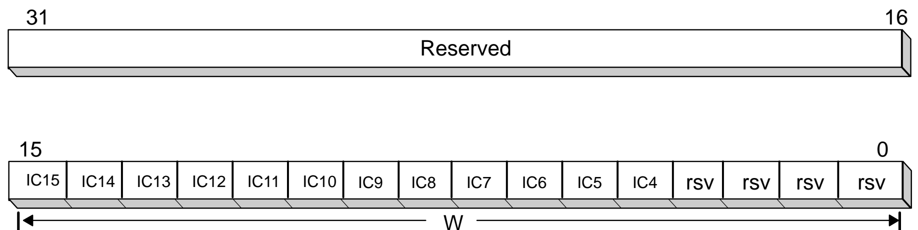
Any write to the ICR is ignored by a simultaneous write to the same bit in the ISR.

Figure 5–8. Interrupt Set Register (ISR)



Legend: W = Writeable by the **MVC** instruction
rsv = Reserved

Figure 5–9. Interrupt Clear Register (ICR)



Legend: W = Writeable by the **MVC** instruction
rsv = Reserved

Example 5–6 and Example 5–7 show code examples to set and clear individual interrupts.

Example 5–6. Code to Set an Individual Interrupt (INT6) and Read the Flag Register

```
MVK.S2   40h,B3
MVC.S2   B3,ISR
NOP
MVC.S2   IFR,B4
```

Example 5–7. Code to Clear an Individual Interrupt (INT6) and Read the Flag Register

```
MVK.S2   40h,B3
MVC.S2   B3,ICR
NOP
MVC.S2   IFR,B4
```

5.3.3 Returning From Interrupt Servicing

After $\overline{RESET}$ goes high, the control registers are brought to a known value and program execution begins at address 0h. After nonmaskable and maskable interrupt servicing, use a branch to the corresponding return pointer register to continue the previous program execution.

5.3.3.1 CPU State After $\overline{RESET}$

After $\overline{RESET}$, the control registers and bits will contain the corresponding values:

- AMR, ISR, ICR, IFR, and ISTP = 0h
- IER = 1h
- IRP and NRP = undefined
- Bits 15–0 of the CSR = 100h in little endian mode, 000h in big endian mode

5.3.3.2 Returning From Nonmaskable Interrupts (NMI Return Pointer Register–NRP)

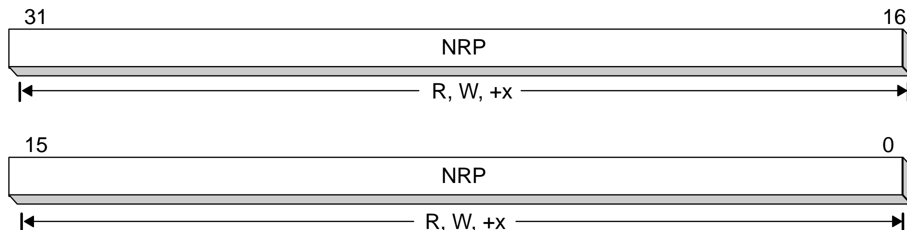
The NMI return pointer register contains the return pointer that directs the CPU to the proper location to continue program execution after NMI processing. A branch using the address in the NRP (**B NRP**) in your interrupt service routine returns to the program flow when NMI servicing is complete. Example 5–8 shows how to return from an NMI.

Example 5–8. Code to Return from NMI

```
B   .S2    NRP    ; return, sets NMIE
NOP        5      ; delay slots
```

The NRP contains the 32-bit address of the first execute packet in the program flow that was not executed because of a nonmaskable interrupt. Although you can write a value to this register, any subsequent interrupt processing may overwrite that value. Figure 5–10 shows the NRP register.

Figure 5–10. NMI Return Pointer (NRP)



Legend: R = Readable by the **MVC** instruction
W = Writeable by the **MVC** instruction
+x = value undefined after reset

5.3.3.3 Returning From Maskable Interrupts (Interrupt Return Pointer Register - IRP)

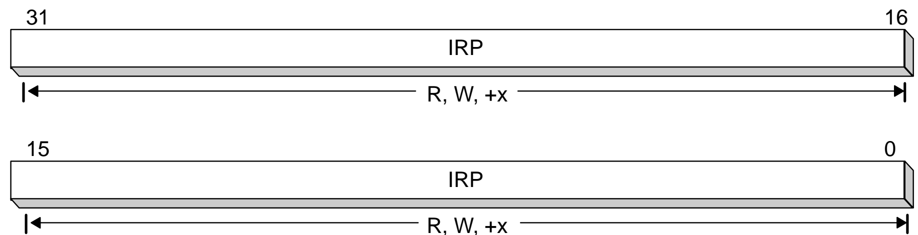
The interrupt return pointer register contains the return pointer that directs the CPU to the proper location to continue program execution after processing a maskable interrupt. A branch using the address in the IRP (**B IRP**) in your interrupt service routine returns to the program flow when interrupt servicing is complete. Example 5–9 shows how to return from a maskable interrupt.

Example 5–9. Code to Return from a Maskable Interrupt

```
B   .S2    IRP    ; return, moves PGIE to GIE
NOP        5      ; delay slots
```

The IRP contains the 32-bit address of the first execute packet in the program flow that was not executed because of a maskable interrupt. Although you can write a value to this register, any subsequent interrupt processing may overwrite that value. Figure 5–11 shows the IRP register.

Figure 5–11. Interrupt Return Pointer (IRP)



Legend: R = Readable by the **MVC** instruction
W = Writeable by the **MVC** instruction
+x = Value undefined after reset

5.4 Interrupt Detection and Processing

When an interrupt occurs, it sets a flag in the IFR register. Depending on certain conditions, the interrupt may or may not be processed. This section discusses the mechanics of setting the flag bit, the conditions for processing an interrupt, and the order of operation for detecting and processing an interrupt. The similarities and differences between reset and nonreset interrupts are also discussed.

5.4.1 Setting the Interrupt Flag – Nonreset

Figure 5–12 shows the processing of a nonreset interrupt (INTm). The flag (IFm) for INTm in the IFR is set following the low-to-high transition of the INTm signal on the CPU boundary. This transition is detected on a clock-cycle by clock-cycle basis and is not affected by memory stalls that might extend a CPU cycle. Once there is a low-to-high transition on an external interrupt pin (cycle 1), it takes two clock cycles for the signal to reach the CPU boundary (cycle 3). When the interrupt signal enters the CPU, it is has been detected (cycle 4). Two clock cycles after detection, the interrupt's corresponding flag bit in the IFR is set (cycle 6).

In Figure 5–12, IFm is set during CPU cycle 6. You could attempt to clear bit IFm by using an **MVC** instruction to write a 1 to bit m of the ICR in execute packet n + 3 (during CPU cycle 4). However, in this case, the automated write by the interrupt detection logic takes precedence and IFm remains set.

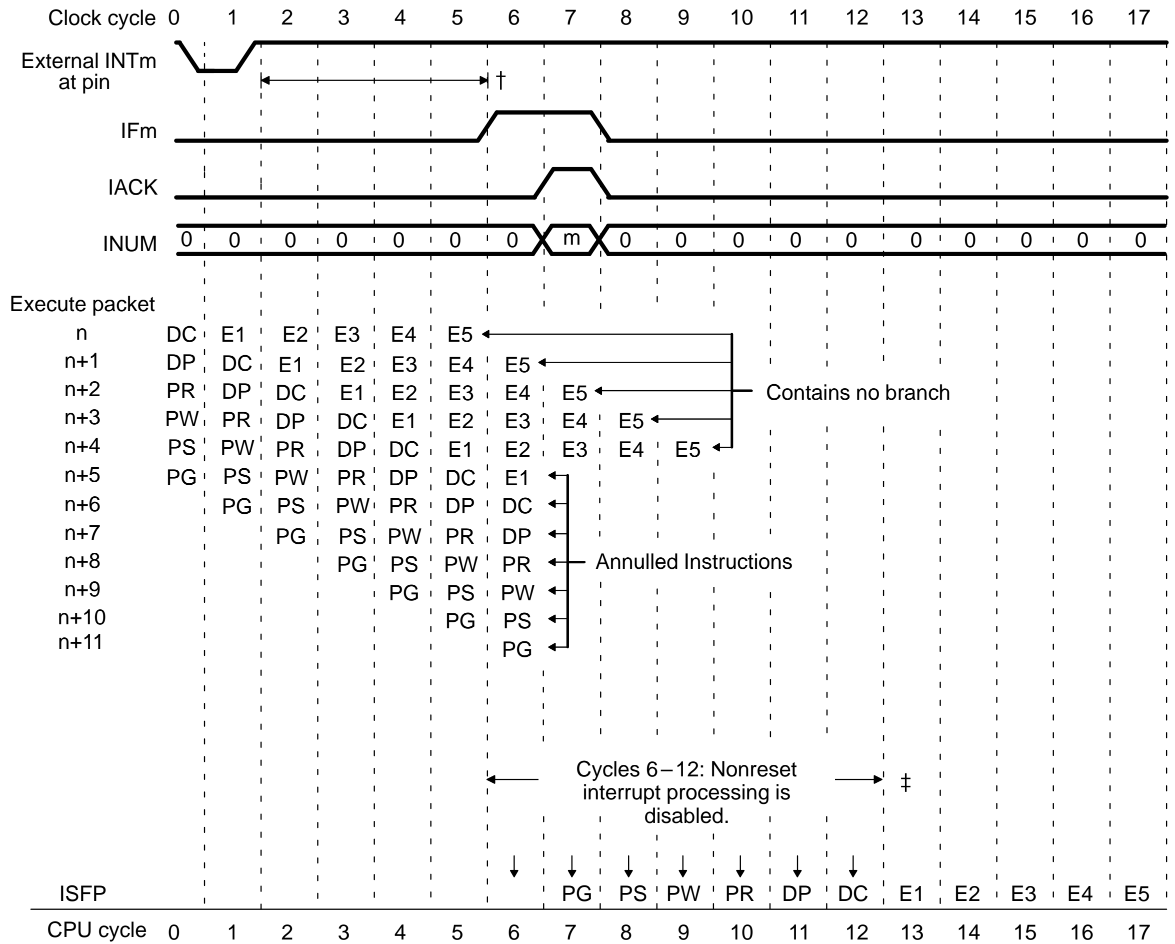
Figure 5–12 assumes INTm is the highest priority pending interrupt and is enabled by GIE and NMIE as necessary. If it is not, IFm remains set until either you clear it by writing a1 to bit m of the ICR, or the processing of INTm occurs.

5.4.2 Conditions for Processing an Interrupt – Nonreset

In clock cycle 4 of Figure 5–12, a nonreset interrupt in need of processing is detected. For this interrupt to be processed, the following conditions must be valid on the same clock cycle and are evaluated every clock cycle:

- IFm is set during CPU cycle 6. (This determination is made in CPU cycle 4 by the interrupt logic.)
- There is not another higher priority IFm bit set in the IFR.
- The corresponding bit in the IER is set (IEm = 1).
- GIE = 1
- NMIE = 1
- The five previous execute packets (n through n+4) do not contain a branch (even if the branch is not taken) and are not in the delay slots of a branch. Any pending interrupt will be taken as soon as pending branches are completed.

Figure 5–12. Interrupt Detection and Processing: Pipeline Operation–Nonreset



† IFm is set on the next CPU cycle boundary after a 4-clock cycle delay after the rising edge of INTm.

‡ After this point, interrupts are still disabled. All nonreset interrupts are disabled when NMIE = 0. All maskable interrupts are disabled when GIE = 0.

5.4.3 Actions Taken During Interrupt Processing – Nonreset

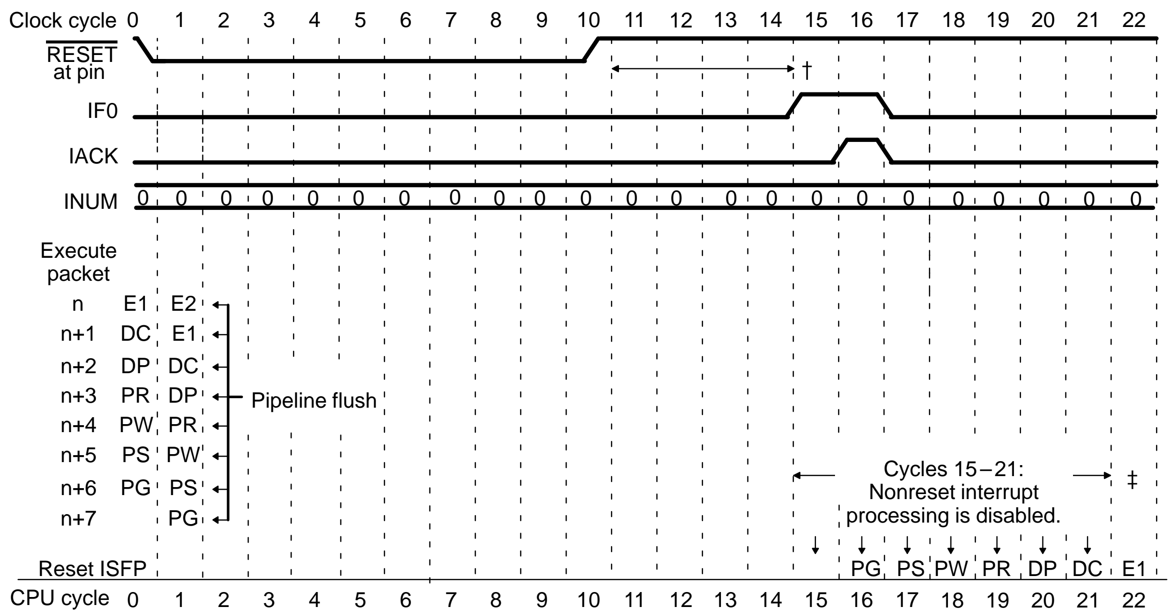
During CPU cycles 6–12 of Figure 5–12, the following interrupt processing actions occur:

- Processing of subsequent nonreset interrupts is disabled.
- For all interrupts except NMI, PGIE is set to the value of GIE and then GIE is cleared.
- For NMI, NMIE is cleared.
- The next execute packets (from n + 5 on) are annulled. If an execute packet is annulled during a particular pipeline stage, it does not modify any CPU state. Annulling also forces an instruction to be annulled in future pipeline stages.
- The address of the first annulled execute packet (n+5) is loaded in to the NRP (in the case of NMI) or IRP (for all other interrupts).
- A branch to the address held in ISTP (the pointer to the ISFP for INTm) is forced into the E1 phase of the pipeline during cycle 7.
- During cycle 7, IACK is asserted and the proper INUMx signals are asserted to indicate which interrupt is being processed. The timings for these signals in Figure 5–12 represent only the signals' characteristics inside the CPU. The external signals may be delayed and be longer in duration to handle external devices. Check the data sheet for your specific device for particular timing values.
- IFm is cleared during cycle 8.

5.4.4 Setting the Interrupt Flag – $\overline{\text{RESET}}$

$\overline{\text{RESET}}$ must be held low for a minimum of ten clock cycles. Four clock cycles after it goes high, processing of the reset vector begins. The flag for $\overline{\text{RESET}}$ (IF0) in the IFR is set by the low-to-high transition of the $\overline{\text{RESET}}$ signal on the CPU boundary. In Figure 5–13, IF0 is set during CPU cycle 15. This transition is detected on a clock-cycle by clock-cycle basis and is not affected by memory stalls that might extend a CPU cycle.

Figure 5–13. Interrupt Detection and Processing: Pipeline Operation–$\overline{\text{RESET}}$



† IF0 is set on the next CPU cycle boundary after a 4-clock cycle delay after the rising edge of $\overline{\text{RESET}}$.

‡ After this point, interrupts are still disabled. All nonreset interrupts are disabled when NMIE = 0. All maskable interrupts are disabled when GIE = 0.

5.4.5 Actions Taken During Interrupt Processing – $\overline{\text{RESET}}$

A low signal on the $\overline{\text{RESET}}$ pin is the only requirement to process a reset. Once $\overline{\text{RESET}}$ makes a high-to-low transition, the pipeline is flushed and CPU registers are returned to their reset values. GIE, NMIE, and the ISTB in the ISTP are cleared. For the CPU state after $\overline{\text{RESET}}$, see section 5.3.3.1. Four clock cycles after the subsequent low-to-high transition, the IF0 bit is set in the IFR.

During CPU cycles 15–21 of Figure 5–12, the following reset processing actions occur:

- ❑ Processing of subsequent nonreset interrupts is disabled because GIE and NMIE are cleared.
- ❑ A branch to the address held in ISTP (the pointer to the ISFP for INT0) is forced into the E1 phase of the pipeline during cycle 16.
- ❑ During cycle 16, IACK is asserted and the proper INUMx signals are asserted to indicate $\overline{\text{RESET}}$ is being processed.
- ❑ IF0 is cleared during cycle 17.

Note:

Code which starts running after reset must explicitly enable GIE, NMIE and IER to allow interrupts to be processed.

5.5 Performance Considerations

The interaction of the ’C62xx CPU and sources of interrupts present performance issues for you to consider when you are developing your code.

5.5.1 General Performance

- **Overhead**. Overhead for all CPU interrupts is 7 cycles. You can see this in Figure 5–12, where no new instructions are entering the E1 pipeline phase during CPU cycles 6 through 12 (cycles 15–21 for $\overline{\text{RESET}}$ in Figure 5–13).
- **Latency**. Interrupt latency is 11 cycles (21 cycles for $\overline{\text{RESET}}$). In Figure 5–12, although the interrupt is active in cycle 2, execution of interrupt service code does not begin until cycle 13.
- **Frequency**. The logic clears the nonreset interrupt (IFm) on cycle 8, with any incoming interrupt having highest priority. Thus, an interrupt can be recognized every second cycle. Also, because a low-to-high transition is necessary, an interrupt can occur only every second cycle. However, the frequency of interrupt processing depends on the time required for interrupt service and whether you reenable interrupts during processing, thereby allowing nested interrupts. Effectively only two occurrences of a specific interrupt can be recognized in two cycles.

5.5.2 Pipeline Interaction

Because the serial or parallel encoding of fetch packets does not affect the DC through E5 phases of the pipeline, no conflicts between code parallelism and interrupts exist. There are three operations or conditions that can affect, or are affected by, interrupts:

- **Branches.** Interrupts are delayed if any execute packets n through n + 4 in Figure 5–12 contain a branch or are in the delay slots of a branch.
- **Memory stalls.** Memory stalls delay interrupt processing, because they inherently extend CPU cycles.
- **Multicycle NOPs.** Multicycle **NOP**s (including **IDLE**) operate like other instructions when interrupted, except when an interrupt causes annulment of any but the first cycle of a multicycle **NOP**. In that case, the address of the *next* execute packet in the pipeline is saved in the NRP or the IRP. This prevents returning to an **IDLE** instruction or a multicycle **NOP** that was interrupted.

5.6 Programming Considerations

The interaction of the 'C62xx CPU and sources of interrupts present programming issues for you to consider when you are developing your code.

5.6.1 Single Assignment Programming

Example 5–10 shows code without single assignment and Example 5–11 shows code using the single assignment programming method.

To avoid unpredictable operation, you must employ the single assignment method to code that can be interrupted. When an interrupt occurs, all instructions entering E1 prior to the beginning of interrupt processing are allowed to complete execution (through E5). All other instructions are annulled and refetched upon return from interrupt. The instructions encountered after the return from the interrupt do not experience any delay slots from the instructions prior to processing the interrupt. Thus, instructions with delay slots prior to the interrupt can appear, to the instructions after the interrupt, to have fewer delay slots than they actually have.

For example, suppose that register A1 contained zero and register A0 pointed to a memory location containing a value of ten before reaching the code in Example 5–10. The ADD instruction, which is in a delay slot of the LDW, sums A2 with the value in A1 (zero) and the result in A3 is just a copy of A2. If an interrupt occurred between the LDW and ADD, the LDW would complete the update of A1 (ten), the interrupt would be processed, and the ADD would sum A1 (ten) with A2 and place the result in A3 (equal to A2 + ten). Obviously, this situation produces incorrect results.

In Example 5–11, the single assignment method is used. The register A1 is assigned only to the ADD input and not to the result of the LDW. Regardless of the value of A6 with or without an interrupt, A1 does not change before it is summed with A2. Result A3 is equal to A2.

Example 5–10. Code Without Single Assignment: Multiple Assignment of A1

```
LDW    .D1    *A0,A1
ADD    .L1    A1,A2,A3
NOP           3
MPY    .M1    A1,A4,A5  ; uses new A1
```

Example 5–11. Code Using Single Assignment

```
LDW    .D1    *A0,A6
ADD    .L1    A1,A2,A3
NOP           3
MPY    .M1    A6,A4,A5 ; uses A6
```

5.6.2 Nested Interrupts

Generally, when the CPU enters an interrupt service routine, interrupts are disabled. However, when the interrupt service routine is for one of the maskable interrupts (INT4–INT15), an NMI can interrupt processing of the maskable interrupt. In other words, an NMI can interrupt a maskable interrupt, but neither an NMI nor a maskable interrupt can interrupt an NMI.

There may be times when you want to allow an interrupt service routine to be interrupted by another (particularly higher priority) interrupt. Even though the processor by default does not allow interrupt service routines to be interrupted unless the source is an NMI, it is possible to nest interrupts under software control. The process requires you to save the original IRP (or NRP) and IER to memory or registers (either registers not used or saved if used by subsequent interrupts), and if you desire, to set up a new set of interrupt enables once the ISR is entered, and save the CSR. Then you could set the GIE bit, which would reenable interrupts inside the interrupt service routine.

5.6.3 Manual Interrupt Processing

You can poll the IFR and IER to detect interrupts manually and then branch to the value held in the ISTP as shown below in Example 5–12.

The code sequence begins by copying the address of the highest priority interrupt from the ISTP to the register B2. The next instruction extracts the number of the interrupt, which is used later to clear the interrupt. The branch to the interrupt service routine comes next with a parallel instruction to set up the ICR word.

The last five instructions fill the delay slots of the branch. First, the 32-bit return address is stored in the B2 register and then copied to the interrupt return pointer (IRP). Finally, the number of the highest priority interrupt, stored in B1, is used to shift the ICR word in B1 to clear the interrupt.

Example 5–12. Manual Interrupt Processing

```
            MVC     .S2     ISTP,B2        ; get related ISFP address
            EXTU    .S2     B2,23,27,B1    ; extract HPEINT
     [B1]   B       .S2     B2             ; branch to interrupt
||   [B1]   MVK     .S1     1,A0           ; setup ICR word
     [B1]   MVK     .S2     RET_ADR,B2     ; create return address
     [B1]   MVKH    .S2     RET_ADR,B2     ;
     [B1]   MVC     .S2     B2,IRP         ; save return address
     [B1]   SHL     .S2     A0,B1,B1       ; create ICR word
     [B1]   MVC     .S2     B1,ICR         ; clear interrupt flag
     RET_ADR:       (Post interrupt service routine Code)
```

5.6.4 Traps

A trap behaves like an interrupt, but is created and controlled with software. The trap condition can be stored in any one of the conditional registers: A1, A2, B0, B1, or B2. If the trap condition is valid, a branch to the trap handler routine processes the trap and the return.

Example 5–13 and Example 5–14 show a trap call and the return code sequence, respectively. In the first code sequence, the address of the trap handler code is loaded into register B0 and the branch is called. In the delay slots of the branch, the context is saved in the B0 register, the GIE bit is cleared to disable maskable interrupts, and the return pointer is stored in the B1 register. If the trap handler were within the 21-bit offset for a branch using a displacement, the MVKH instructions could be eliminated, thus shortening the code sequence.

The trap is processed with the code located at the address pointed to by the label TRAP_HANDLER. If the B0 or B1 registers are needed in the trap handler, their contents must be stored to memory and restored before returning. The code shown in Example 5–14 should be included at the end of the trap handler code to restore the context prior to the trap and return to the TRAP_RETURN address.

Example 5–13. Code Sequence to Invoke a Trap

```
[A1]   MVK    .S2    TRAP_HANDLER,B0    ; load 32-bit trap address
[A1]   MVKH   .S2    TRAP_HANDLER,B0
[A1]   B      .S2    B0                 ; branch to trap handler
[A1]   MVC    .S2    CSR,B0             ; read CSR
[A1]   AND    .S2    -2,B0,B1           ; disable interrupts: GIE = 0
[A1]   MVC    .S2    B1,CSR             ; write to CSR
[A1]   MVK    .S2    TRAP_RETURN,B1     ; load 32-bit return address
[A1]   MVKH   .S2    TRAP_RETURN,B1
TRAP_RETURN:      (post-trap code)
```

Note: A1 contains the trap condition.

Example 5–14. Code Sequence for Trap Return

```
B      .S2    B1          ; return
MVC    .S2    B0,CSR      ; restore CSR
NOP           4           ; delay slots
```

Appendix A

Glossary

A

address: The location of a word in memory.

addressing mode: The method by which an instruction calculates the location of an object in memory.

ALU: *arithmetic logic unit*. The part of the CPU that performs arithmetic and logic operations.

annul: Any instruction that is annulled does not complete its pipeline stages.

B

bootloader: A built-in segment of code that transfers code from an external source to program memory at power-up.

C

clock cycles: Cycles based on the input from the external clock.

code: A set of instructions written to perform a task; a computer program or part of a program.

CPU cycle: The period during which a particular execute packet is in a particular pipeline stage. CPU cycle boundaries always occur on clock cycle boundaries; however, memory stalls can cause CPU cycles to extend over multiple clock cycles.

D

data memory: A memory region used for storing and manipulating data.

delay slot: A CPU cycle that occurs after the first execution phase (E1) of an instruction in which results from the instruction are not available.

E

execute packet: A block of instructions that execute in parallel.

external interrupt: A hardware interrupt triggered by a specific value on a pin.

F

fetch packet: A block of program data containing up to eight instructions.

G

global interrupt enable (GIE): A bit in the control status register (CSR) used to enable or disable maskable interrupts.

H

hardware interrupt: An interrupt triggered through physical connections with on-chip peripherals or external devices.

I

interrupt: A condition causing program flow to be redirected to a location in the interrupt service table (IST).

interrupt service fetch packet (ISFP): A fetch packet used to service interrupts. If eight instructions are insufficient, the user must branch out of this block for additional interrupt service. If the delay slots of the branch do not reside within the ISFP, execution continues from execute packets in the next fetch packet (the next ISFP).

interrupt service table (IST): 16 contiguous ISFPs, each corresponding to a condition in the interrupt flag register (IFR). The IST resides in memory accessible by the program memory system. The IST must be aligned on a 256-word boundary (32 fetch packets x 8 words/fetch packet). Although only 16 interrupts are defined, space in the IST is reserved for 32 for future expansion. The IST's location is determined by the interrupt service table pointer (ISTP) register.

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 interrupt: A hardware interrupt that can be enabled or disabled through software.

memory stall: When the CPU is stalled waiting on a memory load or store.

MSB: *most significant bit*. The highest-order bit in a word.

N

nested interrupt: A higher-priority interrupt that must be serviced before completion of the current interrupt service routine.

nonmaskable interrupt: An interrupt that can be neither masked nor disabled.

O

overflow: A condition in which the result of an arithmetic operation exceeds the capacity of the register used to hold that result.

P

pipeline: A method of executing instructions in an assembly-line fashion.

program memory: A memory region used for storing and executing programs.

R

register: A group of bits used for holding data or for controlling or specifying the status of a device.

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.

S

shifter: A hardware unit that shifts bits in a word to the left or to the right.

sign extension: An operation that fills the high order bits of a number with the sign bit.

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 for every wait state.

Z

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.

Index

.D functional units 2-3 to 2-6

.L functional units 2-3 to 2-6

.M functional units 2-3 to 2-6

.S functional units 2-3 to 2-6

1X and 2X paths, conflicts 2-5, 3-14

40-bit data 2-2 to 2-4
  conflicts 3-15

A

ABS instruction 3-25

ADD(U) instruction 3-26 to 3-29

ADD2 instruction 3-32

ADDA using circular addressing 3-19

ADDA(B)(H)(W) instruction 3-29 to 3-31

ADDK instruction 3-31

address paths 2-5

addressing mode register
  field encoding 2-7
  figure 2-7

addressing modes
  circular mode 3-18
  linear mode 3-18

AMR
  field encoding 2-7
  figure 2-7

AND instruction 3-33

annulled execute packet 5-20

applications
  TMS320 1-3
  TMS320 family 1-2

architecture 1-6

assembler conflict detectability for writes 3-17

B

B instruction
  using a displacement 3-34 to 3-36
  using a register 3-36 to 3-38

B IRP instruction 3-38, 5-6, 5-11, 5-17

B NRP instruction 3-40, 5-13, 5-16

block size calculations 2-8

branch execution, block diagram 4-16

branch instruction
  using a displacement 3-34 to 3-36
  using a register 3-36 to 3-38

branch instruction phases, figure 4-15

branch instruction types 4-15

branching
  and multicycle NOPs 4-20
  to additional interrupt service routine 5-7
  to the middle of an execute packet 3-12

C

circular addressing 2-7
  block size calculations 2-8

circular addressing mode 3-18

clearing an individual interrupt 5-15

clearing interrupts 5-14

clock cycle 4-9

CLR instruction 3-42 to 3-44

CMPEQ instruction 3-44 to 3-46

CMPGT(U) instruction 3-46 to 3-49

CMPLT(U) instruction 3-49 to 3-52

code, definition A-1

conditional operations 3-13

conditional registers 3-13

conflict detectability in write operations 3-17

control
  individual interrupts 5-13
  of interrupts 5-11
control register, register addresses for accessing 3-75
control register file 2-6
control registers
  interrupt 5-10
  table 2-6
control status register (CSR) 2-9
  fields, table 2-9
  figure 2-9, 5-11
CPU 1-7 to 1-9
  control register file 2-6
  data paths 2-1 to 2-10
  functional units 2-3
  general-purpose register files 2-2
  load and store paths 2-5
CPU cycle 4-9, 4-10
CPU data paths, figure 2-4
CPU execution, block diagram 4-5
cross path conflicts 3-14
cross paths 2-5
CSR 2-6, 2-9, 5-10, 5-11
  fields, table 2-9
  figure 2-9

D

.D functional units 2-3 to 2-6
data address paths 2-5
data address pointer 4-14
data and program memory stalls 4-22
data load accesses, versus program memory accesses 4-21
data paths 2-1, 2-2
  block diagram 2-4
DC pipeline phase 4-4
decoding instructions 4-4
delay slot summary, table 3-9
delay slots 3-9, 4-10
  stores 4-15
detection of interrupts 5-18
disabling an individual interrupt, example 5-14
disabling maskable interrupts globally, example 5-12
DP pipeline phase 4-4, 4-18

E

E1–E5 pipeline phases 4-5
enabling an individual interrupt, example 5-14
enabling maskable interrupts globally, example 5-12
execute packets 3-10, 4-17
  multicycle NOPs in 4-19
execute phases of the pipeline 4-21
  figure 4-5
execution notations 3-2 to 3-4
EXT instruction 3-52 to 3-55
EXTU instruction 3-55 to 3-58

F

fetch packet (FP) 3-10, 4-17, 5-6
fetch phases of the pipeline 4-21
  figure 4-3
fetching instructions 4-3
flag, interrupt 5-18, 5-21
functional unit to instruction mapping 3-5 to 3-7
functional units 2-3

G

general-purpose register files
  cross paths 2-5
  data address paths 2-5
  memory, load, and store paths 2-5
general-purpose registers, constraints on 3-16 to 3-18
GIE bit 5-4, 5-11, 5-18, 5-20
globally controlling interrupts, enabling and disabling 5-11

H

HPEINT 5-8

I

IACK signal 5-4, 5-20, 5-22

ICR. *See* interrupt clear register

IDLE instruction 3-58

IER. *See* interrupt enable register

IFm bits 5-20, 5-22

IFR. *See* interrupt flag register

instruction descriptions 3-21

individual interrupt control 5-13

instruction operation, notations for 3-2

instruction types
  branch instructions 4-15
  execution phases 4-10
  load instructions 4-14
  multiply instructions 4-11 to 4-13
  pipeline execution 4-10
  single-cycle 4-11
  store instructions 4-12 to 4-14

instruction to functional unit mapping 3-4

instructions, descriptions, example 3-22 to 3-25

INT4–INT15 interrupt signals 5-4

interleaved memory bank scheme 4-23
  four-bank memory 4-23
  with two memory spaces 4-24

interrupt clear register (ICR) 5-13 to 5-15
  writing to 5-14

interrupt control 5-11
  individual 5-13

interrupt control registers 5-10
  table 5-10

interrupt detection and processing 5-18 to 5-21
  figure 5-19, 5-21

interrupt enable register (IER) 5-4, 5-10, 5-13, 5-18

interrupt flag, setting 5-18, 5-21

interrupt flag register (IFR) 5-2, 5-4, 5-10, 5-13, 5-14
  writing to 5-14
  figure 5-14

interrupt performance 5-23
  frequency 5-23
  latency 5-23
  overhead 5-23

interrupt pipeline interaction
  branching 5-23
  code parallelism 5-23
  memory stalls 5-23
  multicycle NOPs 5-23

interrupt priorities 5-3

interrupt processing, actions taken during 5-20, 5-22

interrupt processing, manual 5-25

interrupt return pointer (IRP), 5-10, 5-17
  figure 5-17

interrupt service fetch packet 5-6

interrupt service table
  relocation of 5-9
  table 5-5

interrupt service table pointer (ISTP) 5-8, 5-10, 5-20, 5-22, 5-25

interrupt set register (ISR) 5-10, 5-15

interrupts
  branching 5-20, 5-22
  clearing 5-14
  manual processing 5-25
  nesting 5-25
  overview 5-2
  programming considerations 5-24
  setting 5-14
  signals used 5-2
  traps 5-26
  types of 5-2

introduction 1-1 to 1-8

INUM3–INUM0 signals 5-4, 5-20, 5-22

invoking a trap 5-26

IRP register 5-10, 5-17

ISFP. *See* interrupt service fetch packet

ISR. *See* interrupt set register

IST. *See* interrupt service table 5-5

ISTB 5-8, 5-9

ISTP. *See* interrupt service table pointer

L

.L functional units 2-3 to 2-6

latency 3-9

LD(B/BU)(H/HU)(W) instruction
  15-bit constant offset 3-63 to 3-65
  5-bit unsigned constant offset or register offset 3-59 to 3-63

linear addressing mode 3-18
LMBD instruction 3-65 to 3-67
load address generation, syntax 3-20
load and store paths 2-5
load conflicts 3-15
load execution, block diagram 4-14
load from memory banks, example 4-23
load instruction phases, figure 4-14
load instruction types 4-14
load or store to the same memory location, rules 4-13
load paths 2-5
loads, and memory banks 4-23
loads with circular addressing 3-18
long (40-bit) data
  conflicts 3-15
  register pairs 2-2 to 2-4

M

.M functional units 2-3 to 2-6
mapping
  functional unit to instruction 3-5 to 3-7
  instruction to functional unit 3-4
maskable interrupt
  description 5-4
  return from 5-17
maskable interrupts 5-4
memory
  considerations 4-21
  internal 1-8
memory accesses, pipeline phases used during 4-21
memory bank hits 4-23
memory paths 2-5
memory stalls 4-22
MPY(U/US/SU) instruction 3-67
MPYH instruction 3-69
MPYHL instruction 3-71
MPYLH instruction 3-72
multicycle NOPs 4-18
multicycle NOPs, in execute packets, figure 4-19
multiply execution diagram 4-12
multiply instruction phases, figure 4-11
multiply instruction types 4-11 to 4-13
MV instruction 3-73
MVC instruction 3-74 to 3-77, 5-18, 5-21
  writing to IFR or ICR 5-14
MVK instruction 3-77 to 3-79
MVK(H)(LH) instruction 3-79 to 3-81

N

NEG instruction 3-81
nesting interrupts 5-25
NMI. *See* nonmaskable interrupt
NMI return pointer (NRP), figure 5-16
NMIE bit 5-4, 5-13, 5-18
nonmaskable interrupt (NMI) 5-3, 5-20, 5-25
  return from, example 5-16
nonmaskable interrupt return pointer (NRP) 5-10, 5-16
NOP instruction 3-82 to 3-84, 4-4, 5-6
NORM instruction 3-84 to 3-86
NOT instruction 3-86
notations for instructions 3-2 to 3-4
NRP 5-10
NRP register 5-16

O

opcode map 3-7
operands, examples 3-22 to 3-24
OR instruction 3-87
overview, TMS320 family 1-2

P

parallel code, example 3-12
parallel fetch packets 3-11
parallel operations 3-10
partially serial fetch packets 3-12
p-bit 3-10
PCE1 3-34
performance considerations, pipeline 4-17
peripherals 1-8
PFC. *See* program fetch counter
PG pipeline phase 4-3
PGIE bit 5-11, 5-20

pipeline
  decode stage 4-4
  execute stage 4-5
  factors that provide flexibility 4-1
  fetch stage 4-3
  operation overview 4-2
  performance considerations 4-17
  phases 4-6
  stages 4-2
pipeline execution 4-10
pipeline operation
  fetch packets with different numbers of execute packets, figure 4-18
  multiple execute packets in a fetch packet 4-17
  one execute packet per fetch packet 4-6
  summary 4-6 to 4-10
pipeline phases
  functional block diagram 4-8
  operations occurring during, table 4-7
  used during memory accesses 4-21
polling the IFR and IER 5-25
PR pipeline phase 4-3
program and data memory stalls 4-22
program counter 3-34
program fetch counter 3-34
program memory accesses, versus data load accesses 4-21
PS pipeline phase 4-3
push, definition A-3
PW pipeline phase 4-3

R

reading the IFR 5-15
register files
  cross paths 2-5
  data address paths 2-5
  general-purpose 2-2
  memory, load, and store paths 2-5
register read constraints 3-16
register storage scheme, 40-bit data, figure 2-3
register write conflicts 3-16 to 3-18
relocation of the interrupt service table (IST), 5-9
$\overline{\text{RESET}}$, CPU state after 5-16
reset interrupt 5-3
$\overline{\text{RESET}}$ signal 5-3
resource constraints 3-14
returning from a trap 5-26
returning from interrupt servicing 5-16
returning from maskable interrupts 5-17
returning from NMI 5-16

S

.S functional units 2-3 to 2-6
SADD instruction 3-88 to 3-90
SAT instruction 3-90 to 3-92
serial fetch packets 3-11
SET instruction 3-92 to 3-94
setting an individual interrupt, example 5-15
setting interrupts 5-14
setting the interrupt flag 5-18, 5-21
SHL instruction 3-94 to 3-96
SHR instruction 3-96 to 3-98
SHRU instruction 3-98
single assignment 5-24
single-cycle instruction execution, block diagram 4-11
single-cycle instructions 4-11
SMPY(L)(H) instruction 3-99 to 3-101
SSHL instruction 3-101 to 3-103
SSUB instruction 3-103 to 3-105
ST(B)(H)(W) instruction 15-bit offset 3-108 to 3-110
ST(B/BU)(H/HU)(W), register offset or 5-bit unsigned constant offset 3-105 to 3-108
store address generation, syntax 3-20
store conflicts 3-15
store execution diagram 4-13
store instruction phases, figure 4-12
store instruction types 4-12 to 4-14
store or load to the same memory location, rules 4-13
store paths 2-5
stores 3-106
stores with circular addressing 3-18
SUB(U) instruction 3-110 to 3-113
SUB2 instruction 3-116
SUBA using circular addressing 3-19
SUBA(B)(H)(W) instruction 3-113 to 3-115
SUBC instruction 3-115

T

TMS320 family 1-2 to 1-6
- advantages 1-2
- applications 1-2 to 1-3
- history 1-2
- overview 1-2

TMS320C62xx devices
- features 1-4
- options 1-4 to 1-6
- block diagram 1-6
- performance 1-4

traps 5-26

X

XOR instruction 3-117

Z

ZERO instruction 3-118