TMS320C6000 CPU and Instruction Set Reference Guide

Literature Number: SPRU189D
March 1999

Printed on Recycled Paper

IMPORTANT NOTICE

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

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

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

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

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

Copyright © 1999, Texas Instruments Incorporated

Preface

Read This First

About This Manual

This reference guide describes the CPU architecture, pipeline, instruction set, and interrupts for the TMS320C6000 digital signal processors (DSPs). Unless otherwise specified, all references to the ’C6000 refer to the TMS320C6000 platform of DSPs, ’C62x refers to the TMS320C62x fixed-point DSPs in the ’C6000 platform, and ’C67x refers to the TMS320C67x floating-point DSPs in the ’C6000 platform.

How to Use This Manual

Use this manual as a reference for the architecture of the TMS320C6000 CPU. First-time readers should read Chapter 1 for general information about TI DSPs, the features of the ’C6000, and the applications for which the ’C6000 is best suited.

Read chapters 2, 5, 6, and 7 to grasp the concepts of the architecture. Chapter 3 and Chapter 4 contain detailed information about each instruction and is best used as reference material; however, you may want to read sections 3.1 through 3.9 and sections 4.1 through 4.6 for general information about the instruction set and to understand the instruction descriptions, then browse through Chapter 3 and Chapter 4 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, *TMS320C62x/C67x Fixed-Point Instruction Set* Chapter 4, *TMS320C67x Floating-Point Instruction Set*
Conditional operations	Chapter 3, *TMS320C62x/C67x Fixed-Point Instruction Set* Chapter 4, *TMS320C67x Floating-Point Instruction Set*
Control registers	Chapter 2, *CPU Data Paths and Control*
CPU architecture and data paths	Chapter 2, *CPU Data Paths and Control*
Delay slots	Chapter 3, *TMS320C62x/C67x Fixed-Point Instruction Set* Chapter 4, *TMS320C67x Floating-Point Instruction Set* Chapter 5, *TMS320C62x Pipeline* Chapter 6, *TMS320C67x Pipeline*
General-purpose register files	Chapter 2, *CPU Data Paths and Control*
Instruction set	Chapter 3, *TMS320C62x/C67x Fixed-Point Instruction Set* Chapter 4, *TMS320C67x Floating-Point Instruction Set*
Interrupts and control registers	Chapter 7, *Interrupts*
Parallel operations	Chapter 3, *TMS320C62x/C67x Fixed-Point Instruction Set* Chapter 4, *TMS320C67x Floating-Point Instruction Set*
Pipeline phases and operation	Chapter 5, *TMS320C62x Pipeline* Chapter 6, *TMS320C67x Pipeline*
Reset	Chapter 7, *Interrupts*

If you are interested in topics that are not listed here, check *Related Documentation From Texas Instruments*, on page vi, for brief descriptions of other 'C6x-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, *TMS320C62x/C67x Fixed-Point 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.

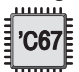
- Square brackets, [and], and parentheses, (and), are used to identify optional items. If you use an optional item, you must specify the information within brackets or parentheses; however, you do not enter the brackets or parentheses themselves. Here is an example of an instruction that has optional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

- A special icon is used to indicate material that applies only to the floating-point (’C67x) DSP:



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.

TMS320C62x/C67x Technical Brief (literature number SPRU197) gives an introduction to the ’C62x/C67x digital signal processors, development tools, and third-party support.

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

TMs320C6202 Digital Signal Processor Data Sheet (literature number SPRS072) describes the features of the TMS320C6202 fixed-point DSP and provides pinouts, electrical specifications, and timings for the device.

TMS320C6211 Digital Signal Processor Data Sheet (literature number SPRS073) describes the features of the TMS320C6211 fixed-point DSP and provides pinouts, electrical specifications, and timings for the device.

TMS320C6701 Digital Signal Processor Data Sheet (literature number SPRS067) describes the features of the TMS320C6701 floating-point DSP and provides pinouts, electrical specifications, and timings for the device.

TMS320C6000 Peripherals Reference Guide (literature number SPRU190) describes common peripherals available on the TMS320C6000 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.

TMS320C62x/C67x Programmer’s Guide (literature number SPRU198) describes ways to optimize C and assembly code for the TMS320C62x/C67x DSPs and includes application program examples.

TMS320C6000 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 ’C6000 generation of devices.

TMS320C6000 Optimizing C Compiler User's Guide (literature number SPRU187) describes the 'C6000 C compiler and the assembly optimizer. This C compiler accepts ANSI standard C source code and produces assembly language source code for the 'C6000 generation of devices. The assembly optimizer helps you optimize your assembly code.

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.

Trademarks

TI, 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: dsph@ti.com

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

Contents

Summarizes the features of the TMS320 family of products and presents typical applications. Describes the TMS320C62x/C67x DSPs and lists their key features.

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 TMS320C62x/C67x . . . 1-5
1.4 TMS320C62x/C67x Architecture . . . 1-7
1.4.1 Central Processing Unit (CPU) . . . 1-8
1.4.2 Internal Memory . . . 1-8
1.4.3 Peripherals . . . 1-9

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

Summarizes the TMS320C62x/C67x architecture and describes the primary components of the CPU.

2.1 General-Purpose Register Files . . . 2-4
2.2 Functional Units . . . 2-6
2.3 Register File Cross Paths . . . 2-7
2.4 Memory, Load, and Store Paths . . . 2-7
2.5 Data Address Paths . . . 2-7
2.6 TMS320C62x/C67x Control Register File . . . 2-8
2.6.1 Addressing Mode Register (AMR) . . . 2-9
2.6.2 Control Status Register (CSR) . . . 2-11
2.6.3 E1 Phase Program Counter (PCE1) . . . 2-12
2.7 TMS320C67x Extensions to the Control Register File . . . 2-13
2.7.1 Floating-Point Adder Configuration Register (FADCR) . . . 2-14
2.7.2 Floating-Point Auxiliary Configuration Register (FAUCR) . . . 2-16
2.7.3 Floating-Point Multiplier Configuration Register (FMCR) . . . 2-18

3 TMS320C62x/C67x Fixed-Point Instruction Set 3-1

Describes the assembly language instructions that are common to both the TMS320C62x and TMS320C67x, 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 TMS320C62x/C67x Opcode Map 3-9
3.4 Delay Slots 3-12
3.5 Parallel Operations 3-13
3.5.1 Example Parallel Code 3-15
3.5.2 Branching Into the Middle of an Execute Packet 3-15
3.6 Conditional Operations 3-16
3.7 Resource Constraints 3-17
3.7.1 Constraints on Instructions Using the Same Functional Unit 3-17
3.7.2 Constraints on Cross Paths (1X and 2X) 3-17
3.7.3 Constraints on Loads and Stores 3-18
3.7.4 Constraints on Long (40-Bit) Data 3-18
3.7.5 Constraints on Register Reads 3-19
3.7.6 Constraints on Register Writes 3-19
3.8 Addressing Modes 3-21
3.8.1 Linear Addressing Mode 3-21
3.8.2 Circular Addressing Mode 3-21
3.8.3 Syntax for Load/Store Address Generation 3-23
3.9 Individual Instruction Descriptions 3-24

4 TMS320C67x Floating-Point Instruction Set 4-1

Describes the TMS320C67x floating-point instruction set, including examples of each instruction. Provides information about addressing modes and resource constraints.

4.1 Instruction Operation and Execution Notations 4-2
4.2 Mapping Between Instructions and Functional Units 4-4
4.3 Overview of IEEE Standard Single- and Double-Precision Formats 4-6
4.4 Delay Slots 4-11
4.5 TMS320C67x Instruction Constraints 4-12
4.6 Individual Instruction Descriptions 4-15

5 TMS320C62x Pipeline 5-1

Describes phases, operation, and discontinuities for the TMS320C62x CPU pipeline.

5.1 Pipeline Operation Overview 5-2
5.1.1 Fetch 5-2
5.1.2 Decode 5-4
5.1.3 Execute 5-5
5.1.4 Summary of Pipeline Operation 5-6
5.2 Pipeline Execution of Instruction Types 5-11
5.2.1 Single-Cycle Instructions 5-12

5.2.2 Multiply Instructions 5-12
5.2.3 Store Instructions 5-13
5.2.4 Load Instructions 5-15
5.2.5 Branch Instructions 5-16
5.3 Performance Considerations 5-18
5.3.1 Pipeline Operation With Multiple Execute Packets in a Fetch Packet 5-18
5.3.2 Multicycle NOPs 5-20
5.3.3 Memory Considerations 5-22

6 TMS320C67x Pipeline 6-1

Describes phases, operation, and discontinuities for the TMS320C67x CPU pipeline.

6.1 Pipeline Operation Overview 6-2
6.1.1 Fetch 6-2
6.1.2 Decode 6-4
6.1.3 Execute 6-5
6.1.4 Summary of Pipeline Operation 6-6
6.2 Pipeline Execution of Instruction Types 6-13
6.3 Functional Unit Hazards 6-20
6.3.1 .S-Unit Hazards 6-21
6.3.2 .M-Unit Hazards 6-25
6.3.3 .L-Unit Hazards 6-30
6.3.4 D-Unit Instruction Hazards 6-34
6.3.5 Single-Cycle Instructions 6-38
6.3.6 16 × 16-Bit Multiply Instructions 6-39
6.3.7 Store Instructions 6-40
6.3.8 Load Instructions 6-42
6.3.9 Branch Instructions 6-44
6.3.10 2-Cycle DP Instructions 6-46
6.3.11 4-Cycle Instructions 6-47
6.3.12 INTDP Instruction 6-47
6.3.13 DP Compare Instructions 6-48
6.3.14 ADDDP/SUBDP Instructions 6-49
6.3.15 MPYI Instructions 6-50
6.3.16 MPYID Instructions 6-50
6.3.17 MPYDP Instructions 6-51
6.4 Performance Considerations 6-52
6.4.1 Pipeline Operation With Multiple Execute Packets in a Fetch Packet 6-52
6.4.2 Multicycle NOPs 6-54
6.4.3 Memory Considerations 6-56

7 Interrupts 7-1

Describes the TMS320C62x/C67x interrupts, including reset and nonmaskable interrupts (NMI), and explains interrupt control, detection, and processing.

7.1 Overview of Interrupts 7-2
7.1.1 Types of Interrupts and Signals Used 7-2
7.1.2 Interrupt Service Table (IST) 7-5
7.1.3 Summary of Interrupt Control Registers 7-10
7.2 Globally Enabling and Disabling Interrupts (Control Status Register–CSR) 7-11
7.3 Individual Interrupt Control 7-13
7.3.1 Enabling and Disabling Interrupts (Interrupt Enable Register–IER) 7-13
7.3.2 Status of, Setting, and Clearing Interrupts (Interrupt Flag, Set, and Clear Registers–IFR, ISR, ICR) 7-14
7.3.3 Returning From Interrupt Servicing 7-16
7.4 Interrupt Detection and Processing 7-18
7.4.1 Setting the Nonreset Interrupt Flag 7-18
7.4.2 Conditions for Processing a Nonreset Interrupt 7-18
7.4.3 Actions Taken During Nonreset Interrupt Processing 7-21
7.4.4 Setting the $\overline{\text{RESET}}$ Interrupt Flag for the TMS320C62x/C67x 7-22
7.4.5 Actions Taken During $\overline{\text{RESET}}$ Interrupt Processing 7-23
7.5 Performance Considerations 7-24
7.5.1 General Performance 7-24
7.5.2 Pipeline Interaction 7-24
7.6 Programming Considerations 7-25
7.6.1 Single Assignment Programming 7-25
7.6.2 Nested Interrupts 7-26
7.6.3 Manual Interrupt Processing 7-26
7.6.4 Traps 7-27

A Glossary A-1

Defines terms and abbreviations used throughout this book.

Figures

1–1 TMS320C62x/C67x Block Diagram . . . 1-7
2–1 TMS320C62x CPU Data Paths . . . 2-2
2–2 TMS320C67x CPU Data Paths . . . 2-3
2–3 Storage Scheme for 40-Bit Data in a Register Pair . . . 2-5
2–4 Addressing Mode Register (AMR) . . . 2-9
2–5 Control Status Register (CSR) . . . 2-11
2–6 E1 Phase Program Counter (PCE1) . . . 2-12
2–7 Floating-Point Adder Configuration Register (FADCR) . . . 2-14
2–8 Floating-Point Auxiliary Configuration Register (FAUCR) . . . 2-16
2–9 Floating-Point Multiplier Configuration Register (FMCR) . . . 2-18
3–1 TMS320C62x/C67x Opcode Map . . . 3-10
3–2 Basic Format of a Fetch Packet . . . 3-13
3–3 Examples of the Detectability of Write Conflicts by the Assembler . . . 3-20
4–1 Single-Precision Floating-Point Fields . . . 4-8
4–2 Double-Precision Floating-Point Fields . . . 4-9
5–1 Fixed-Point Pipeline Stages . . . 5-2
5–2 Fetch Phases of the Pipeline . . . 5-3
5–3 Decode Phases of the Pipeline . . . 5-4
5–4 Execute Phases of the Pipeline and Functional Block Diagram of the TMS320C62x . . . 5-5
5–5 Fixed-Point Pipeline Phases . . . 5-6
5–6 Pipeline Operation: One Execute Packet per Fetch Packet . . . 5-6
5–7 Functional Block Diagram of TMS320C62x Based on Pipeline Phases . . . 5-8
5–8 Single-Cycle Instruction Phases . . . 5-12
5–9 Single-Cycle Execution Block Diagram . . . 5-12
5–10 Multiply Instruction Phases . . . 5-12
5–11 Multiply Execution Block Diagram . . . 5-13
5–12 Store Instruction Phases . . . 5-13
5–13 Store Execution Block Diagram . . . 5-14
5–14 Load Instruction Phases . . . 5-15
5–15 Load Execution Block Diagram . . . 5-15
5–16 Branch Instruction Phases . . . 5-16
5–17 Branch Execution Block Diagram . . . 5-17
5–18 Pipeline Operation: Fetch Packets With Different Numbers of Execute Packets . . . 5-19
5–19 Multicycle NOP in an Execute Packet . . . 5-20
5–20 Branching and Multicycle NOPs . . . 5-21

5–21 Pipeline Phases Used During Memory Accesses 5-22
5–22 Program and Data Memory Stalls 5-23
5–23 4-Bank Interleaved Memory 5-24
5–24 4-Bank Interleaved Memory With Two Memory Spaces 5-25
6–1 Floating-Point Pipeline Stages 6-2
6–2 Fetch Phases of the Pipeline 6-3
6–3 Decode Phases of the Pipeline 6-4
6–4 Execute Phases of the Pipeline and Functional Block Diagram of the TMS320C67x 6-5
6–5 Floating-Point Pipeline Phases 6-6
6–6 Pipeline Operation: One Execute Packet per Fetch Packet 6-6
6–7 Functional Block Diagram of TMS320C67x Based on Pipeline Phases 6-10
6–8 Single-Cycle Instruction Phases 6-38
6–9 Single-Cycle Execution Block Diagram 6-38
6–10 Multiply Instruction Phases 6-39
6–11 Multiply Execution Block Diagram 6-39
6–12 Store Instruction Phases 6-40
6–13 Store Execution Block Diagram 6-41
6–14 Load Instruction Phases 6-42
6–15 Load Execution Block Diagram 6-43
6–16 Branch Instruction Phases 6-44
6–17 Branch Execution Block Diagram 6-45
6–18 2-Cycle DP Instruction Phases 6-46
6–19 4-Cycle Instruction Phases 6-47
6–20 INTDP Instruction Phases 6-48
6–21 DP Compare Instruction Phases 6-48
6–22 ADDDP/SUBDP Instruction Phases 6-49
6–23 MPYI Instruction Phases 6-50
6–24 MPYID Instruction Phases 6-51
6–25 MPYDP Instruction Phases 6-51
6–26 Pipeline Operation: Fetch Packets With Different Numbers of Execute Packets 6-53
6–27 Multicycle NOP in an Execute Packet 6-54
6–28 Branching and Multicycle NOPs 6-55
6–29 Pipeline Phases Used During Memory Accesses 6-56
6–30 Program and Data Memory Stalls 6-57
6–31 8-Bank Interleaved Memory 6-58
6–32 8-Bank Interleaved Memory With Two Memory Spaces 6-59
7–1 Interrupt Service Table 7-5
7–2 Interrupt Service Fetch Packet 7-6
7–3 IST With Branch to Additional Interrupt Service Code Located Outside the IST 7-7
7–4 Interrupt Service Table Pointer (ISTP) 7-8
7–5 Control Status Register (CSR) 7-11
7–6 Interrupt Enable Register (IER) 7-13
7–7 Interrupt Flag Register (IFR) 7-14

7–8 Interrupt Set Register (ISR) 7-15
7–9 Interrupt Clear Register (ICR) 7-15
7–10 NMI Return Pointer (NRP) 7-16
7–11 Interrupt Return Pointer (IRP) 7-17
7–12 TMS320C62x Nonreset Interrupt Detection and Processing: Pipeline Operation 7-19
7–13 TMS320C67x Nonreset Interrupt Detection and Processing: Pipeline Operation 7-20
7–14 $\overline{\text{RESET}}$ Interrupt Detection and Processing: Pipeline Operation 7-22

Tables

1–1 Typical Applications for the TMS320 DSPs 1-3
2–1 40-Bit/64-Bit Register Pairs 2-4
2–2 Functional Units and Operations Performed 2-6
2–3 Control Registers 2-8
2–4 Addressing Mode Register (AMR) Mode Select Field Encoding 2-9
2–5 Block Size Calculations 2-10
2–6 Control Status Register Field Descriptions 2-11
2–7 Control Register File Extensions 2-13
2–8 Floating-Point Adder Configuration Register Field Descriptions 2-15
2–9 Floating-Point Auxiliary Configuration Register Field Descriptions 2-17
2–10 Floating-Point Multiplier Configuration Register Field Descriptions 2-19
3–1 Fixed-Point 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 TMS320C62x/C67x Opcode Map Symbol Definitions 3-9
3–5 Delay Slot and Functional Unit Latency Summary 3-12
3–6 Registers That Can Be Tested by Conditional Operations 3-16
3–7 Indirect Address Generation for Load/Store 3-23
3–8 Relationships Between Operands, Operand Size, Signed/Unsigned, Functional Units, and Opfields for Example Instruction (ADD) 3-26
3–9 Program Counter Values for Example Branch Using a Displacement 3-41
3–10 Program Counter Values for Example Branch Using a Register 3-43
3–11 Program Counter Values for B IRP 3-45
3–12 Program Counter Values for B NRP 3-47
3–13 Data Types Supported by Loads 3-67
3–14 Address Generator Options 3-67
3–15 Data Types Supported by Loads 3-72
3–16 Register Addresses for Accessing the Control Registers 3-87
3–17 Data Types Supported by Stores 3-123
3–18 Address Generator Options 3-123
3–19 Data Types Supported by Stores 3-127
4–1 Floating-Point Instruction Operation and Execution Notations 4-2
4–2 Instruction to Functional Unit Mapping 4-4
4–3 Functional Unit to Instruction Mapping 4-4
4–4 IEEE Floating-Point Notations 4-7
4–5 Special Single-Precision Values 4-8

4–6 Hex and Decimal Representation for Selected Single-Precision Values 4-9
4–7 Special Double-Precision Values 4-10
4–8 Hex and Decimal Representation for Selected Double-Precision Values 4-10
4–9 Delay Slot and Functional Unit Latency Summary 4-11
4–10 Address Generator Options 4-52
5–1 Operations Occurring During Fixed-Point Pipeline Phases 5-7
5–2 Execution Stage Length Description for Each Instruction Type 5-11
5–3 Program Memory Accesses Versus Data Load Accesses 5-22
5–4 Loads in Pipeline From Example 5–2 5-25
6–1 Operations Occurring During Floating-Point Pipeline Phases 6-7
6–2 Execution Stage Length Description for Each Instruction Type 6-13
6–3 Single-Cycle .S-Unit Instruction Hazards 6-21
6–4 DP Compare .S-Unit Instruction Hazards 6-22
6–5 2-Cycle DP .S-Unit Instruction Hazards 6-23
6–6 Branch .S-Unit Instruction Hazards 6-24
6–7 16 × 16 Multiply .M-Unit Instruction Hazards 6-25
6–8 4-Cycle .M-Unit Instruction Hazards 6-26
6–9 MPYI .M-Unit Instruction Hazards 6-27
6–10 MPYID .M-Unit Instruction Hazards 6-28
6–11 MPYDP .M-Unit Instruction Hazards 6-29
6–12 Single-Cycle .L-Unit Instruction Hazards 6-30
6–13 4-Cycle .L-Unit Instruction Hazards 6-31
6–14 INTDP .L-Unit Instruction Hazards 6-32
6–15 ADDDP/SUBDP .L-Unit Instruction Hazards 6-33
6–16 Load .D-Unit Instruction Hazards 6-34
6–17 Store .D-Unit Instruction Hazards 6-35
6–18 Single-Cycle .D-Unit Instruction Hazards 6-36
6–19 LDDW Instruction With Long Write Instruction Hazards 6-37
6–20 Single-Cycle Execution 6-38
6–21 16 × 16-Bit Multiply Execution 6-39
6–22 Store Execution 6-40
6–23 Load Execution 6-42
6–24 Branch Execution 6-44
6–25 2-Cycle DP Execution 6-46
6–26 4-Cycle Execution 6-47
6–27 INTDP Execution 6-48
6–28 DP Compare Execution 6-48
6–29 ADDDP/SUBDP Execution 6-49
6–30 MPYI Execution 6-50
6–31 MPYID Execution 6-50
6–32 MPYDP Execution 6-51
6–33 Program Memory Accesses Versus Data Load Accesses 6-56
6–34 Loads in Pipeline From Example 6–2 6-59

7–1 Interrupt Priorities 7-3
7–2 Interrupt Service Table Pointer (ISTP) Field Descriptions 7-8
7–3 Interrupt Control Registers 7-10
7–4 Control Status Register (CSR) Interrupt Control Field Descriptions 7-11

Examples

3–1 Fully Serial p-Bit Pattern in a Fetch Packet 3-14
3–2 Fully Parallel p-Bit Pattern in a Fetch Packet 3-14
3–3 Partially Serial p-Bit Pattern in a Fetch Packet 3-15
3–4 LDW in Circular Mode 3-22
3–5 ADDAH in Circular Mode 3-22
5–1 Execute Packet in Figure 5–7 5-9
5–2 Load From Memory Banks 5-24
6–1 Execute Packet in Figure 6–7 6-12
6–2 Load From Memory Banks 6-58
7–1 Relocation of Interrupt Service Table 7-9
7–2 Code Sequence to Disable Maskable Interrupts Globally 7-12
7–3 Code Sequence to Enable Maskable Interrupts Globally 7-12
7–4 Code Sequence to Enable an Individual Interrupt (INT9) 7-14
7–5 Code Sequence to Disable an Individual Interrupt (INT9) 7-14
7–6 Code to Set an Individual Interrupt (INT6) and Read the Flag Register 7-15
7–7 Code to Clear an Individual Interrupt (INT6) and Read the Flag Register 7-15
7–8 Code to Return From NMI 7-16
7–9 Code to Return from a Maskable Interrupt 7-17
7–10 Code Without Single Assignment: Multiple Assignment of A1 7-25
7–11 Code Using Single Assignment 7-25
7–12 Manual Interrupt Processing 7-26
7–13 Code Sequence to Invoke a Trap 7-27
7–14 Code Sequence for Trap Return 7-27

Chapter 1

Introduction

The TMS320C6x generation of digital signal processors is part of the TMS320 family of digital signal processors (DSPs). The TMS320C62x devices are fixed-point DSPs in the TMS320C6x generation, and the TMS320C67x devices are floating-point DSPs in the TMS320C6x generation. The TMS320C62x and TMS320C67x are code compatible and both use the VelociTI™ architecture, a high-performance, advanced VLIW (very long instruction word) architecture, making these DSPs excellent choices for multichannel and multifunction applications.

The VelociTI architecture of the ’C62x and ’C67x make them the first off-the-shelf DSPs 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 TMS320C62x/C67x 1-5

1.4 TMS320C62x/C67x Architecture 1-7

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 transformations	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	Personal digital assistants (PDA)	Speech vocoding
Channel multiplexing	Speaker phones	Text-to-speech
Data encryption	Spread spectrum communications	Voice mail
Digital PBXs	Digital subscriber loop (xDSL)	
Digital speech interpolation (DSI)	Video conferencing	
DTMF encoding/decoding	X.25 packet switching	
Echo cancellation		

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 local loop base stations
- Beam-forming base stations
- Remote access servers (RAS)
- Digital subscriber loop (DSL) systems
- Cable modems
- Multichannel telephony systems
- Virtual reality 3-D graphics
- Speech recognition
- Audio
- Radar
- Atmospheric modeling
- Finite element analysis
- Imaging (examples: fingerprint recognition, ultrasound, and MRI)

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 global positioning systems (GPS) navigation and accident avoidance
- Remote medical diagnostics

1.3 Features and Options of the TMS320C62x/C67x

The ’C62x devices operate at 200 MHz (5-ns cycle time). The ’C67x devices operate at 167 MHz (6-ns cycle time). Both DSPs execute 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 ’C62x/C67x have 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 ’C62x/C67x 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.

The ’C67x has these additional features:

- Peak 1336 MIPS at 167 MHz
- Peak 1G FLOPS at 167 MHz for single-precision operations
- Peak 250M FLOPS at 167 MHz for double-precision operations
- Peak 688M FLOPS at 167 MHz for multiply and accumulate operations
- Hardware support for single-precision (32-bit) and double-precision (64-bit) IEEE floating-point operations
- 32 $\times$ 32-bit integer multiply with 32- or 64-bit result

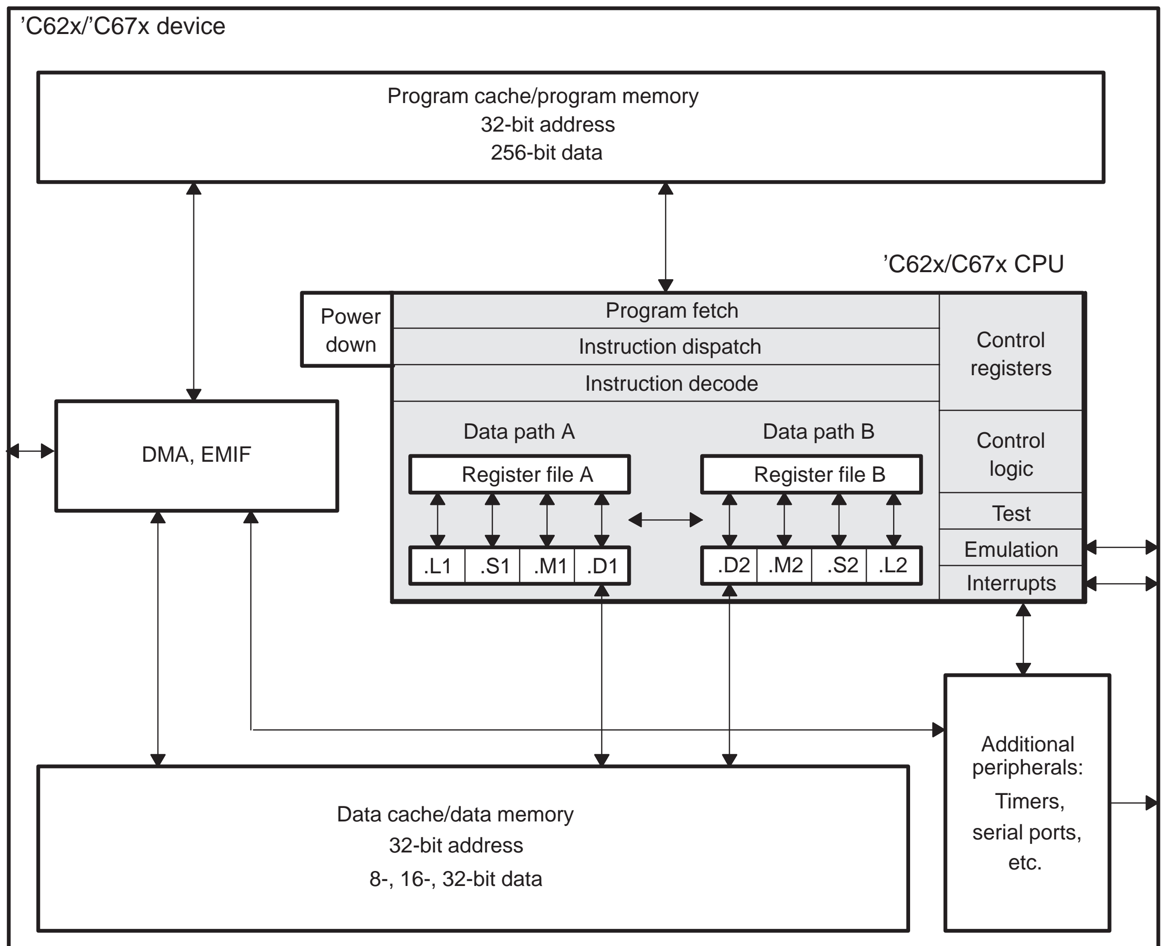
A variety of memory and peripheral options are available for the ’C62x/C67x:

- 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 access to ’C62x/C67x memory and peripherals
- Multichannel DMA controller
- Multichannel serial port(s)
- 32-bit timer(s)

1.4 TMS320C62x/C67x Architecture

Figure 1–1 is the block diagram for the TMS320C62x/C67x DSPs. The ’C62x/C67x 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. TMS320C62x/C67x Block Diagram



1.4.1 Central Processing Unit (CPU)

The ’C62x/C67x CPU, shaded in Figure 1–1, is common to all the ’C62x/C67x 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 Chapter 2, *CPU Data Paths and Control*. 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 5, *TMS320C62x Pipeline*, and Chapter 6, *TMS320C67x Pipeline*.

1.4.2 Internal Memory

The ’C62x/C67x have 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 ’C62x/C67x have two 32-bit internal ports to access internal data memory. The ’C62x/C67x have a single internal port to access internal program memory, with an instruction-fetch width of 256 bits.

1.4.3 Peripherals

The following peripheral modules can complement the CPU on the 'C62x/C67x DSPs. Some devices have 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 interface
- Power-down logic that can halt CPU activity, peripheral activity, and phased-locked loop (PLL) activity to reduce power consumption

Chapter 2

CPU Data Paths and Control

This chapter focuses on the CPU, providing information about the data paths and control registers. The two register files and the data crosspaths are described.

Figure 2–1 and Figure 2–2 show the components of the data paths the ’C62x and C67x, repectively. These components 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)

Topic **Page**

2.1 General-Purpose Register Files 2-4

2.2 Functional Units 2-6

2.3 Register File Cross Paths 2-7

2.4 Memory, Load, and Store Paths 2-7

2.5 Data Address Paths 2-7

2.6 TMS320C62x/C67x Control Register File 2-8

2.7 TMS320C67x Extensions to the Control Register File 2-13

Figure 2–1. TMS320C62x CPU Data Paths

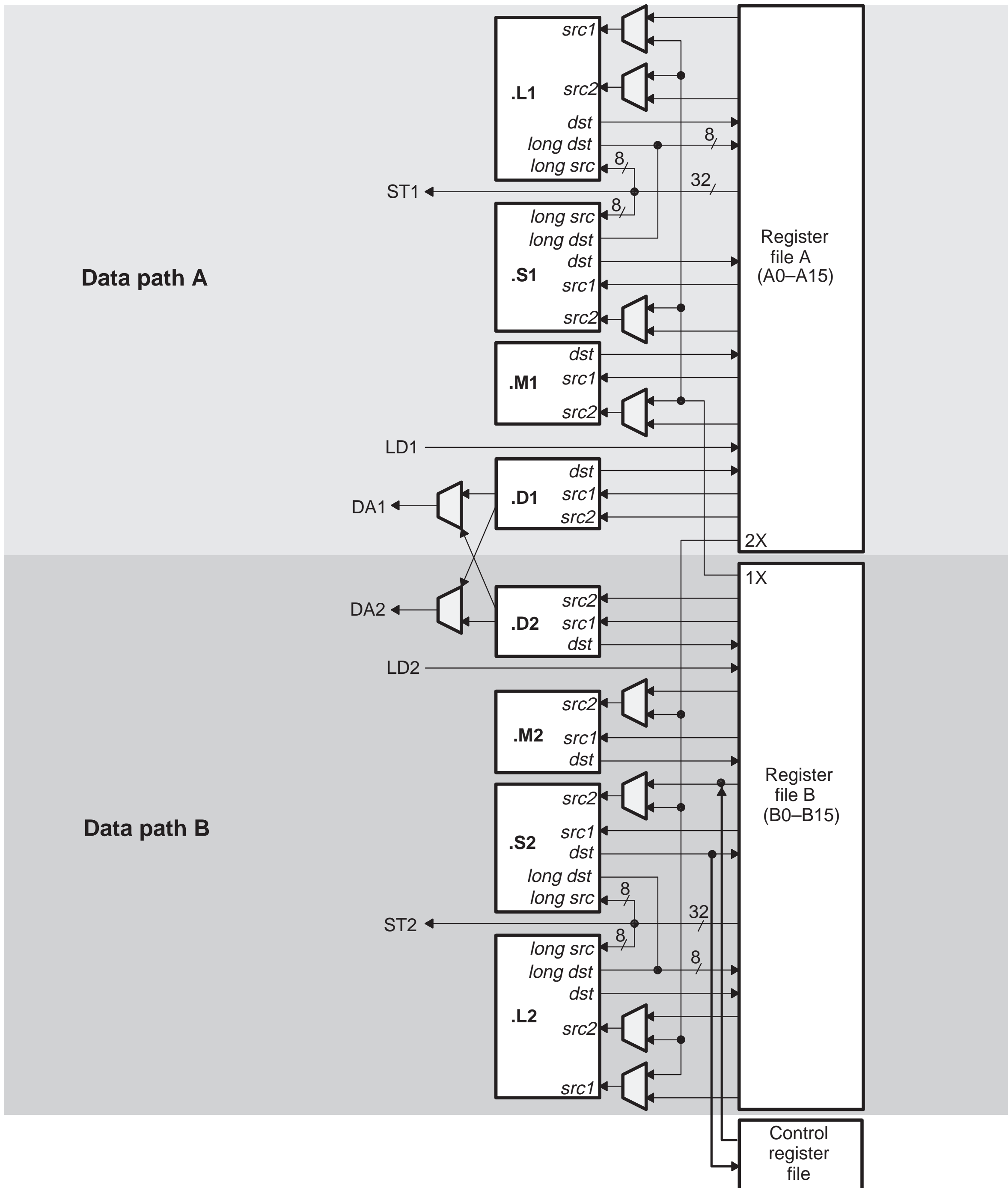
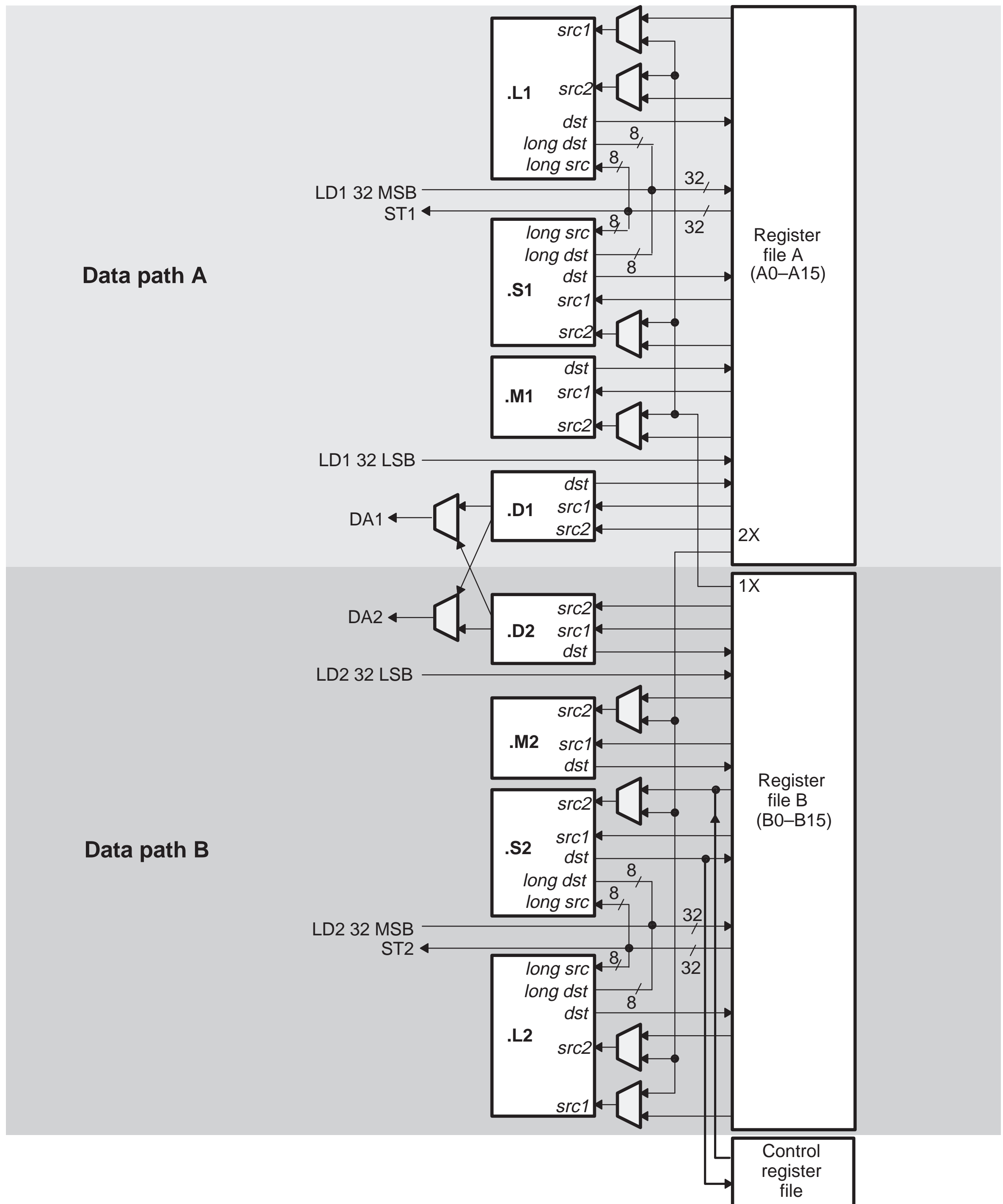


Figure 2–2. TMS320C67x CPU Data Paths



2.1 General-Purpose Register Files

There are two general-purpose register files (A and B) in the ’C62x/C67x 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.

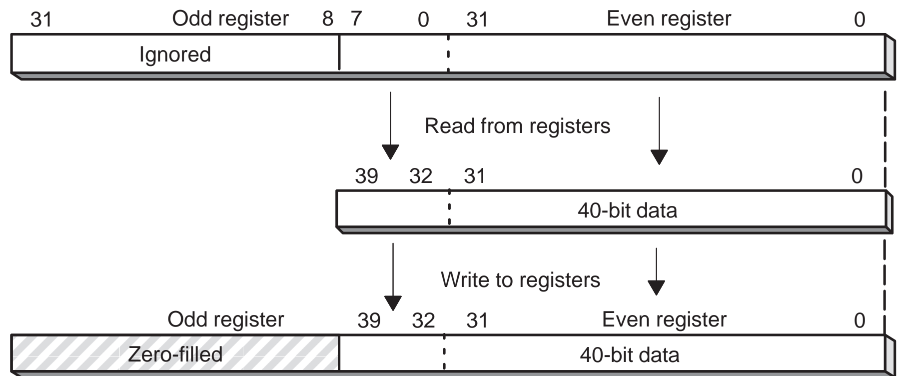
The general-purpose register files support 32- and 40-bit fixed-point data. The 32-bit data can be contained in any general-purpose register. The ’C67x also supports 32-bit single-precision and 64-bit double-precision data. The 40-bit data is contained across two registers; the 32 LSBs of the data are placed in an even register and the remaining eight MSBs are placed in the eight 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. The ’C67x also uses these register pairs to hold 64-bit double-precision floating-point values. See Chapter 4 for more information on double-precision floating-point values.

Table 2–1. 40-Bit/64-Bit Register Pairs

Register Files	
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 2–3 illustrates the register storage scheme for 40-bit long 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 2–3. Storage Scheme for 40-Bit Data in a Register Pair



2.2 Functional Units

The eight functional units in the ’C62x/C67x 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	Fixed-Point Operations	Floating-Point 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	Arithmetic operations DP → SP, INT → DP, INT → SP conversion 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)	Compare Reciprocal and reciprocal square-root operations Absolute value operations SP → DP conversion operations
.M unit (.M1, .M2)	16 × 16 bit multiply operations	32 × 32 bit fixed-point multiply operations Floating-point 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)	Load doubleword with 5-bit constant offset

Note: Fixed-point operations are available on both the ’C62x and the ’C67x. Floating-point operations and 32-bit fixed-point multiply are available only on the ’C67x.

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 *src1* and *src2*. Four units (.L1, .L2, .S1, and .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.

2.3 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.

Only two cross paths, 1X and 2X, exist in the 'C62x/C67x CPUs. This limits one source read from each data path's opposite register file per cycle, or two cross-path source reads per cycle.

2.4 Memory, Load, and Store Paths

There are two 32-bit paths for loading data from memory to the register file: LD1 for register file A, and LD2 for register file B. The 'C67x also has a second 32-bit load path for both register files A and B, which allows the LDDW instruction to simultaneously load two 32-bit registers into side A and two 32-bit registers into side 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.

2.5 Data Address Paths

The data address paths (DA1 and DA2 in Figure 2–1 and Figure 2–2) 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.

2.6 TMS320C62x/C67x Control Register File

One unit (.S2) can read from and write to the control register file, as shown in Figure 2–1 and Figure 2–2. 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, *TMS320C62x/C67x Fixed-Point Instruction Set*, for information on how to use this instruction.

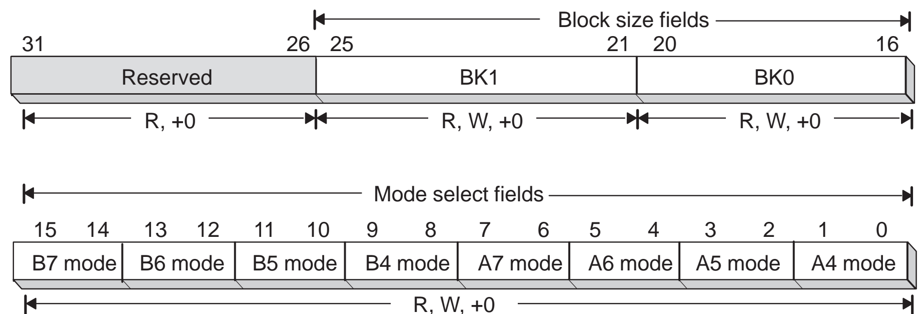
Table 2–3. Control Registers

Register			
Abbreviation	**Name**	**Description**	**Page**
AMR	Addressing mode register	Specifies whether to use linear or circular addressing for each of eight registers; also contains sizes for circular addressing	2-9
CSR	Control status register	Contains the global interrupt enable bit, cache control bits, and other miscellaneous control and status bits	2-11
IFR	Interrupt flag register	Displays status of interrupts	7-14
ISR	Interrupt set register	Allows you to set pending interrupts manually	7-14
ICR	Interrupt clear register	Allows you to clear pending interrupts manually	7-14
IER	Interrupt enable register	Allows enabling/disabling of individual interrupts	7-13
ISTP	Interrupt service table pointer	Points to the beginning of the interrupt service table	7-8
IRP	Interrupt return pointer	Contains the address to be used to return from a maskable interrupt	7-16
NRP	Nonmaskable interrupt return pointer	Contains the address to be used to return from a nonmaskable interrupt	7-16
PCE1	Program counter, E1 phase	Contains the address of the fetch packet that contains the execute packet in the E1 pipeline stage	2-12

2.6.1 Addressing Mode Register (AMR)

For each of the eight registers (A4–A7, B4–B7) that can perform linear or circular addressing, the AMR specifies the addressing mode. A 2-bit field for each register selects 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 fields and block size fields are shown in Figure 2–4, and the mode select field encoding is shown in Table 2–4.

Figure 2–4. 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 Register (AMR) Mode Select 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 0 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.

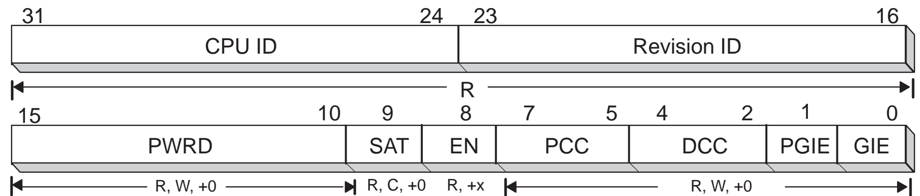
Table 2–5. Block Size Calculations

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

2.6.2 Control Status Register (CSR)

The CSR, shown in Figure 2–5, 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 *TMS320C6201/C6701 Peripherals Reference Guide* for more information on these options.

Figure 2–5. 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 2–6. Control Status Register Field Descriptions

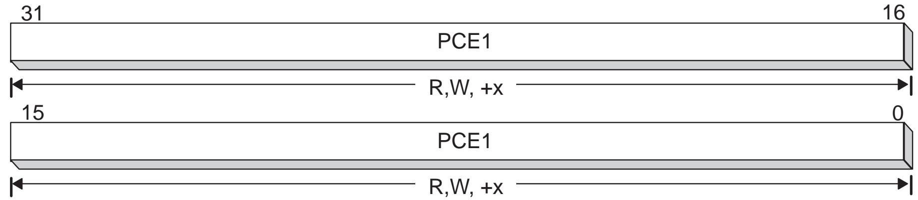
Bit Position	Width	Field Name	Function
31-24	8	CPU ID	CPU ID; defines which CPU. CPU ID = 00b: indicates ’C62x, CPU ID= 10b: indicates ’C67x
23-16	8	Revision ID	Revision ID; defines silicon revision of the CPU
15-10	6	PWRD	Control power-down modes; the values are always read as zero.†
9	1	SAT	The saturate 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. This bit will not be modified by a conditional instruction whose condition is false.
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 *TMS320C6201/C6701 Peripherals Reference Guide* for more information.

2.6.3 E1 Phase Program Counter (PCE1)

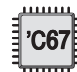
The PCE1, shown in Figure 2–6, contains the 32-bit address of the execute packet in the E1 pipeline phase.

Figure 2–6. E1 Phase Program Counter (PCE1)



Legend: R Readable by the **MVC** instruction
W Writeable by the **MVC** instruction
+x Value undefined after reset

2.7 TMS320C67x Extensions to the Control Register File

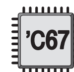


The ’C67x has three additional configuration registers to support floating point operations. The registers specify the desired floating-point rounding mode for the .L and .M units. They also contain fields to warn if *src1* and *src2* are NaN or denormalized numbers, and if the result overflows, underflows, is inexact, infinite, or invalid. There are also fields to warn if a divide by 0 was performed, or if a compare was attempted with a NaN source. Table 2–7 shows the additional registers used by the ’C67x. The OVER, UNDER, INEX, INVAL, DENn, NANn, INFO, UNORD and DIV0 bits within these registers will not be modified by a conditional instruction whose condition is false.

Table 2–7. Control Register File Extensions

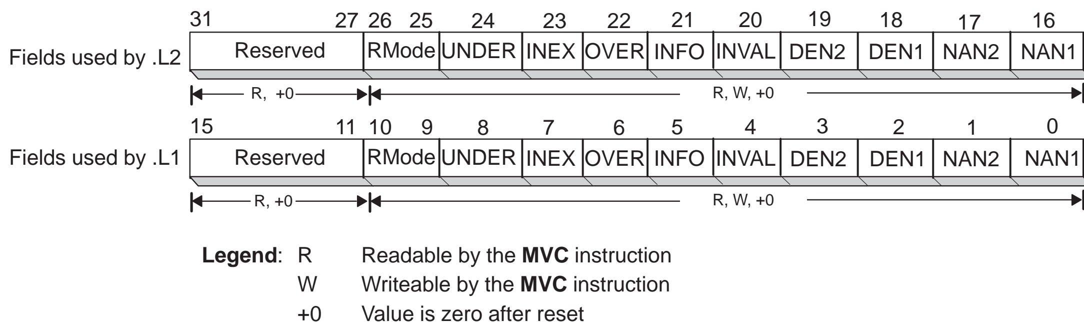
Register			
Abbreviation	**Name**	**Description**	**Page**
FADCR	Floating-point adder configuration register	Specifies underflow mode, rounding mode, NaNs, and other exceptions for the .L unit.	2-14
FAUCR	Floating-point auxiliary configuration register	Specifies underflow mode, rounding mode, NaNs, and other exceptions for the .S unit.	2-16
FMCR	Floating-point multiplier configuration register	Specifies underflow mode, rounding mode, NaNs, and other exceptions for the .M unit.	2-18

2.7.1 Floating-Point Adder Configuration Register (FADCR)



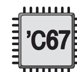
The floating-point configuration register (FADCR) contains fields that specify underflow or overflow, the rounding mode, NaNs, denormalized numbers, and inexact results for instructions that use the .L functional units. FADCR has a set of fields specific to each of the .L units, .L1 and .L2. Figure 2–7 shows the layout of FADCR. The functions of the fields in the FADCR are shown in Table 2–8.

Figure 2–7. Floating-Point Adder Configuration Register (FADCR)



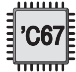
Legend: R Readable by the **MVC** instruction
W Writeable by the **MVC** instruction
+0 Value is zero after reset

Table 2–8. Floating-Point Adder Configuration Register Field Descriptions



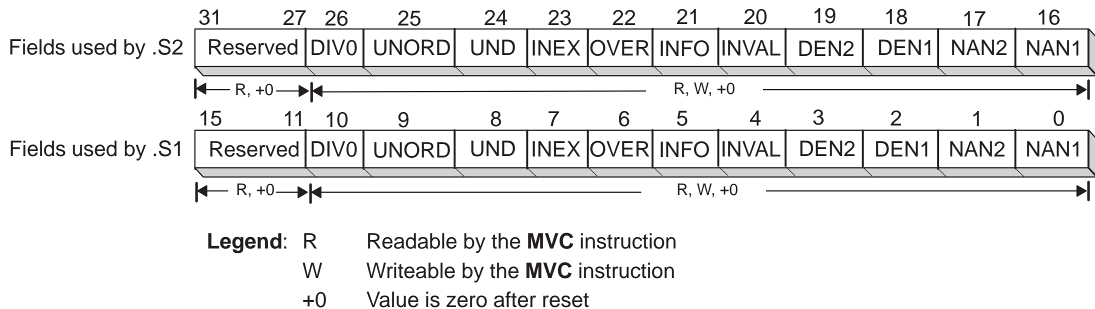
Bit Position	Width	Field Name	Function
31–27	5		Reserved
26–25	2	Rmode .L2	Value 00: Round toward nearest representable floating-point number Value 01: Round toward 0 (truncate) Value 10: Round toward infinity (round up) Value 11: Round toward negative infinity (round down)
24	1	UNDER .L2	Set to 1 when result underflows
23	1	INEX .L2	Set to 1 when result differs from what would have been computed had the exponent range and precision been unbounded; never set with INVAL
22	1	OVER .L2	Set to 1 when result overflows
21	1	INFO .L2	Set to 1 when result is signed infinity
20	1	INVAL .L2	Set to 1 when a signed NaN (SNaN) is a source, NaN is a source in a floating-point to integer conversion, or when infinity is subtracted from infinity
19	1	DEN2 .L2	*src2* is a denormalized number
18	1	DEN1 .L2	*src1* is a denormalized number
17	1	NAN2 .L2	*src2* is NaN
16	1	NAN1 .L2	*src1* is NaN
15–11	5		Reserved
10–9	2	Rmode .L1	Value 00: Round toward nearest even representable floating-point number Value 01: Round toward 0 (truncate) Value 10: Round toward infinity (round up) Value 11: Round toward negative infinity (round down)
8	1	UNDER .L1	Set to 1 when result underflows
7	1	INEX .L1	Set to 1 when result differs from what would have been computed had the exponent range and precision been unbounded; never set with INVAL
6	1	OVER .L1	Set to 1 when result overflows
5	1	INFO .L1	Set to 1 when result is signed infinity
4	1	INVAL .L1	Set to 1 when a signed NaN is a source, NaN is a source in a floating-point to integer conversion, or when infinity is subtracted from infinity
3	1	DEN2 .L1	*src2* is a denormalized number
2	1	DEN1 .L1	*src1* is a denormalized number
1	1	NAN2 .L1	*src2* is NaN
0	1	NAN1 .L1	*src1* is NaN

2.7.2 Floating-Point Auxiliary Configuration Register (FAUCR)



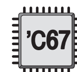
The floating-point auxiliary register (FAUCR) contains fields that specify underflow or overflow, the rounding mode, NaNs, denormalized numbers, and inexact results for instructions that use the .S functional units. FAUCR has a set of fields specific to each of the .S units, .S1 and .S2. Figure 2–8 shows the layout of FAUCR. The functions of the fields in the FAUCR are shown in Table 2–9.

Figure 2–8. Floating-Point Auxiliary Configuration Register (FAUCR)



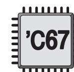
Legend: R Readable by the **MVC** instruction
W Writeable by the **MVC** instruction
+0 Value is zero after reset

Table 2–9. Floating-Point Auxiliary Configuration Register Field Descriptions



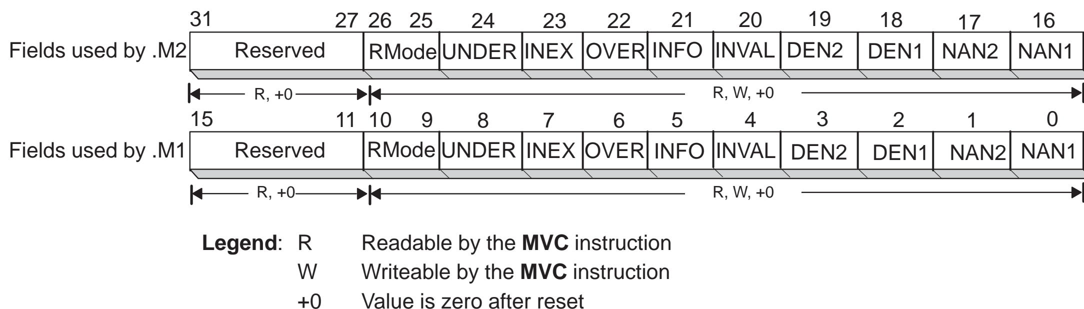
Bit Position	Width	Field Name	Function
31–27	5		Reserved
26	1	DIV0 .S2	Set to 1 when 0 is source to reciprocal operation
25	1	UNORD .S2	Set to 1 when NaN is a source to a compare operation
24	1	UNDER .S2	Set to 1 when result underflows
23	1	INEX .S2	Set to 1 when result differs from what would have been computed had the exponent range and precision been unbounded; never set with INVAL
22	1	OVER .S2	Set to 1 when result overflows
21	1	INFO .S2	Set to 1 when result is signed infinity
20	1	INVAL .S2	Set to 1 when a signed NaN (SNaN) is a source, NaN is a source in a floating-point to integer conversion, or when infinity is subtracted from infinity
19	1	DEN2 .S2	*src2* is a denormalized number
18	1	DEN1 .S2	*src1* is a denormalized number
17	1	NAN2 .S2	*src2* is NaN
16	1	NAN1 .S2	*src1* is NaN
15–11	5		Reserved
10	1	DIV0 .S1	Set to 1 when 0 is source to reciprocal operation
9	1	UNORD .S1	Set to 1 when NaN is a source to a compare operation
8	1	UNDER .S1	Set to 1 when result underflows
7	1	INEX .S1	Set to 1 when result differs from what would have been computed had the exponent range and precision been unbounded; never set with INVAL
6	1	OVER .S1	Set to 1 when result overflows
5	1	INFO .S1	Set to 1 when result is signed infinity
4	1	INVAL .S1	Set to 1 when SNaN is a source, NaN is a source in a floating-point to integer conversion, or when infinity is subtracted from infinity
3	1	DEN2 .S1	*src2* is a denormalized number
2	1	DEN1 .S1	*src1* is a denormalized number
1	1	NAN2 .S1	*src2* is a NaN
0	1	NAN1 .S1	*src1* is a NaN

2.7.3 Floating-Point Multiplier Configuration Register (FMCR)



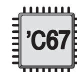
The floating-point multiplier configuration register (FMCR) contains fields that specify underflow or overflow, the rounding mode, NaNs, denormalized numbers, and inexact results for instructions that use the .M functional units. FMCR has a set of fields specific to each of the .M units, .M1 and .M2. Figure 2–9 shows the layout of FMCR. The functions of the fields in the FMCR are shown in Table 2–10.

Figure 2–9. Floating-Point Multiplier Configuration Register (FMCR)



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

Table 2–10. Floating-Point Multiplier Configuration Register Field Descriptions



Bit Position	Width	Field Name	Function
31–27	5		Reserved
26–25	2	Rmode .M2	Value 00: Round toward nearest representable floating-point number Value 01: Round toward 0 (truncate) Value 10: Round toward infinity (round up) Value 11: Round toward negative infinity (round down)
24	1	UNDER .M2	Set to 1 when result underflows
23	1	INEX .M2	Set to 1 when result differs from what would have been computed had the exponent range and precision been unbounded; never set with INVAL
22	1	OVER .M2	Set to 1 when result overflows
21	1	INFO .M2	Set to 1 when result is signed infinity
20	1	INVAL .M2	Set to 1 when SNaN is a source, NaN is a source in a floating-point to integer conversion, or when infinity is subtracted from infinity
19	1	DEN2 .M2	*src2* is a denormalized number
18	1	DEN1 .M2	*src1* is a denormalized number
17	1	NAN2 .M2	*src2* is NaN
16	1	NAN1 .M2	*src1* is NaN
15–11	5		Reserved
10–9	2	Rmode .M1	Value 00: Round toward nearest representable floating-point number Value 01: Round toward 0 (truncate) Value 10: Round toward infinity (round up) Value 11: Round toward negative infinity (round down)
8	1	UNDER .M1	Set to 1 when result underflows
7	1	INEX .M1	Set to 1 when result differs from what would have been computed had the exponent range and precision been unbounded; never set with INVAL
6	1	OVER .M1	Set to 1 when result overflows
5	1	INFO .M1	Set to 1 when result is signed infinity
4	1	INVAL .M1	Set to 1 when SNaN is a source, NaN is a source in a floating-point to integer conversion, or when infinity is subtracted from infinity
3	1	DEN2 .M1	*src2* is a denormalized number
2	1	DEN1 .M1	*src1* is a denormalized number
1	1	NAN2 .M1	*src2* is NaN
0	1	NAN1 .M1	*src1* is NaN

Chapter 3

TMS320C62x/C67x Fixed-Point Instruction Set

The ’C62x and the ’C67x share an instruction set. All of the instructions valid for the ’C62x are also valid for the ’C67x. However, because the ’C67x is a floating-point device, there are some instructions that are unique to it and do not execute on the fixed-point device. This chapter describes the assembly language instructions that are common to both the ’C62x and ’C67x digital signal processors. Also described are parallel operations, conditional operations, resource constraints, and addressing modes.

Instructions unique to the ’C67x (floating-point addition, subtraction, multiplication, and others) are described in Chapter 4.

Topic **Page**

3.1 Instruction Operation and Execution Notations 3-2
3.2 Mapping Between Instructions and Functional Units 3-4
3.3 TMS320C62x/C67x Opcode Map 3-9
3.4 Delay Slots 3-12
3.5 Parallel Operations 3-13
3.6 Conditional Operations 3-16
3.7 Resource Constraints 3-17
3.8 Addressing Modes 3-21
3.9 Individual Instruction Descriptions 3-24

3.1 Instruction Operation and Execution Notations

Table 3–1 explains the symbols used in the fixed-point instruction descriptions.

Table 3–1. Fixed-Point Instruction Operation and Execution Notations

Symbol	Meaning
abs(x)	Absolute value of x
and	Bitwise AND
–a	Perform 2s-complement subtraction using the addressing mode defined by the AMR
+a	Perform 2s-complement addition using the addressing mode defined by the AMR
$b_{y..z}$	Selection of bits y through z of bit string b
cond	Check for either *creg* equal to 0 or *creg* not equal to 0
creg	3-bit field specifying a conditional register
cstn	n-bit constant field (for example, cst5)
int	32-bit integer value
lmb0(x)	Leftmost 0 bit search of x
lmb1(x)	Leftmost 1 bit search of x
long	40-bit integer value
lsbn or LSBn	n least significant bits (for example, lsb16)
msbn or MSBn	n most significant bits (for example, msb16)
nop	No operation
norm(x)	Leftmost nonredundant sign bit of x
not	Bitwise logical complement
or	Bitwise OR
op	Opfields
R	Any general-purpose register
scstn	n-bit signed constant field
sint	Signed 32-bit integer value
slong	Signed 40-bit integer value
slsb16	Signed 16 LSB of register
smsb16	Signed 16 MSB of register

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

Symbol	Meaning
–s	Perform 2s-complement subtraction and saturate the result to the result size if an overflow occurs
+s	Perform 2s-complement addition and saturate the result to the result size if an overflow occurs
ucstn	n-bit unsigned constant field (for example, ucst5)
uint	Unsigned 32-bit integer value
ulong	Unsigned 40-bit integer value
ulsb16	Unsigned 16 LSB of register
umsb16	Unsigned 16 MSB of register
x clear *b,e*	Clear a field in x, specified by b (beginning bit) and e (ending bit)
x ext *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)
x set *b,e*	Set field in x to all 1s, specified by b (beginning bit) and e (ending bit)
xor	Bitwise exclusive OR
xsint	Signed 32-bit integer value that can optionally use cross path
xslsb16	Signed 16 LSB of register that can optionally use cross path
xsmsb16	Signed 16 MSB of register that can optionally use cross path
xuint	Unsigned 32-bit integer value that can optionally use cross path
xulsb16	Unsigned 16 LSB of register that can optionally use cross path
xumsb16	Unsigned 16 MSB of register that can optionally use cross path
$\rightarrow$	Assignment
+	Addition
$\times$	Multiplication
–	Subtraction
<<	Shift left
>>s	Shift right with sign extension
>>z	Shift right with a zero fill

3.2 Mapping Between Instructions and Functional Units

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

Table 3–2. Instruction to Functional Unit Mapping

.L Unit	.M Unit	.S Unit		.D Unit	
ABS	MPY	ADD	SET	ADD	STB (15-bit offset)‡
ADD	MPYU	ADDK	SHL	ADDAB	STH (15-bit offset)‡
ADDU	MPYUS	ADD2	SHR	ADDAH	STW (15-bit offset)‡
AND	MPYSU	AND	SHRU	ADDAW	SUB
CMPEQ	MPYH	B disp	SSHL	LDB	SUBAB
CMPGT	MPYHU	B IRP†	SUB	LDBU	SUBAH
CMPGTU	MPYHUS	B NRP†	SUBU	LDH	SUBAW
CMPLT	MPYHSU	B reg	SUB2	LDHU	ZERO
CMPLTU	MPYHL	CLR	XOR	LDW	
LMBD	MPYHLU	EXT	ZERO	LDB (15-bit offset)‡	
MV	MPYHULS	EXTU		LDBU (15-bit offset)‡	
NEG	MPYHSLU	MV		LDH (15-bit offset)‡	
NORM	MPYLH	MVC†		LDHU (15-bit offset)‡	
NOT	MPYLHU	MVK		LDW (15-bit offset)‡	
OR	MPYLUHS	MVKH		MV	
SADD	MPYLSHU	MVKLH		STB	
SAT	SMPY	NEG		STH	
SSUB	SMPYHL	NOT		STW	
SUB	SMPYLH	OR			
SUBU	SMPYH				
SUBC					
XOR					
ZERO					

† S2 only
‡ D2 only

Table 3–3. Functional Unit to Instruction Mapping

	’C62x/’C67x Functional Units			
Instruction	**.L Unit**	**.M Unit**	**.S Unit**	**.D Unit**
ABS	✓			
ADD	✓		✓	✓
ADDU	✓			
ADDAB				✓
ADDAH				✓
ADDAW				✓
ADDK			✓	
ADD2			✓	
AND	✓		✓	
B			✓	
B IRP			✓†	
B NRP			✓†	
B reg			✓†	
CLR			✓	
CMPEQ	✓			
CMPGT	✓			
CMPGTU	✓			
CMPLT	✓			
CMPLTU	✓			
EXT			✓	
EXTU			✓	
IDLE				
LDB mem				✓
LDBU mem				✓
LDH mem				✓
LDHU mem				✓

† S2 only
‡ D2 only

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

	’C62x/’C67x Functional Units			
Instruction	**.L Unit**	**.M Unit**	**.S Unit**	**.D Unit**
LDW mem				✓
LDB mem (15-bit offset)				✓‡
LDBU mem (15-bit offset)				✓‡
LDH mem (15-bit offset)				✓‡
LDHU mem (15-bit offset)				✓‡
LDW mem (15-bit offset)				✓‡
LMBD	✓			
MPY		✓		
MPYU		✓		
MPYUS		✓		
MPYSU		✓		
MPYH		✓		
MPYHU		✓		
MPYHUS		✓		
MPYHSU		✓		
MPYHL		✓		
MPYHLU		✓		
MPYHULS		✓		
MPYHSLU		✓		
MPYLH		✓		
MPYLHU		✓		
MPYLUHS		✓		
MPYLSHU		✓		
MV	✓		✓	✓
MVC†			✓	
MVK			✓	

† S2 only
‡ D2 only

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

	'C62x/'C67x Functional Units			
Instruction	**.L Unit**	**.M Unit**	**.S Unit**	**.D Unit**
MVKH			✓	
MVKLH			✓	
NEG	✓		✓	
NOP				
NORM	✓			
NOT	✓		✓	
OR	✓		✓	
SADD	✓			
SAT	✓			
SET			✓	
SHL			✓	
SHR			✓	
SHRU			✓	
SMPY		✓		
SMPYH		✓		
SMPYHL		✓		
SMPYLH		✓		
SSHL			✓	
SSUB	✓			
STB mem				✓
STH mem				✓
STW mem				✓
STB mem (15-bit offset)				✓‡
STH mem (15-bit offset)				✓‡
STW mem (15-bit offset)				✓‡
SUB	✓		✓	✓

† S2 only
‡ D2 only

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

Instruction	’C62x/’C67x Functional Units			
	.L Unit	.M Unit	.S Unit	.D Unit
SUBU	✓		✓	
SUBAB				✓
SUBAH				✓
SUBAW				✓
SUBC	✓			
SUB2			✓	
XOR	✓		✓	
ZERO	✓		✓	✓

† S2 only
‡ D2 only

3.3 TMS320C62x/C67x Opcode Map

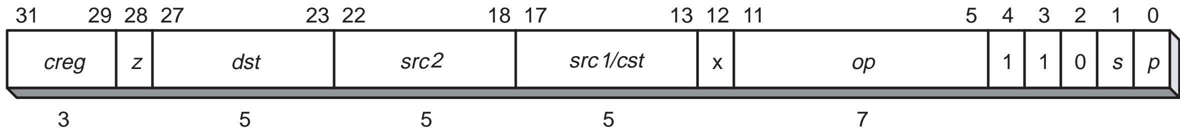
Table 3–4 and the instruction descriptions in this chapter explain the field syntaxes and values. The ’C62x and ’C67x opcodes are mapped in Figure 3–1.

Table 3–4. TMS320C62x/C67x Opcode Map Symbol Definitions

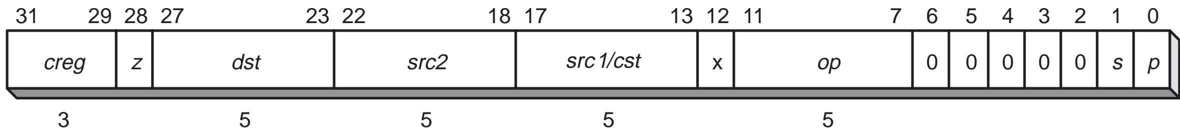
Symbol	Meaning
baseR	base address register
creg	3-bit field specifying a conditional register
cst	constant
csta	constant a
cstb	constant b
dst	destination
h	MVK or MVKH bit
ld/st	load/store opfield
mode	addressing mode
offsetR	register offset
op	opfield, field within opcode that specifies a unique instruction
p	parallel execution
r	LDDW bit
rsv	reserved
s	select side A or B for destination
src2	source 2
src1	source 1
ucstn	n-bit unsigned constant field
x	use cross path for *src2*
y	select .D1 or .D2
z	test for equality with zero or nonzero

Figure 3–1. TMS320C62x/C67x Opcode Map

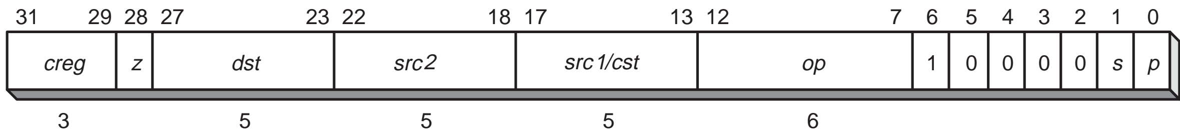
Operations on the .L unit



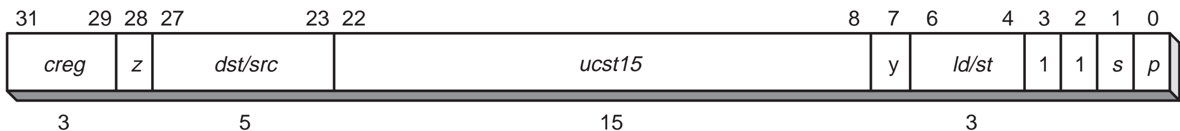
Operations on the .M unit



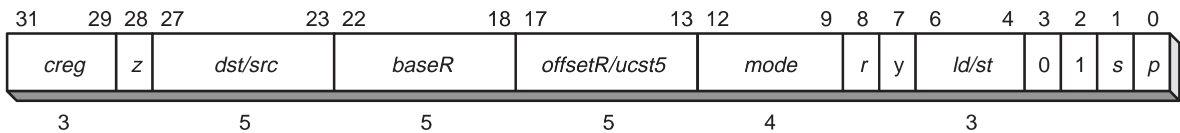
Operations on the .D unit



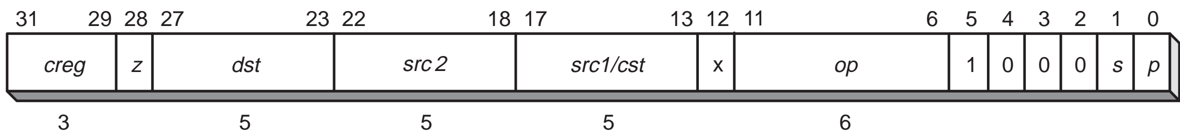
Load/store with 15-bit offset on the .D unit



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



Operations on the .S unit



ADDK on the .S unit

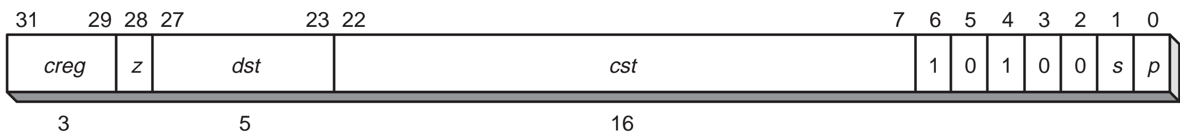
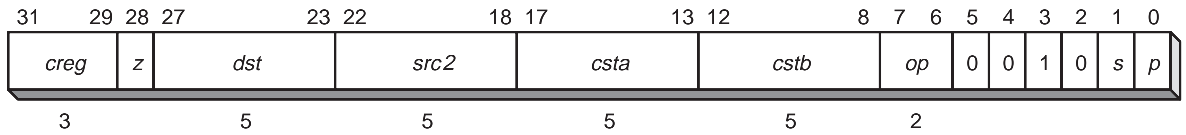
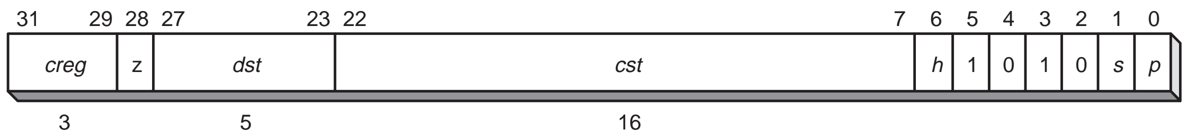


Figure 3–1. TMS320C62x/C67x Opcode Map (Continued)

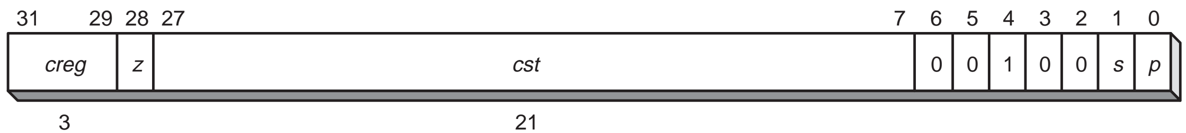
Field operations (immediate forms) on the .S unit



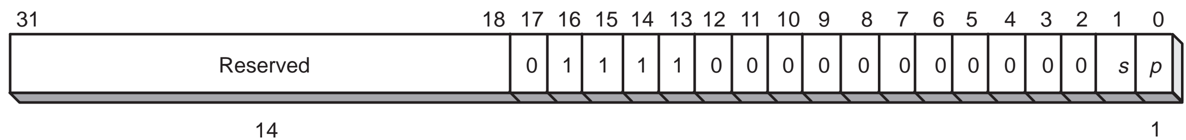
MVK and MVKH on the .S unit



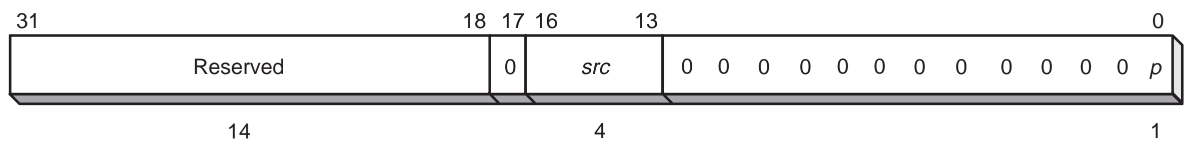
Bcond disp on the .S unit



IDLE



NOP



3.4 Delay Slots

The execution of fixed-point instructions can be defined in terms of delay slots. The number of delay slots is equivalent to the number of cycles required after the source operands are read for the result to be available for reading. 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 3–5 shows the number of delay slots associated with each type of instruction.

Delay slots are equivalent to an execution or result latency. All of the instructions that are common to the 'C62x and 'C67x have a functional unit latency of 1. This means that a new instruction can be started on the functional unit each cycle. Single-cycle throughput is another term for single-cycle functional unit latency.

Table 3–5. Delay Slot and Functional Unit Latency Summary

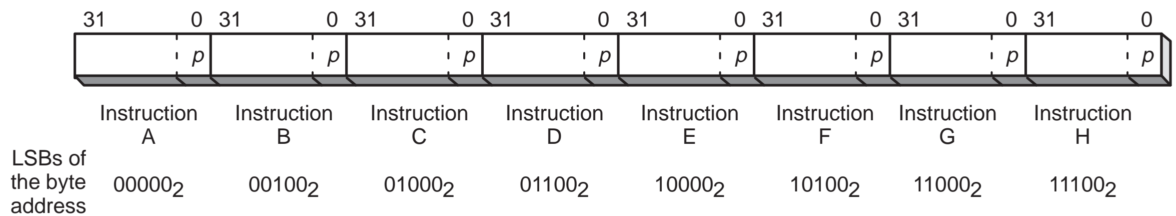
Instruction Type	Delay Slots	Functional Unit Latency	Read Cycles†	Write Cycles†	Branch Taken†
NOP (no operation)	0	1			
Store	0	1	i	i	
Single cycle	0	1	i	i	
Multiply (16 × 16)	1	1	i	i + 1	
Load	4	1	i	i, i + 4§	
Branch	5	1	i‡		i + 5

† Cycle i is in the E1 pipeline phase.
‡ The branch to label, branch to IRP, and branch to NRP instructions instruction does not read any registers.
§ The write on cycle i + 4 uses a separate write port from other .D unit instructions.

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. Fetch packets are aligned on 256-bit (8-word) boundaries.

Figure 3–2. Basic Format of a Fetch Packet



The execution of the individual instructions is partially controlled by a bit in each instruction, the *p*-bit. The *p*-bit (bit 0) determines whether the instruction executes in parallel with another instruction. 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.

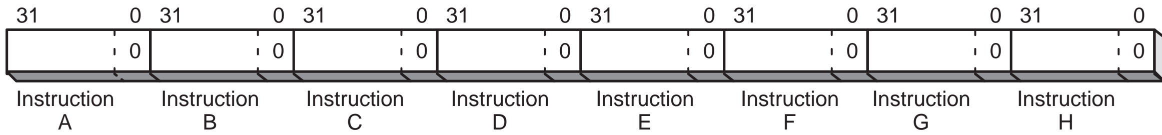
An execute packet cannot cross an 8-word boundary. Therefore, the last *p*-bit in a fetch packet is always set to 0, and each fetch packet starts a new execute packet. There are three types of *p*-bit patterns for fetch packets. These three *p*-bit patterns result in the following execution sequences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



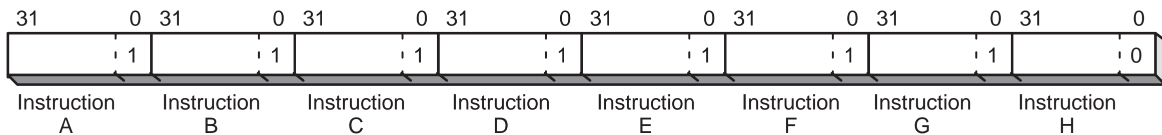
results in this execution sequence:

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:



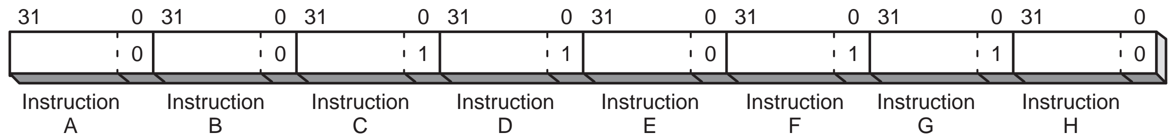
results in this execution sequence:

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 execute 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. If your result depends on executing A,B, or C, the branch to the middle of the execute packet will produce an erroneous result.

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 specified condition register is tested at the beginning of the E1 pipeline stage for all instructions. For more information on the pipeline, see Chapter 5, *TMS320C62x Pipeline*, and Chapter 6, *TMS320C67x Pipeline*. 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 3–6.

Table 3–6. Registers That Can Be Tested by Conditional Operations

Specified 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.

Conditional instructions are represented in code by using square brackets, [ ], surrounding the condition register name. 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 inverse 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 describe how an instruction can use each of the resources.

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 Figure 3–1 on page 3-10).

3.7.3 Constraints on Loads and Stores

Load/store instructions can use an address pointer from one register file while loading to or storing from the other register file. Two load/store instructions using a destination/source from the same register file cannot be issued in the same execute packet. The address register must be on the same side as the .D unit used.

The following execute packet is invalid:

```
   LDW.D1    *A0,A1 ; \ .D2 unit must use the address
|| LDW.D2    *A2,B2 ; / register from the B register file
```

The following execute packet is valid:

```
   LDW.D1    *A0,A1 ; \ Address registers from correct
|| 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 packets are valid:

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

   LDW.D1    *A0,B2 ; \ Loading to
|| LDW.D2    *B0,A1 ; / 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. All instructions with a long result on the .S and .L units have zero delay slots. See section 2.1 on page 2-4 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 sequences are invalid:

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

          MPY    .M1    A1,A1,A4 ; five reads of register A1
||        ADD    .L1    A1,A1,A5
||        SUB    .D2x   A1,B2,B3
```

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

Two instructions cannot write to the same register on the same cycle. Two instructions with the same destination can be scheduled in parallel as long as they do not write to the destination register on the same cycle. For example, a **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 unless a branch occurs after the **MPY**, causing the **ADD** not to be issued.

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

However, this code sequence is valid:

```
        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 the instructions in L5 may or may not be 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 ’C62x and ’C67x are linear, circular using BK0, and circular using BK1. The mode is specified by the addressing mode register, or AMR (defined in Chapter 2).

All registers can perform linear addressing. Only 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. **LDB(U)/LDH(U)/LDW**, **STB/STH/STW**, **ADDAB/ADDAH/ADDAW/ADDAD**, and **SUBAB/SUBAH/SUBAW** instructions all use the AMR to determine what type of address calculations are performed for these registers.

3.8.1 Linear Addressing Mode

3.8.1.1 LD/ST Instructions

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

3.8.1.2 ADDA/SUBA Instructions

For integer addition and subtraction 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

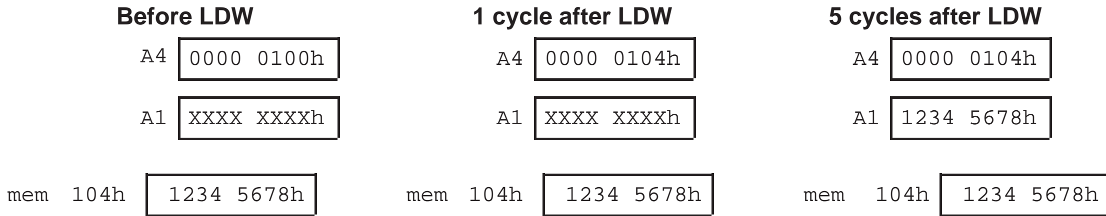
The BK0 and BK1 fields in the AMR specify block sizes for circular addressing. See section 2.6.1, on page 2-9, for more information on the AMR.

3.8.2.1 LD/ST Instructions

After shifting *offsetR/cst* to the left by 2, 1, or 0 for **LDW**, **LDH(U)**, or **LDB(U)**, 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 $\times$ 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 (124h – 20h = 104h).

3.8.2.2 ADDA/SUBA Instructions

After shifting *src1*/*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 *src2* remain unchanged. All other carries/borrows propagate as usual. If you specify *src1* 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 $\times$ 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,A4
```

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 (126h – 20h = 106h).

3.8.3 Syntax for Load/Store Address Generation

The ’C62x and ’C67x CPUs have a load/store architecture, which means that the only way to access data in memory is with a load or store instruction. Table 3–7 shows the syntax of an indirect address to a memory location. Sometimes a large offset is required for a load/store. In this case you can use the B14 or B15 register as the base register, and use a 15-bit constant (*ucst15*) as the offset.

Table 3–7. Indirect Address Generation for Load/Store

Addressing Type	No Modification of Address Register	Preincrement or Predecrement of Address Register	Postincrement or Postdecrement of Address Register
Register indirect	*R	*++R *– –R	*R++ *R– –
Register relative	*+R[*ucst5*] *–R[*ucst5*]	*++R[*ucst5*] *– –R[*ucst5*]	*R++[*ucst5*] *R– –[*ucst5*]
Register relative with 15-bit constant offset	*+B14/B15[*ucst15*]	not supported	not supported
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 fixed-point instruction set for the ’C62x and ’C67x. Each instruction presents the following information:

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

The **ADD** instruction is used as an example to familiarize you with the way each instruction is described. The example describes the kind of information you will find in each part of the individual instruction description and where to obtain 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, repeated from the summary figure on page 3-10 shows the various fields that make up each instruction. These fields are described in Table 3–4 on page 3-9.

There are instructions that can be executed on more than one functional unit. Table 3–8 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 *src1* (*scst5*), *src2* (*slong*), and *dst* (*slong*) are all signed values.

In the third row, *src1*, *src2*, 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 is set (shown in the opcode map).

Table 3–8. Relationships Between Operands, Operand Size, Signed/Unsigned, Functional Units, and Opfields for Example Instruction (ADD)

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 xuint 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*	scst5 xsint sint	.L1, .L2	0000010	ADD
src1 *src2* *dst*	scst5 slong slong	.L1, .L2	0100000	ADD
src1 *src2* *dst*	sint xsint sint	.S1, .S2	000111	ADD
src1 *src2* *dst*	scst5 xsint sint	.S1, .S2	000110	ADD
src2 *src1* *dst*	sint sint sint	.D1, .D2	010000	ADD
src2 *src1* *dst*	sint ucst5 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.

Pipeline This section contains a table that shows the sources read from, the destinations written to, and the functional unit used during each execution cycle of the instruction.

Instruction Type This section gives the type of instruction. See section 5.2 on page 5-11 for information about the pipeline execution of this type of instruction.

Delay Slots This section gives the number of delay slots the instruction takes to execute See section 3.4 on page 3-12 for an explanation of delay slots.

Functional Unit Latency

This section gives the number of cycles that the functional unit is in use during the execution of the instruction.

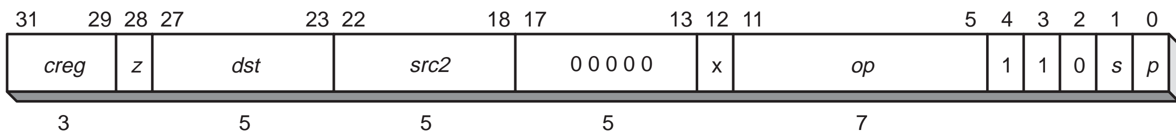
Example Examples of instruction execution. If applicable, register and memory values are given before and after instruction execution.

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

Opcode



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 sint is determined as follows:

1) If *src2* ≥ 0, then *src2* → *dst*
2) If *src2* < 0 and *src2* ≠ -2^{31}, then –*src2* → *dst*
3) If *src2* = -2^{31}, then $2^{31} - 1$ → *dst*

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

1) If *src2* ≥ 0, then *src2* → *dst*
2) If *src2* < 0 and *src2* ≠ -2^{39}, then –*src2* → *dst*
3) If *src2* = -2^{39}, then $2^{39} - 1$ → *dst*

Pipeline

Pipeline Stage	E1
Read	*src2*
Written	*dst*
Unit in use	.L

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

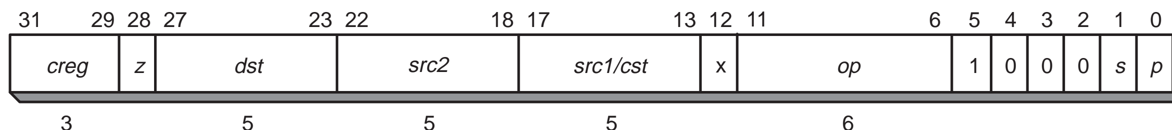
ADD (.unit) *src1*, *src2*, *dst*
or
ADDU (.L1 or .L2) *src1*, *src2*, *dst*
or
ADD (.D1 or .D2) *src2*, *src1*, *dst*

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

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 xuint 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*	scst5 xsint sint	.L1, .L2	0000010
src1 *src2* *dst*	scst5 slong slong	.L1, .L2	0100000
src1 *src2* *dst*	sint xsint sint	.S1, .S2	000111
src1 *src2* *dst*	scst5 xsint sint	.S1, .S2	000110
src2 *src1* *dst*	sint sint sint	.D1, .D2	010000
src2 *src1* *dst*	sint ucst5 sint	.D1, .D2	010010

Opcode .L unit

Opcode .S unit



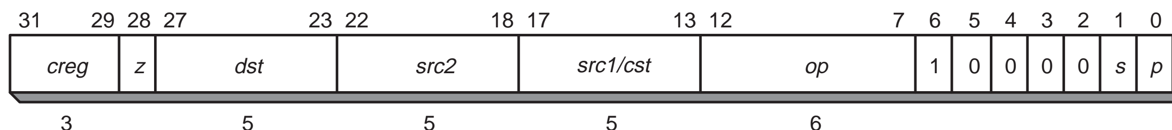
Description for .L1, .L2 and .S1, .S2 Opcodes

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

Opcode .D unit



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

Pipeline

Pipeline Stage	E1
Read	*src1, src2*
Written	*dst*
Unit in use	.L, .S, or .D

Instruction Type Single-cycle

Delay Slots 0

Example 1

```
ADD .L2X    A1,B1,B2
```

	Before instruction			1 cycle after instruction	
A1	0000 325Ah	12890	A1	0000 325Ah	
B1	FFFF FF12h	−238	B1	FFFF FF12h	
B2	XXXX XXXXh		B2	0000 316Ch	12652

Example 2

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

	Before instruction			1 cycle after instruction		
A1	0000 325Ah	12890†	A1	0000 325Ah		
A2	FFFF FF12h	4294967058†	A2	FFFF FF12h		
A5:A4	XXXX XXXX		A5:A4	0000 0001h	0000 316Ch	4294979948‡

Example 3

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

	Before instruction				1 cycle after instruction		
A1	0000 325Ah		12890	A1	0000 325Ah		
A3:A2	0000 00FFh	FFFF FF12h	1099511627538‡	A3:A2	0000 00FFh	FFFF FF12h	
A5:A4	0000 0000h	0000 0000h	0	A5:A4	0000 0000h	0000 316Ch	12652‡

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

Example 4

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

	Before instruction				1 cycle after instruction		
A1	0000 325Ah		12890	A1	0000 325Ah		
A3:A2	0000 00FFh	FFFF FF12h	−228§	A3:A2	0000 00FFh	FFFF FF12h	
A5:A4	0000 0000h	0000 0000h	0§	A5:A4	0000 0000h	0000 316Ch	12652§

§ Signed 40-bit (long) integer

Example 5

```
ADD .L1     -13,A1,A6
```

	Before instruction			1 cycle after instruction	
A1	0000 325Ah	12890	A1	0000 325Ah	
A6	XXXX XXXXh		A6	0000 324Dh	12877

Example 6

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

	Before instruction			1 cycle after instruction	
A1	0000 325Ah	12890	A1	0000 325Ah	
A6	XXXX XXXXh		A6	0000 3274h	12916

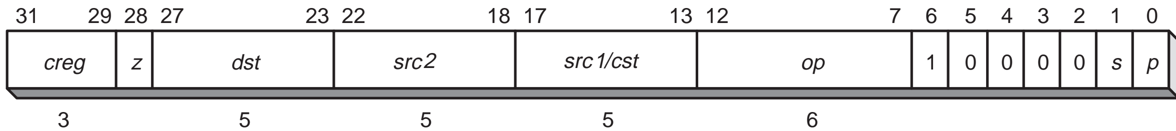
Syntax

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

Opcode



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 2.6.1). *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* $\rightarrow$ *dst*
else nop

Pipeline

Pipeline stage	E1
Read	*src1, src2*
Written	*dst*
Unit in use	.D

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
```

	Before instruction		1 cycle after instruction
A2	0000 000Bh	A2	0000 000Bh
A4	0000 0100h	A4	0000 0106h
AMR	0002 0001h	AMR	0002 0001h

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

Example 3

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

	Before instruction		1 cycle after instruction
A4	0002 0000h	A4	0002 0000h
AMR	0002 0001h	AMR	0002 0001h

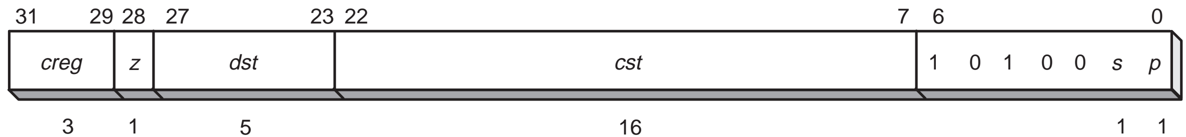
```
BK0 = 2 → size = 8
A4 in circular addressing mode using BK0
```

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

.unit = .S1 or .S2

Opcode map field used...	For operand type...	Unit
cst *dst*	*scst16* uint	.S1, .S2

Opcode



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

Pipeline

Pipeline Stage	E1
Read	*cst*
Written	*dst*
Unit in use	.S

Instruction Type Single-cycle

Delay Slots 0

Example

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

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

.unit = .S1 or .S2

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

Opcode

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

Pipeline

Pipeline Stage	E1
Read	*src1, src2*
Written	*dst*
Unit in use	.S

Instruction Type Single-cycle

Delay Slots 0

Example

```
ADD2 .S1X  A1,B1,A2
```

	Before instruction			1 cycle after instruction	
A1	0021 37E1h	33 14305	A1	0021 37E1h	
A2	XXXX XXXXh		A2	03BB 1C99h	955 7321
B1	039A E4B8h	922 58552	B1	039A E4B8h	

Syntax

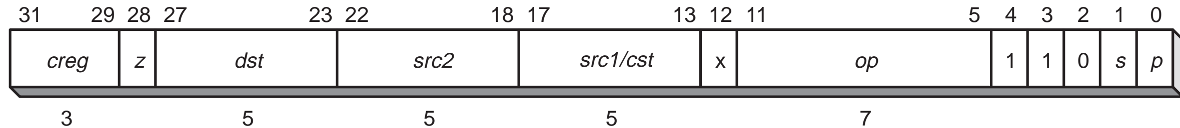
AND (.unit) *src1*, *src2*, *dst*

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

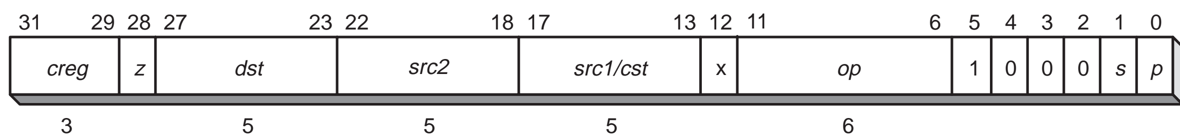
Opcode map field used...	For operand type...	Unit	Opfield
src1 *src2* *dst*	uint xuint uint	.L1, .L2	1111011
src1 *src2* *dst*	scst5 xuint uint	.L1, .L2	1111010
src1 *src2* *dst*	uint xuint uint	.S1, .S2	011111
src1 *src2* *dst*	scst5 xuint uint	.S1, .S2	011110

Opcode

.L unit form:



.S unit form:



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

Pipeline

Pipeline Stage	E1
Read	*src1, src2*
Written	*dst*
Unit in use	.L or .S

Instruction Type Single-cycle

Example 1

```
AND .L1X   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

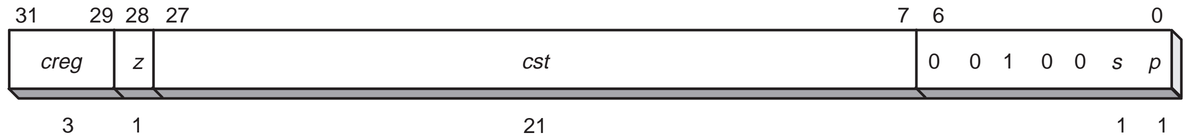
Syntax

B (.unit) label

.unit = .S1 or .S2

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

Opcode



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 if one branch uses a displacement and the other uses a register, IRP, or NRP. As long as only one branch has a true condition, the code executes in a 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) The execute packets in the delay slots of a branch cannot be interrupted. This is true regardless of whether the branch is taken.

3) See section 3.5.2 on page 3-15 for information on branching into the middle of an execute packet.

Pipeline

Pipeline Stage	E1	Target Instruction PS	PW	PR	DP	DC	E1
Read							
Written							
Branch Taken							✓
Unit in use	.S						

Instruction Type Branch

Delay Slots 5

Table 3–9 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   .M1X   A3, B3, A4
0000 0010      || SUB   .D1    A5, A6, A6
0000 0014         MPY   .M1    A3, A6, A5
0000 0018         MPY   .M1    A6, A7, A8
0000 001C         SHR   .S1    A4, 15, A4
0000 0020         ADD   .D1    A4, A6, A4
```

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

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	

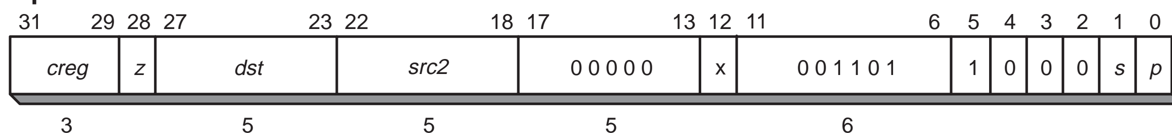
Syntax

B (.unit) *src2*

.unit = .S2

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

Opcode



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 if one branch uses a displacement and the other uses a register, IRP, or NRP. As long as onlly one branch has a true condition, the code executes in a well-defined way.

Execution

if (cond) *src2* $\rightarrow$ PFC
else nop

Notes:

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

Pipeline

		Target Instruction					
Pipeline Stage	**E1**	**PS**	**PW**	**PR**	**DP**	**DC**	**E1**
Read	*src2*						
Written							
Branch Taken							✓
Unit in use	.S2						

Instruction Type Branch

Delay Slots 5

Table 3–10 gives the program counter values and actions for the following code example. In this example, the B10 register holds the value 1000 000Ch.

Example

```
          B10 [1000 000Ch]

1000 0000         B      .S2    B10
1000 0004         ADD    .L1    A1, A2, A3
1000 0008      || ADD    .L2    B1, B2, B3
1000 000C         MPY    .M1X   A3, B3, A4
1000 0010      || SUB    .D1    A5, A6, A6
1000 0014         MPY    .M1    A3, A6, A5
1000 0018         MPY    .M1    A6, A7, A8
1000 001C         SHR    .S1    A4, 15, A4
1000 0020         ADD    .D1    A4, A6, A4
```

Table 3–10. Program Counter Values for Example Branch Using a Register

Cycle	Program Counter Value	Action
Cycle 0	1000 0000h	Branch command executes (target code fetched)
Cycle 1	1000 0004h	
Cycle 2	1000 000Ch	
Cycle 3	1000 0014h	
Cycle 4	1000 0018h	
Cycle 5	1000 001Ch	
Cycle 6	1000 000Ch	Branch target code executes
Cycle 7	1000 0014h	

Syntax **B** (.unit) IRP

.unit = .S2

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

Opcode

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 if one branch uses a displacement and the other uses a register, IRP, or NRP. As long as only one branch has a ture condition, the code executes in a 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) The execute packets in the delay slots of a branch cannot be interrupted. This is true regardless of whether the branch is taken.

Pipeline

		Target Instruction					
Pipeline Stage	**E1**	**PS**	**PW**	**PR**	**DP**	**DC**	**E1**
Read	IRP						
Written							
Branch Taken							✓
Unit in use	.S2						

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 1000 |   IRP = | 0000 1000 |

0000 0020    B      .S2       IRP
0000 0024    ADD    .S1       A0, A2, A1
0000 0028    MPY    .M1       A1, A0, A1
0000 002C    NOP
0000 0030    SHR    .S1       A1, 15, A1
0000 0034    ADD    .L1       A1, A2, A1
0000 0038    ADD    .L2       B1, B2, B3
```

Table 3–11. Program Counter Values for B IRP

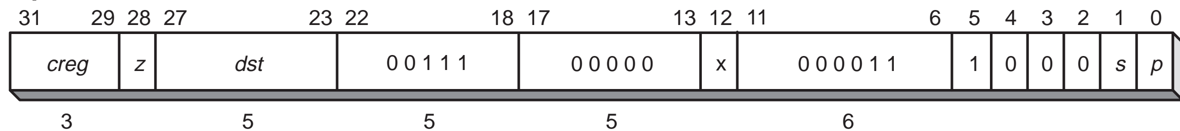
Cycle	Program Counter Value (Hex)	Action
Cycle 0	0000 0020	Branch command executes (target code fetched)
Cycle 1	0000 0024	
Cycle 2	0000 0028	
Cycle 3	0000 002C	
Cycle 4	0000 0030	
Cycle 5	0000 0034	
Cycle 6	0000 1000	Branch target code executes

Syntax **B** (.unit) NRP

.unit = .S2

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

Opcode



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 if one branch uses a displacement and the other uses a register, IRP, or NRP. As long as only one branch has a true condition, the code executes in a 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) The execute packets in the delay slots of a branch cannot be interrupted. This is true regardless of whether the branch is taken.

Pipeline

		Target Instruction					
Pipeline Stage	**E1**	**PS**	**PW**	**PR**	**DP**	**DC**	**E1**
Read	NRP						
Written							
Branch Taken							✓
Unit in use	.S2						

Instruction Type Branch

Delay Slots 5

Table 3–12 gives the program counter values and actions for the following code example.

Example Given that an interrupt occurred at

```
PC = | 0000 1000 |   NRP = | 0000 1000 |

0000 0020    B      .S2       NRP
0000 0024    ADD    .S1       A0, A2, A1
0000 0028    MPY    .M1       A1, A0, A1
0000 002C    NOP
0000 0030    SHR    .S1       A1, 15, A1
0000 0034    ADD    .L1       A1, A2, A1
0000 0038    ADD    .L2       B1, B2, B3
```

Table 3–12. Program Counter Values for B NRP

Cycle	Program Counter Value (Hex)	Action
Cycle 0	0000 0020	Branch command executes (target code fetched)
Cycle 1	0000 0024	
Cycle 2	0000 0028	
Cycle 3	0000 002C	
Cycle 4	0000 0030	
Cycle 5	0000 0034	
Cycle 6	0000 1000	Branch target code executes

Syntax

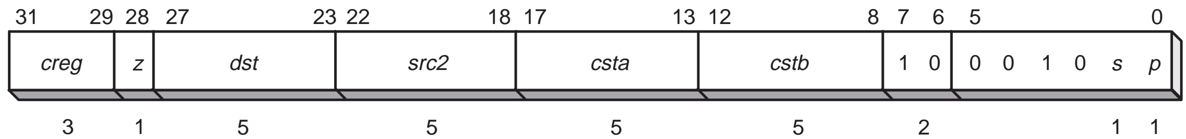
CLR (.unit) *src2*, *csta*, *cstb*, *dst*
or
CLR (.unit) *src2*, *src1*, *dst*

.unit = .S1 or .S2

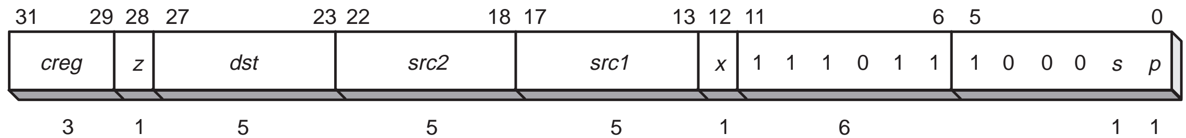
Opcode map field used...	For operand type...	Unit	Opfield
src2 *csta* *cstb* *dst*	uint ucst5 ucst5 uint	.S1, .S2	11
src2 *src1* *dst*	xuint uint uint	.S1, .S2	111111

Opcode

Constant form:

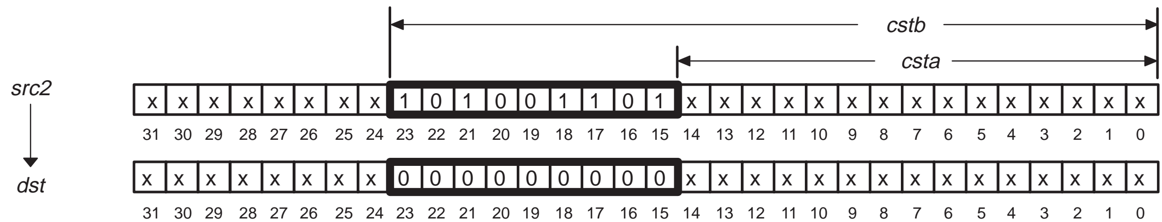


Register form:



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. Only the ten LSBs are valid for the register version of the instruction. If any of the 22 MSBs are non-zero, the result is invalid.



Execution

If the constant form is used:

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

If the register form is used:

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

Pipeline

Pipeline Stage	E1
Read	*src1, src2*
Written	*dst*
Unit in use	.S

Instruction Type Single-cycle

Delay Slots 0

Example 1

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

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

Example 2

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

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

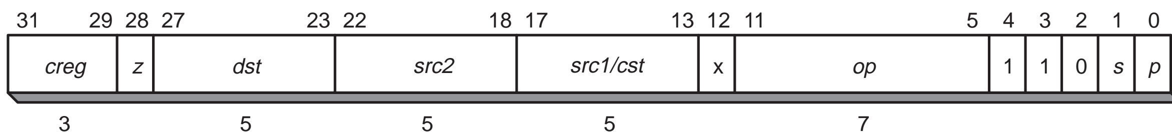
Syntax

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*	scst5 xsint uint	.L1, .L2	1010010
src1 *src2* *dst*	xsint slong uint	.L1, .L2	1010001
src1 *src2* *dst*	scst5 slong uint	.L1, .L2	1010000

Opcode



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
```

Pipeline

Pipeline Stage	E1
Read	*src1, src2*
Written	*dst*
Unit in use	.L

Instruction Type Single-cycle

Delay Slots 0

Example 1

```
CMPEQ .L1X A1,B1,A2
```

	Before instruction			1 cycle after instruction	
A1	0000 4B8h	1208	A1	0000 4B8h	
A2	XXXX XXXXh		A2	0000 0000h	false
B1	0000 4B7h	1207	B1	0000 4B7h	

Example 2

```
CMPEQ .L1  Ch,A1,A2
```

	Before instruction			1 cycle after instruction	
A1	0000 000Ch	12	A1	0000 000Ch	
A2	XXXX XXXXh		A2	0000 0001h	true

Example 3

```
CMPEQ .L2X A1,B3:B2,B1
```

	Before instruction			1 cycle after instruction	
A1	F23A 3789h		A1	F23A 3789h	
B1	XXXX XXXXh		B1	0000 0001h	true
B3:B2	0000 0FFh	F23A 3789h	B3:B2	0000 00FFh	F23A 3789h

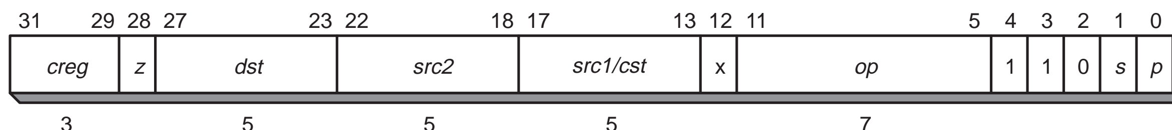
Syntax

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*	scst5 xsint uint	.L1, .L2	1000110	CMPGT
src1 *src2* *dst*	xsint slong uint	.L1, .L2	1000101	CMPGT
src1 *src2* *dst*	scst5 slong uint	.L1, .L2	1000100	CMPGT
src1 *src2* *dst*	uint xuint uint	.L1, .L2	1001111	CMPGTU
src1 *src2* *dst*	ucst4 xuint uint	.L1, .L2	1001110	CMPGTU
src1 *src2* *dst*	xuint ulong uint	.L1, .L2	1001101	CMPGTU
src1 *src2* *dst*	ucst4 ulong uint	.L1, .L2	1001100	CMPGTU

Opcode



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*. Only the four LSBs are valid in the 5-bit *cst* field when the ucst4 operand is used. If the MSB of the *cst* field is non-zero, the result is invalid.

Execution

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

Pipeline

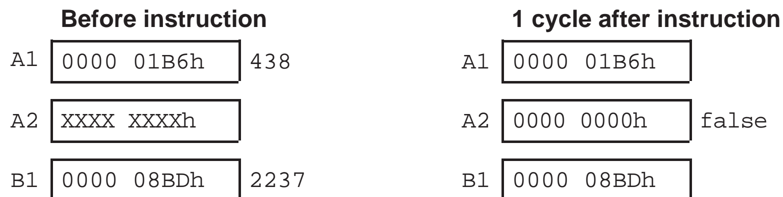
Pipeline Stage	E1
Read	*src1, src2*
Written	*dst*
Unit in use	.L

Instruction Type Single-cycle

Delay Slots 0

Example 1

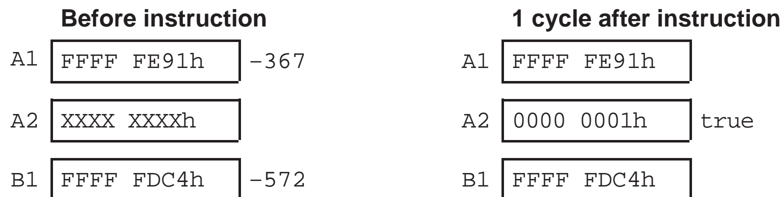
```
CMPGT .L1X A1,B1,A2
```



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

Example 2

```
CMPGT .L1X A1,B1,A2
```



	Before instruction			1 cycle after instruction	
A1	FFFF FE91h	−367	A1	FFFF FE91h	
A2	XXXX XXXXh		A2	0000 0001h	true
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	false

Example 4

```
CMPGT .L1X A1,B1,A2
```

	Before instruction			1 cycle after instruction	
A1	0000 00EBh	235	A1	0000 00EBh	
A2	XXXX XXXXh		A2	0000 0000h	false
B1	0000 00EBh	235	B1	0000 00EBh	

Example 5

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

	Before instruction			1 cycle after instruction	
A1	0000 0128h	296†	A1	0000 0128h	
A2	FFFF FFDEh	4294967262†	A2	FFFF FFDEh	
A3	XXXX XXXXh		A3	0000 0000h	false

Example 6

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

	Before instruction			1 cycle after instruction	
A1	0000 0005h	5†	A1	0000 0005h	
A2	XXXX XXXXh		A2	0000 0001h	true

Example 7

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

	Before instruction				1 cycle after instruction		
A3:A2	0000 0000h	0000 000Ah	10‡	A3:A2	0000 0000h	0000 000Ah	
A4	XXXX XXXXh			A4	0000 0001h	true	

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

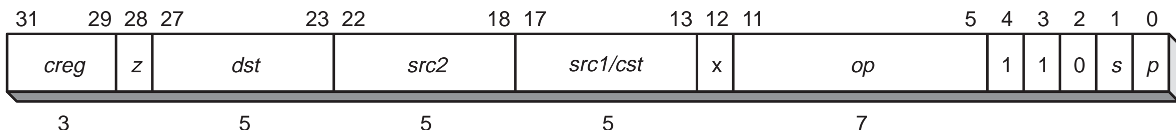
Syntax

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
src1 *src2* *dst*	sint xsint uint	.L1, .L2	1010111	CMPLT
src1 *src2* *dst*	scst5 xsint uint	.L1, .L2	1010110	CMPLT
src1 *src2* *dst*	xsint slong uint	.L1, .L2	1010101	CMPLT
src1 *src2* *dst*	scst5 slong uint	.L1, .L2	1010100	CMPLT
src1 *src2* *dst*	uint xuint uint	.L1, .L2	1011111	CMPLTU
src1 *src2* *dst*	ucst4 xuint uint	.L1, .L2	1011110	CMPLTU
src1 *src2* *dst*	xuint ulong uint	.L1, .L2	1011101	CMPLTU
src1 *src2* *dst*	ucst4 ulong uint	.L1, .L2	1011100	CMPLTU

Opcode



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
```

Pipeline

Pipeline Stage	E1
Read	*src1, src2*
Written	*dst*
Unit in use	.L

Instruction Type Single-cycle

Delay Slots 0

Example 1

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

	Before instruction			1 cycle after instruction	
A1	0000 07E2h	2018	A1	0000 07E2h	
A2	0000 0F6Bh	3947	A2	0000 0F6Bh	
A3	XXXX XXXXh		A3	0000 0001h	true

Example 2

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

	Before instruction			1 cycle after instruction	
A1	FFFF FED6h	−298	A1	FFFF FED6h	
A2	0000 000Ch	12	A2	0000 000Ch	
A3	XXXX XXXXh		A3	0000 0001h	true

Example 3

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

	Before instruction			1 cycle after instruction	
A1	0000 0005h	5	A1	0000 0005h	
A2	XXXX XXXXh		A2	0000 0000h	false

Example 4

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

	Before instruction			1 cycle after instruction	
A1	0000 289Ah	10394†	A1	0000 289Ah	
A2	FFFF F35Eh	4294964062†	A2	FFFF F35Eh	
A3	XXXX XXXXh		A3	0000 0001h	true

† Unsigned 32-bit integer

Example 5

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

	Before instruction			1 cycle after instruction	
A1	0000 000Fh	15†	A1	0000 000Fh	
A2	XXXX XXXXh		A2	0000 0001h	true

Example 6

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

	Before instruction				1 cycle after instruction		
A1		003B 8260h	3900000†	A1	003B 8260h		
A2		XXXX XXXXh		A2	0000 0000h		false
A5:A4	0000 0000h	003A 0002h	3801090‡	A5:A4	0000 0000h	003A 0002h	

† 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 map field used...	For operand type...	Unit
src2 *csta* *cstb* *dst*	sint ucst5 ucst5 sint	.S1, .S2
src2 *src1* *dst*	xsint uint sint	.S1, .S2

Opcode

Constant form:

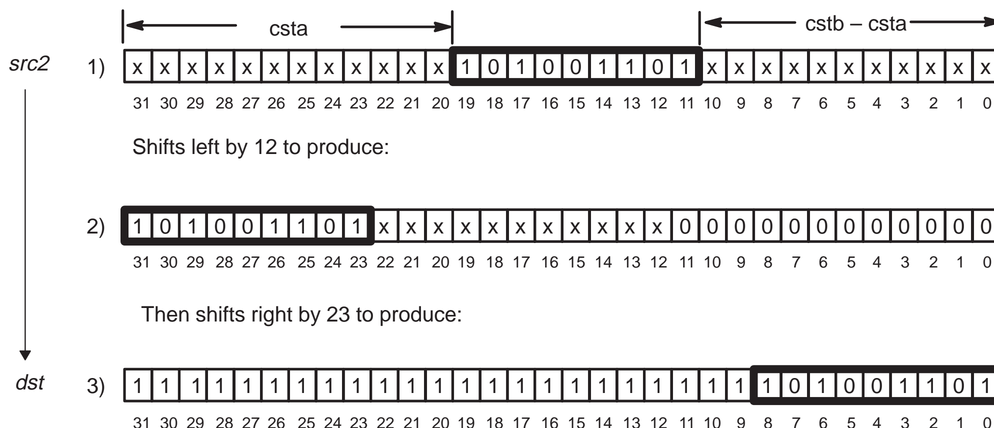
31–29	28	27–23	22–18	17–13	12–8	7–6	5–0					
creg	*z*	*dst*	*src2*	*csta*	*cstb*	0 1	0	0	1	0	*s*	*p*
3	1	5	5	5	5	2					1	1

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

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* = *csta* + LSB of the field. 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 12 and *cstb* is 11 + 12 = 23. Only the ten LSBs are valid for the register version of the instruction. If any of the 22 MSBs are non-zero, the result is invalid.



Execution If the constant form is used:

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

If the register form is used:

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

Pipeline

Pipeline Stage	E1
Read	*src1, src2*
Written	*dst*
Unit in use	.S

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 map field used...	For operand type...	Unit
src2 *csta* *cstb* *dst*	uint ucst5 ucst5 uint	.S1, .S2
src2 *src1* *dst*	xuint uint uint	.S1, .S2

Opcode

Constant width and offset form:

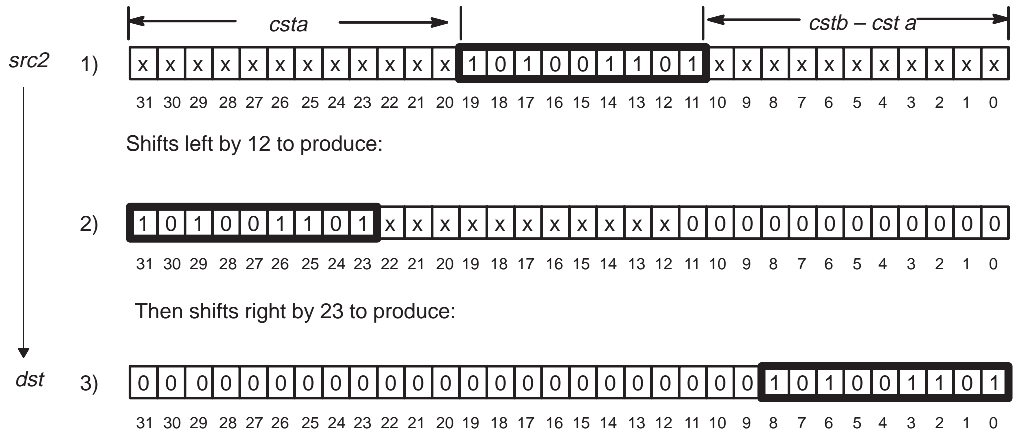
31–29	28	27–23	22–18	17–13	12–8	7–6	5–0
creg	*z*	*dst*	*src2*	*csta*	*cstb*	0 0	0 0 1 0 *s* *p*
3	1	5	5	5	5	2	1 1

Register width and offset form:

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

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* = *csta* + LSB of the field. 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 12 and *cstb* is 11 + 12 = 23. Only the ten LSBs are valid for the register version of the instruction. If any of the 22 MSBs are non-zero, the result is invalid.



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

Pipeline

Pipeline Stage	E1
Read	*src1, src2*
Written	*dst*
Unit in use	.S

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

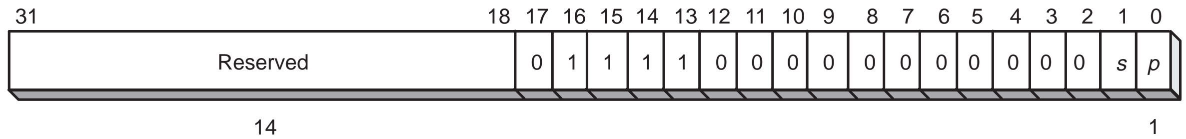
Example 2

```
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 an infinite multicycle **NOP** that terminates upon servicing an interrupt, or a branch occurs due to an **IDLE** instruction being in the delay slots of a branch.

Instruction Type NOP

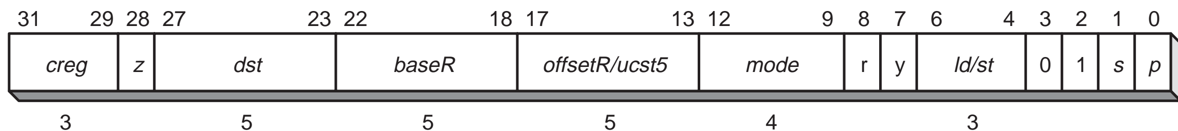
Delay Slots 0

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 loads from memory to a general-purpose register (*dst*). Table 3–13 summarizes the data types supported by loads. Table 3–14 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 2.6.1 on page 2-9).

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. The *r* bit should be set to zero.

Table 3–13. 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 3–14. 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 0 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 *ucst5* 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. You must type either brackets or parentheses around the specified offset if you use the optional offset parameter.

Word and halfword addresses must be aligned on word (two LSBs are 0) and halfword (LSB is 0) boundaries, respectively.

Execution

if (cond) mem → *dst*
else nop

Pipeline

Pipeline Stage	E1	E2	E3	E4	E5
Read	*baseR* *offsetR*				
Written	*baseR*				*dst*
Unit in use	.D				

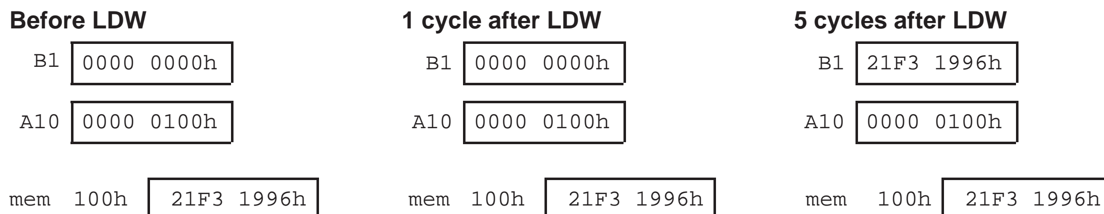
Instruction Type Load

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 5, *TMS320C62x Pipeline*, and Chapter 6, *TMS320C67x Pipeline.*

Example 1

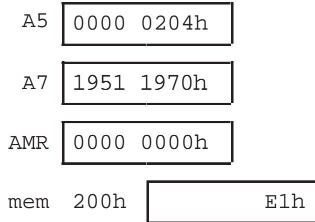
```
LDW .D1    *A10,B1
```



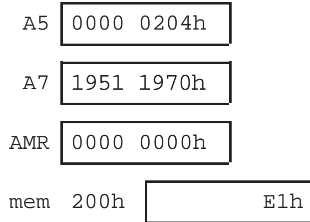
Example 2

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

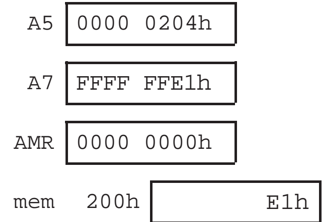
Before LDB



1 cycle after LDB



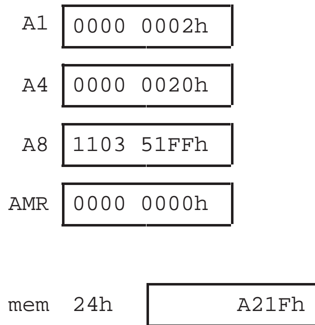
5 cycles after LDB



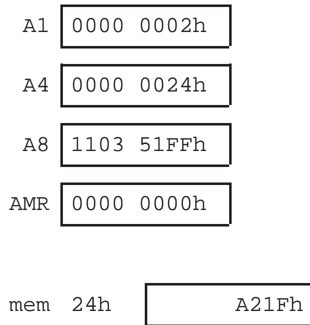
Example 3

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

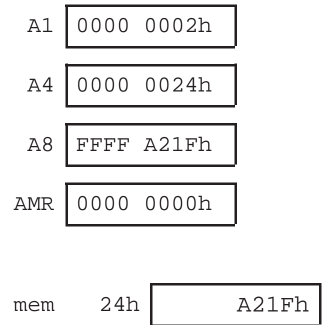
Before LDH



1 cycle after LDH



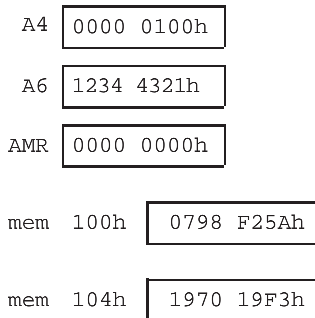
5 cycles after LDH



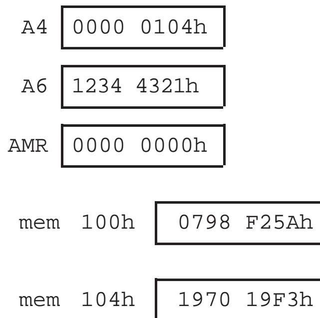
Example 4

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

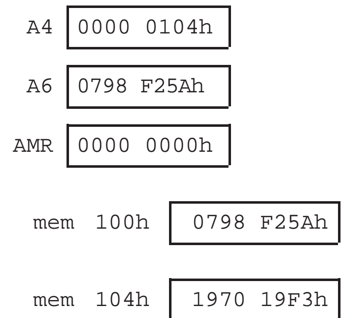
Before LDW



1 cycle after LDW



5 cycles after LDW



Example 5

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

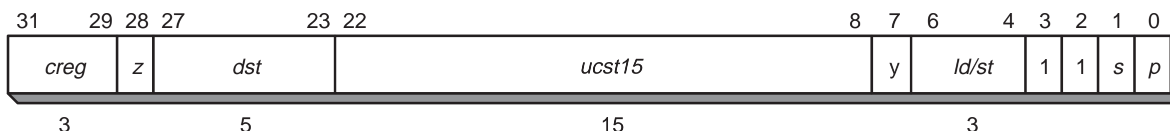
	Before LDW	1 cycle after LDW	5 cycles after LDW
A4	0000 0100h	0000 0104h	0000 0104h
A6	1234 5678h	1234 5678h	0217 6991h
AMR	0000 0000h	0000 0000h	0000 0000h
mem 104h	0217 6991h	0217 6991h	0217 6991h

Syntax

LDB (.unit) *+B14/B15[*ucst15*], *dst*
or
LDH (.unit) *+B14/B15[*ucst15*], *dst*
or
LDW (.unit) *+B14/B15[*ucst15*], *dst*
or
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 3–15 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.

The offset, *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 *baseR*. Subtraction is not supported. 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. You must type either brackets or parentheses around the specified offset if you use the optional offset parameter.

Word and halfword addresses must be aligned on word (two LSBs are 0) and halfword (LSB is 0) boundaries, respectively.

Table 3–15. 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).

Pipeline

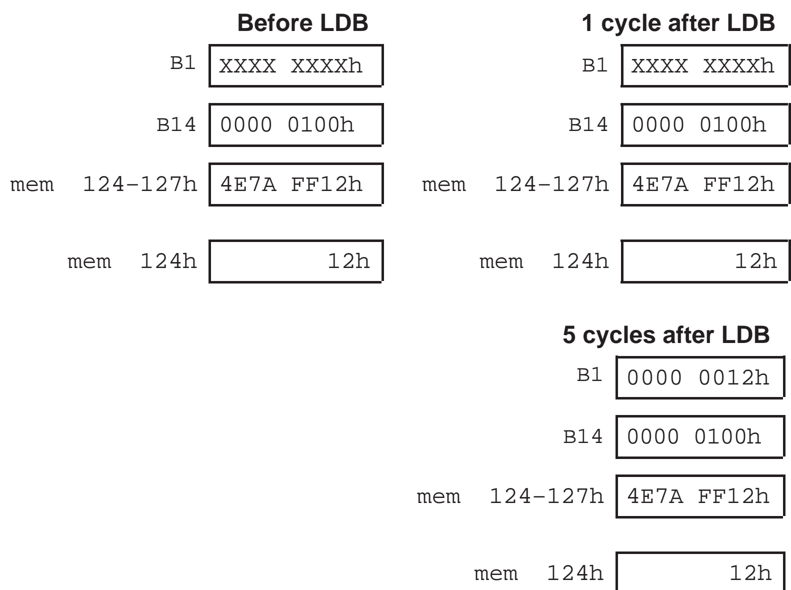
Pipeline Stage	E1	E2	E3	E4	E5
Read	*B14 / B15*				
Written					*dst*
Unit in use	.D2				

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*	cst5 xuint uint	.L1, .L2	1101010

Opcode

31–29	28	27–23	22–18	17–13	12	11–5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1/cst*	x	*op*	1	1	0	*s*	*p*
3		5	5	5		7					

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

Pipeline

Pipeline Stage	E1
Read	*src1, src2*
Written	*dst*
Unit in use	.L

Instruction Type

Single-cycle

Delay Slots

0

Example

```
LMBD .L1   A1,A2,A3
```

	Before instruction		1 cycle after instruction
A1	0000 0001h	A1	0000 0001h
A2	009E 3A81h	A2	009E 3A81h
A3	XXXX XXXXh	A3	0000 0008h

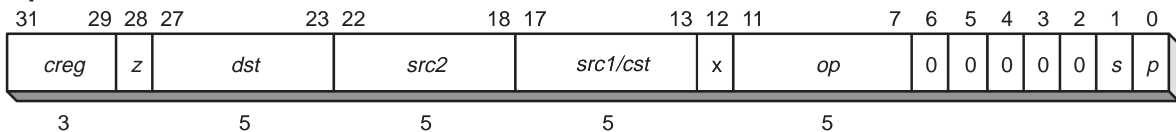
Syntax

MPY (.unit) *src1*, *src2*, *dst*
or
MPYU (.unit) *src1, src2, dst*
or
MPYUS (.unit) *src1, src2, dst*
or
MPYSU (.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*	scst5 xslsb16 sint	.M1, .M2	11000	MPY
src1 *src2* *dst*	scst5 xulsb16 sint	.M1, .M2	11110	MPYSU

Opcode



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

Pipeline

Pipeline Stage	E1	E2
Read	*src1, src2*	
Written		*dst*
Unit in use	.M	

Instruction Type Multiply (16 × 16)

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

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

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

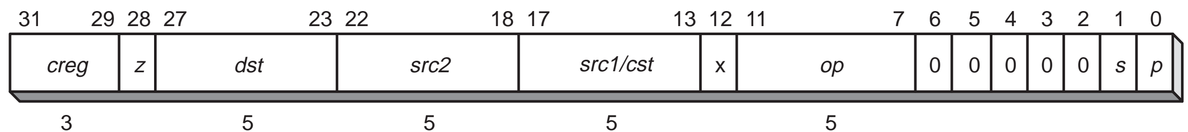
Syntax

MPYH (.unit) *src1*, *src2*, *dst*
or
MPYHU (.unit) *src1*, *src2*, *dst*
or
MPYHUS (.unit) *src1*, *src2*, *dst*
or
MPYHSU (.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

Opcode



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*) × msb16(*src2*) → *dst*
else nop

Pipeline

Pipeline Stage	E1	E2
Read	*src1, src2*	
Written		*dst*
Unit in use	.M	

Instruction Type

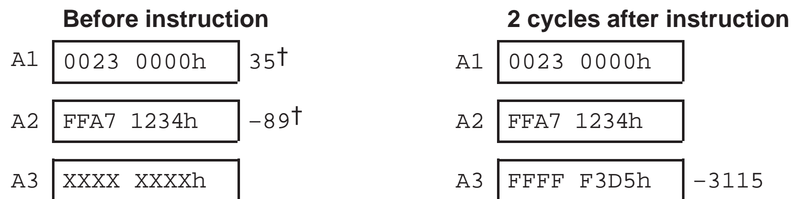
Multiply (16 × 16)

Delay Slots

1

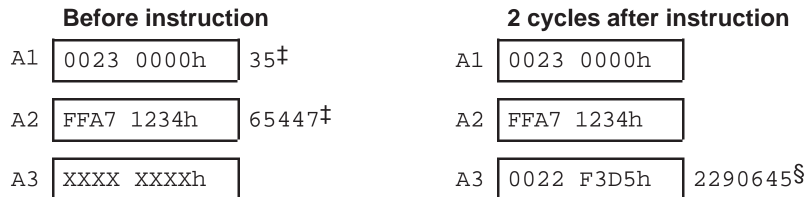
Example 1

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



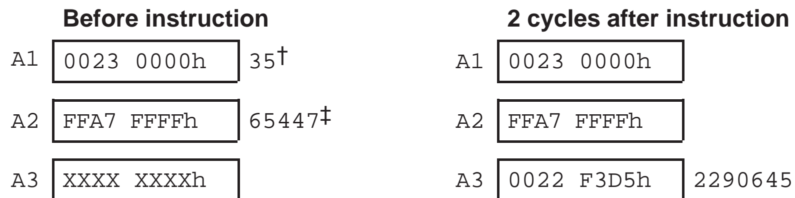
Example 2

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



Example 3

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



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

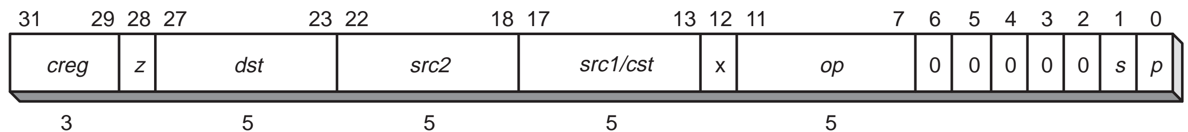
Syntax

MPYHL (.unit) *src1*, *src2*, *dst*
or
MPYHLU (.unit) *src1*, *src2*, *dst*
or
MPYHULS (.unit) *src1*, *src2*, *dst*
or
MPYHSLU (.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

Opcode



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*) × lsb16(*src2*) → *dst*
else nop

Pipeline

Pipeline Stage	E1	E2
Read	*src1, src2*	
Written		*dst*
Unit in use	.M	

Instruction Type

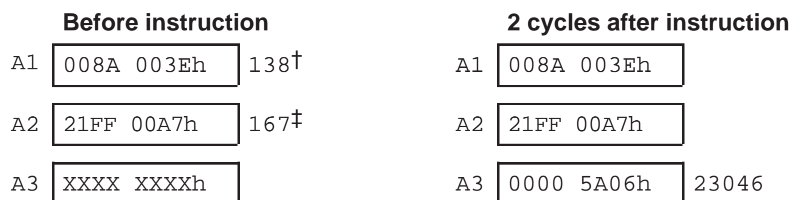
Multiply (16 × 16)

Delay Slots

1

Example

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



† Signed 16-MSB integer
‡ Signed 16-LSB integer

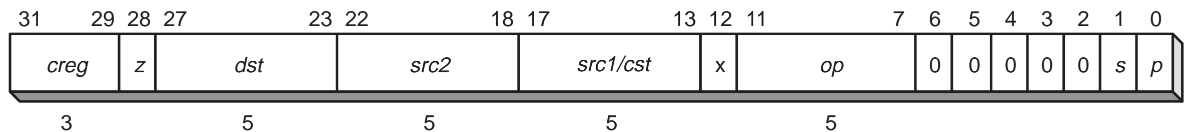
Syntax

MPYLH (.unit) *src1*, *src2*, *dst*
or
MPYLHU (.unit) *src1*, *src2*, *dst*
or
MPYLUHS (.unit) *src1*, *src2*, *dst*
or
MPYLSHU (.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

Opcode



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*) × msb16(*src2*) → *dst*
else nop

Pipeline

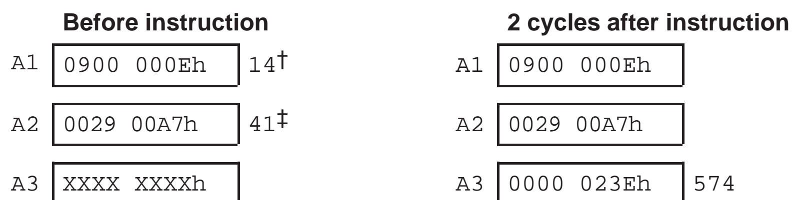
Pipeline Stage	E1	E2
Read	*src1, src2*	
Written		*dst*
Unit in use	.M	

Instruction Type Multiply (16 × 16)

Delay Slots 1

Example

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



† Signed 16-LSB integer
‡ Signed 16-MSB integer

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

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

Opcode map field used...	For operand type...	Unit	Opfield
src *dst*	xsint sint	.L1, .L2	0000010
src *dst*	sint sint	.D1, .D2	010010
src *dst*	slong slong	.L1, .L2	0100001
src *dst*	xsint sint	.S1, .S2	000110

Opcode See **ADD** instruction.

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

Opcode

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 3–16.

Table 3–16. 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
FADCR†	Floating-point adder configuration	10010	R, W
FAUCR†	Floating-point auxiliary configuration	10011	R, W
FMCR†	Floating-point multiplier configuration	10100	R, W

Note: R = Readable by the **MVC** instruction
W = Writeable by the **MVC** instruction
† TMSC320C67x only

Execution

if (cond) *src* → *dst*
else nop

Note:

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

Pipeline

Pipeline Stage	E1
Read	*src2*
Written	*dst*
Unit in use	.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:

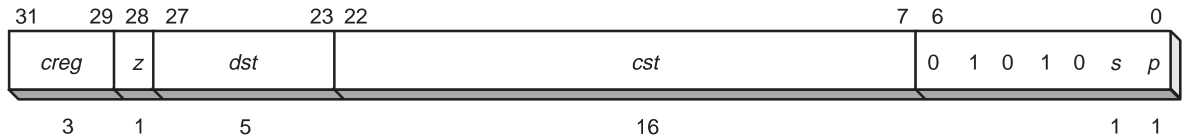
The six MSBs of the AMR are reserved and therefore are not written to.

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

.unit = .S1 or .S2

Opcode map field used...	For operand type...	Unit
cst *dst*	scst16 sint	.S1, .S2

Opcode



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

Execution

if (cond) *scst*16 → *dst*
else nop

Pipeline

Pipeline Stage	E1
Read	
Written	*dst*
Unit in use	.S

Instruction Type Single-cycle

Delay Slots 0

Note:

To load 32-bit constants, such as 0x1234 5678, use the following pair of instructions:

```
MVK   0x5678
MVKLH 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 map field used...	For operand type...	Unit
cst *dst*	uscst16 sint	.S1, .S2

Opcode

31–29	28	27–23	22–7	6					1	0
creg	*z*	*dst*	*cst*	1	1	0	1	0	*s*	*p*
3	1	5	16						1	1

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

Pipeline

Pipeline Stage	E1
Read	
Written	*dst*
Unit in use	.S

Instruction Type Single-cycle

Delay Slots 0

Note:

To load 32-bit constants, such as 0x1234 5678, use the following pair of instructions:

```
MVK    0x5678
MVKLH  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	A1	0A32 7634h

Example 2

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

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

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

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

Opcode map field used...	For operand type...	Unit	Opfield
src *dst*	xsint sint	.S1, .S2	010110
src *dst*	xsint sint	.L1, .L2	0000110
src *dst*	slong slong	.L1, .L2	0100100

Opcode See **SUB** instruction.

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

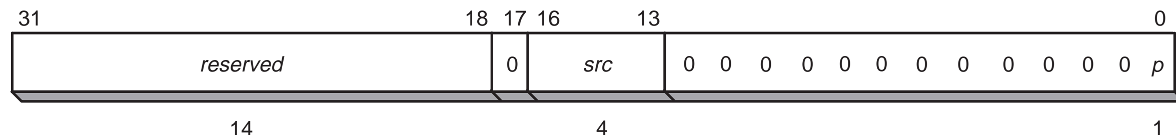
Instruction Type Single-cycle

Delay Slots 0

Syntax **NOP** [*count*]

Opcode map field used...	For operand type...	Unit
src	ucst4	none

Opcode



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.

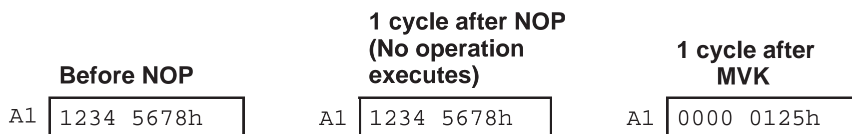
Execution No operation for *count* cycles

Instruction Type **NOP**

Delay Slots 0

Example 1

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



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	A1	0000 0004h
A2	0000 0003h	A2	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

Opcode

31–29	28	27–23	22–18	17–13	12	11–5	4	3	2	1	0
creg	*z*	*dst*	*src2*	0 0 0 0 0	x	*op*	1	1	0	*s*	*p*
3		5	5	5		7					

Description The number of redundant sign bits of *src2* is placed in *dst*. Several examples are shown in the following diagram.

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

src2

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
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

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

src2

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
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

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

src2

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	1	1	1	1	1	1	1	0

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

src2

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	1	1	1	1	1	1	1	1

Execution

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

Instruction Type Single-cycle

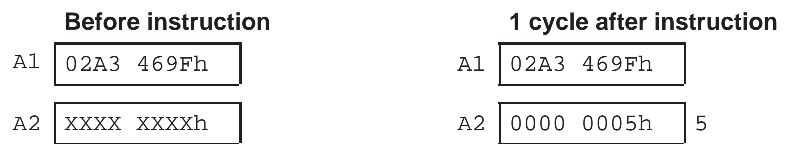
Pipeline

Pipeline Stage	E1
Read	*src2*
Written	*dst*
Unit in use	.L

Delay Slots 0

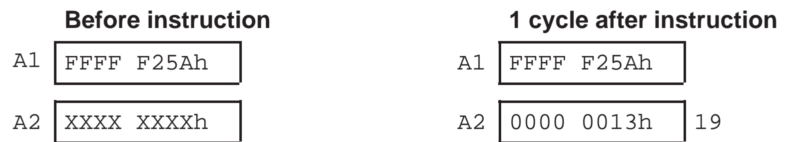
Example 1

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



Example 2

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



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

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

Opcode map field used...	For operand type...	Unit	Opfield
src *dst*	xuint uint	.L1, .L2	1101110
src *dst*	xuint uint	.S1, .S2	001010

Opcode See **XOR** instruction.

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

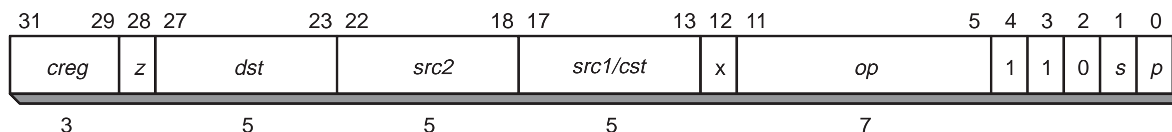
OR (.unit) *src1*, *src2*, *dst*

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

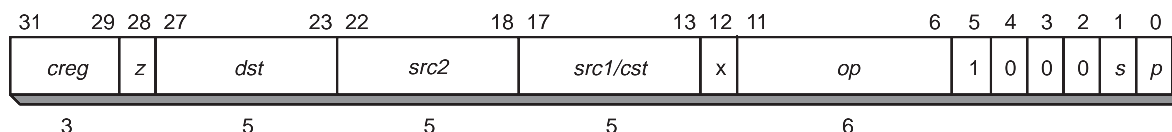
Opcode map field used...	For operand type...	Unit	Opfield
src1 *src2* *dst*	uint xuint uint	.L1, .L2	1111111
src1 *src2* *dst*	scst5 xuint uint	.L1, .L2	1111110
src1 *src2* *dst*	uint xuint uint	.S1, .S2	011011
src1 *src2* *dst*	scst5 xuint uint	.S1, .S2	011010

Opcode

.L unit form:



.S unit form:



Description

A bitwise 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* or *src2* → *dst*
else nop

Pipeline

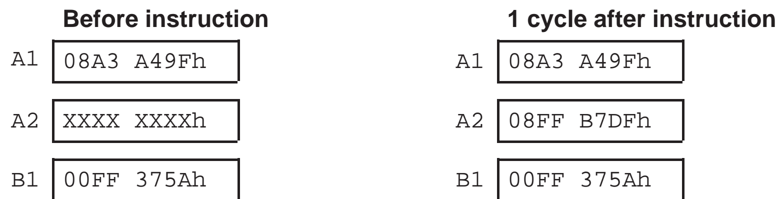
Pipeline Stage	E1
Read	*src1, src2*
Written	*dst*
Unit in use	.L or .S

Instruction Type Single-cycle

Delay Slots 0

Example 1

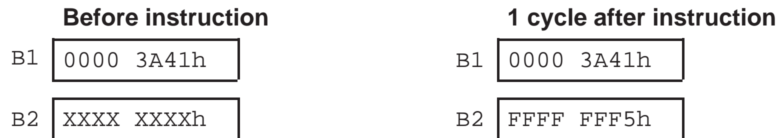
```
OR .L1X    A1,B1,A2
```



	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

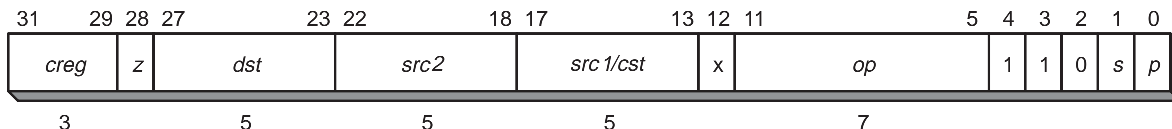
Syntax

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*	scst5 xsint sint	.L1, .L2	0010010
src1 *src2* *dst*	scst5 slong slong	.L1, .L2	0110000

Opcode



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

Pipeline

Pipeline Stage	E1
Read	*src1, src2*
Written	*dst*
Unit in use	.L

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 .L1X B2,A5:A4,A7:A6`

Before instruction

A5:A4	0000 0000h	7C83 39B1h	1922644401†
A7:A6	XXXX XXXXh	XXXX XXXXh	
B2	112A 3FA2h		287981474
CSR	0001 0100h		

1 cycle after instruction

A5:A4	0000 0000h	7C83 39B1h	
A7:A6	0000 0000h	8DAD 7953h	2376956243†
B2	112A 3FA2h		
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 40-bit (long) integer

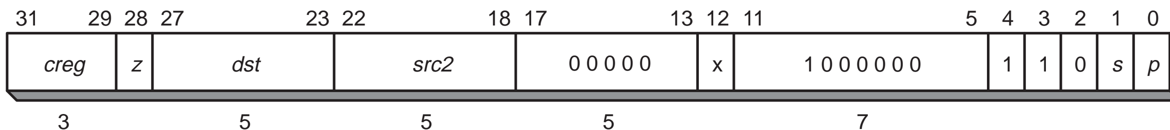
Syntax

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

.unit = .L1 or .L2

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

Opcode



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) → dst
             else if (src2 < –2^31)
                  –2^31 → dst
             else src2_31..0 → dst
             }
else         nop
```

Pipeline

Pipeline Stage	E1
Read	*src2*
Written	*dst*
Unit in use	.L

Instruction Type

Single-cycle

Delay Slots

0

Example 1

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

	Before instruction			1 cycle after instruction			2 cycles after instruction		
A1:A0	0000 001Fh	3413 539Ah	A1:A0	0000 001Fh	3413 539Ah	A1:A0	0000 001Fh	3413 539Ah	
A2	XXXX XXXXh		A2	7FFF FFFFh		A2	7FFF FFFFh		
CSR	0001 0100h		CSR	0001 0100h		CSR	0001 0300h		Saturated

Example 2

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

	Before instruction			1 cycle after instruction			2 cycles after instruction		
B1:B0	0000 0000h	A190 7321h	B1:B0	0000 0000h	A190 7321h	B1:B0	0000 0000h	A190 7321h	
B5	XXXX XXXXh		B5	7FFF FFFFh		B5	7FFF FFFFh		
CSR	0001 0100h		CSR	0001 0100h		CSR	0001 0300h		Saturated

Example 3

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

	Before instruction			1 cycle after instruction			2 cycles after instruction		
B1:B0	0000 00FFh	A190 7321h	B1:B0	0000 00FFh	A190 7321h	B1:B0	0000 00FFh	A190 7321h	
B5	XXXX XXXXh		B5	A190 7321h		B5	A190 7321h		
CSR	0001 0100h		CSR	0001 0100h		CSR	0001 0100h		Not saturated

Syntax

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

.unit = .S1 or .S2

Opcode map field used...	For operand type...	Unit
src2 *csta* *cstb* *dst*	uint ucst5 ucst5 uint	.S1, .S2
src2 *src1* *dst*	xuint uint uint	.S1, .S2

Opcode

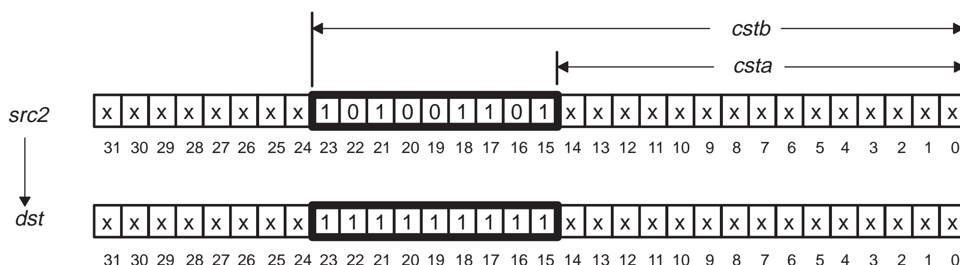
Constant form:

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

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. Only the ten LSBs are valid for the register version of the instruction. If any of the 22 MSBs are non-zero, the result is invalid.



Execution If the constant form is used:

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

If the register form is used:

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

Pipeline

Pipeline Stage	E1
Read	*src1, src2*
Written	*dst*
Unit in use	.S

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 .S2    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

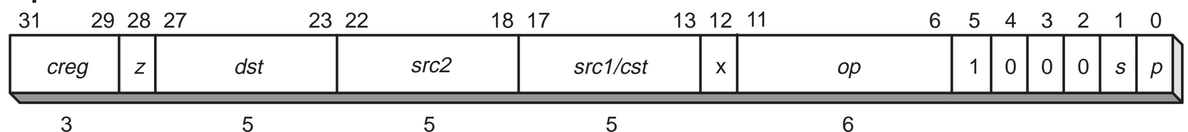
Syntax

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*	xuint uint ulong	.S1, .S2	010011
src2 *src1* *dst*	xsint ucst5 sint	.S1, .S2	110010
src2 *src1* *dst*	slong ucst5 slong	.S1, .S2	110000
src2 *src1* *dst*	xuint ucst5 ulong	.S1, .S2	010010

Opcode



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 valid for the register version of the instruction. If any of the 26 MSBs are non-zero, the result is invalid.

Execution

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

Pipeline

Pipeline Stage	E1
Read	*src1, src2*
Written	*dst*
Unit in use	.S

Instruction Type Single-cycle

Delay Slots 0

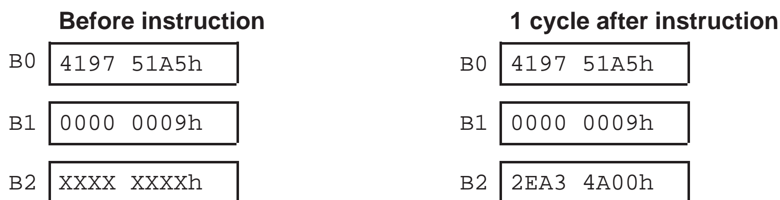
Example 1

```
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

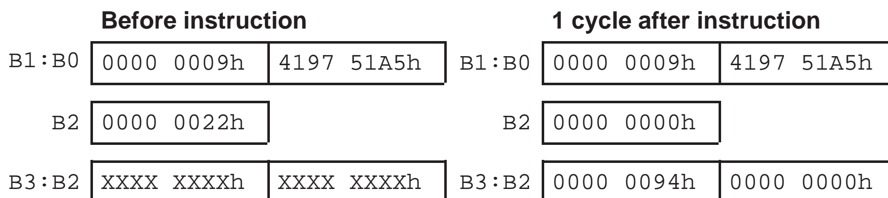
```
SHL .S2    B0,B1,B2
```



	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

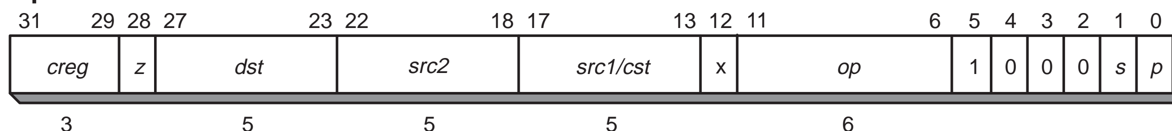
Syntax

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 ucst5 sint	.S1, .S2	110110
src2 *src1* *dst*	slong ucst5 slong	.S1, .S2	110100

Opcode



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 valid for the register version of the instruction. If any of the 26 MSBs are non-zero, the result is invalid.

Execution

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

Pipeline

Pipeline Stage	E1
Read	*src1, src2*
Written	*dst*
Unit in use	.S

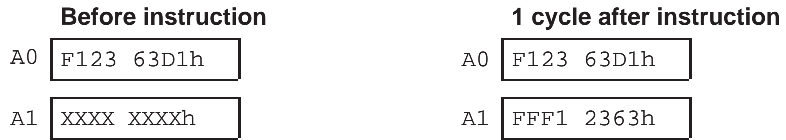
Instruction Type

Single-cycle

Delay Slots 0

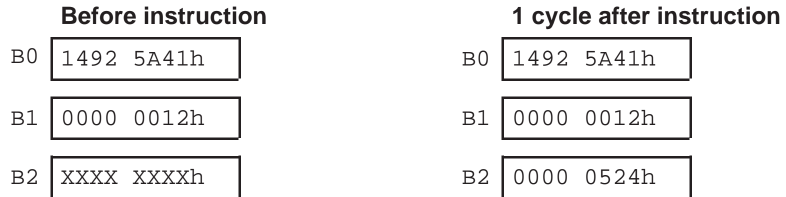
Example 1

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



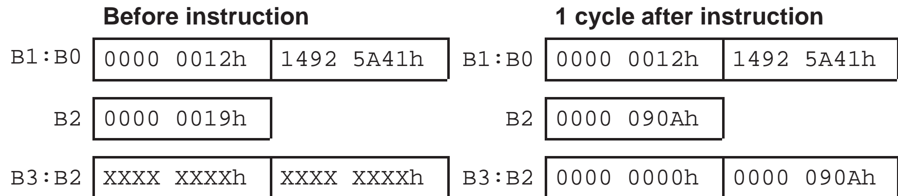
Example 2

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



Example 3

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

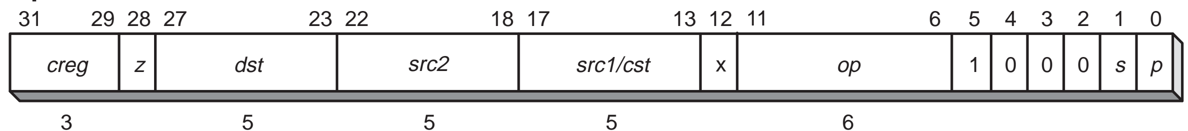


Syntax **SHRU** (.unit) *src2*, *src1*, *dst*

.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 ucst5 uint	.S1, .S2	100110
src2 *src1* *dst*	ulong ucst5 ulong	.S1, .S2	100100

Opcode



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 valid for the register version of the instruction. If any of the 26 MSBs are non-zero, the result is invalid.

Execution

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

Pipeline

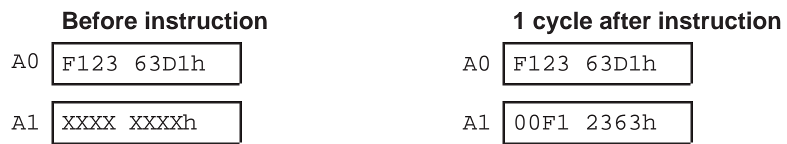
Pipeline Stage	E1
Read	*src1, src2*
Written	*dst*
Unit in use	.S

Instruction Type Single-cycle

Delay Slots 0

Example

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



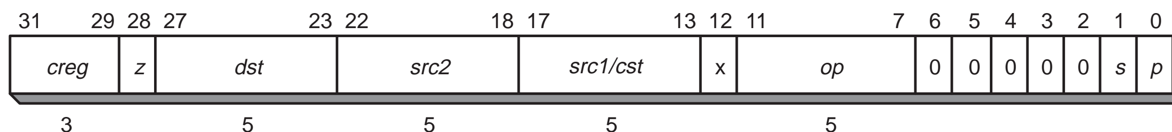
Syntax

SMPY (.unit) *src1*, *src2*, *dst*
or
SMPYHL (.unit) *src1*, *src2*, *dst*
or
SMPYLH (.unit) *src1*, *src2*, *dst*
or
SMPYH (.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

Opcode



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
```

Pipeline

Pipeline Stage	E1	E2
Read	*src1, src2*	
Written		*dst*
Unit in use	.M	

Instruction Type Single-cycle (16 × 16)

Delay Slots 1

Example 1

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

	Before instruction			2 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

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

† Signed 16-MSB integer
‡ Signed 16-LSB integer

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 16-MSB integer
‡ Signed 16-LSB integer

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 ucst5 sint	.S1, .S2	100010

Opcode

Description The *src2* operand is shifted to the left by the *src1* operand. The result is placed in *dst*. When a register is used to specify the shift, the five least significant bits specify the shift amount. Valid values are 0 through 31, and the result of the shift is invalid if the shift amount is greater than 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 0x7FFF FFFF;
              else if (src2 < 0)
                 saturate dst to 0x8000 0000;
              }
else          nop
```

Pipeline

Pipeline Stage	E1
Read	*src1, src2*
Written	*dst*
Unit in use	.S

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

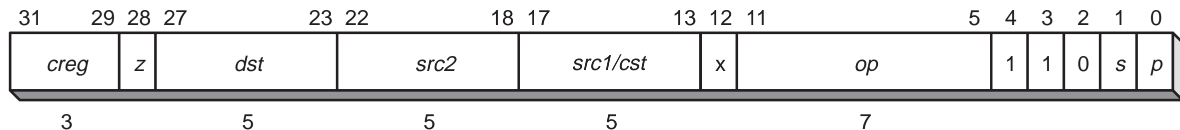
Syntax

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*	scst5 xsint sint	.L1, .L2	0001110
src1 *src2* *dst*	scst5 slong slong	.L1, .L2	0101100

Opcode



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

Pipeline

Pipeline Stage	E1
Read	*src1, src2*
Written	*dst*
Unit in use	.L

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 .L1   A0,A1,A2
```

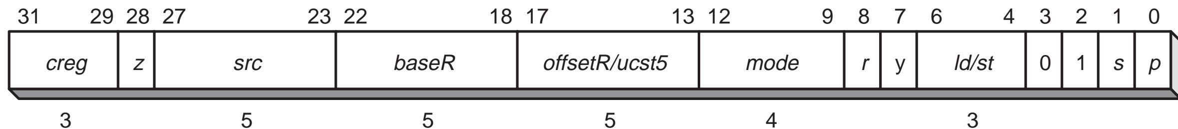
	Before instruction			1 cycle after instruction			2 cycles after instruction	
A0	4367 71F2h	1130852850	A0	4367 71F2h		A0	4367 71F2h	
A1	5A2E 51A3h	1512984995	A1	5A2E 51A3h		A1	5A2E 51A3h	
A2	XXXX XXXXh		A2	E939 204Fh	−382132145	A2	E939 204Fh	
CSR	0001 0100h		CSR	0001 0100h		CSR	0001 0100h	Not saturated

Syntax

STB (.unit) *src*,*+*baseR[offsetR]*
or
STH (.unit) *src*, *+*baseR[offsetR]*
or
STW (.unit) *src*, *+*baseR[offsetR]*

.unit = .D1 or .D2

Opcode



Description

Each of these instructions performs a store to memory from a general-purpose register (*src*). Table 3–17 summarizes the data types supported by stores. Table 3–18 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*/*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 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 2.6.1 on page 2-9).

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. The *r* bit should be set to zero.

Table 3–17. 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 3–18. 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. You must type either brackets or parentheses around the specified offset if you use the optional offset parameter.

Word and halfword addresses must be aligned on word (two LSBs are 0) and halfword (LSB is 0) boundaries, respectively.

Execution

if (cond) *src* → mem
else nop

Instruction Type Store

Pipeline

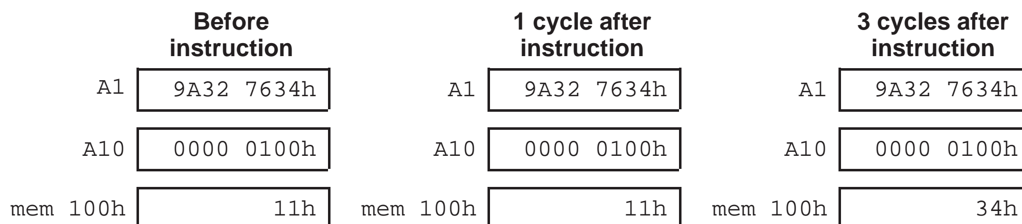
Pipeline Stage	E1	E2	E3
Read	*baseR, offsetR src*		
Written	*baseR*		
Unit in use	.D2		

Delay Slots 0

For more information on delay slots for a store, see Chapter 5, *TMS320C62x Pipeline,* and Chapter 6, *TMS320C67x Pipeline*.

Example 1

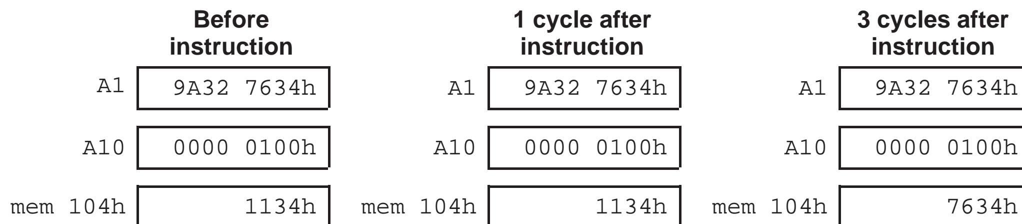
```
STB .D1    A1,*A10
```



	Before instruction	1 cycle after instruction	3 cycles after instruction
A1	9A32 7634h	9A32 7634h	9A32 7634h
A10	0000 0100h	0000 0100h	0000 0100h
mem 100h	11h	11h	34h

Example 2

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



	Before instruction	1 cycle after instruction	3 cycles after instruction
A1	9A32 7634h	9A32 7634h	9A32 7634h
A10	0000 0100h	0000 0100h	0000 0100h
mem 104h	1134h	1134h	7634h

Example 3

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

	Before instruction	1 cycle after instruction	3 cycles after instruction
A1	9A32 7634h	9A32 7634h	9A32 7634h
A10	0000 0100h	0000 0104h	0000 0104h
mem 100h	1111 1134h	1111 1134h	1111 1134h
mem 104h	0000 1111h	0000 1111h	9A32 7634h

Example 4

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

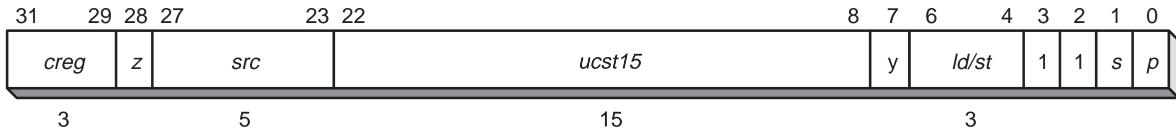
	Before instruction	1 cycle after instruction	3 cycles after instruction
A1	9A32 2634h	9A32 2634h	9A32 2634h
A10	0000 0100h	0000 00F8h	0000 00F8h
A11	0000 0004h	0000 0004h	0000 0004h
mem F8h	0000h	0000h	0000h
mem 100h	0000	0000h	2634h

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 3–19 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 *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. You must type either brackets or parentheses around the specified offset if you use the optional offset parameter.

Word and halfword addresses must be aligned on word (two LSBs are 0) and halfword (LSB is 0) boundaries, respectively.

Table 3–19. 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

Pipeline

Pipeline Stage	E1	E2	E3
Read	B14/B15, *src*		
Written			
Unit in use	.D2		

Instruction Type Store

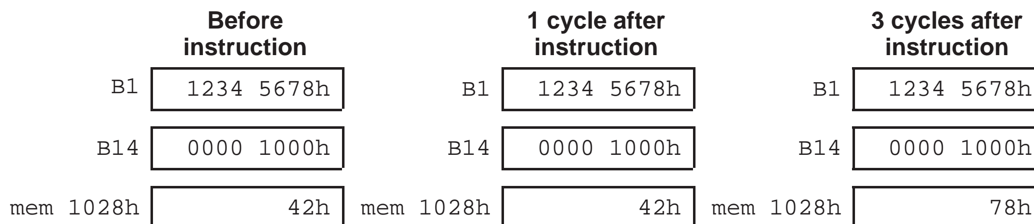
Delay Slots 0

Note:

This instruction executes only on the .D2 unit.

Example

```
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*

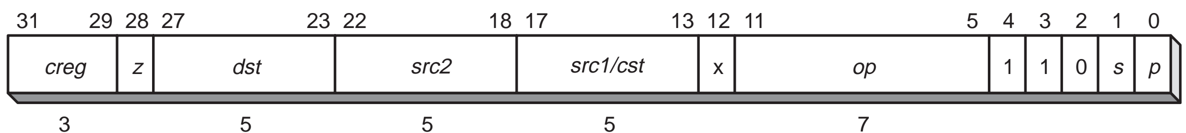
.unit = .L1, .L2, .S1, .S2

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*	scst5 xsint sint	.L1, .L2	0000110	SUB
src1 *src2* *dst*	scst5 slong slong	.L1, .L2	010010 0	SUB
src1 *src2* *dst*	sint xsint sint	.S1, .S2	010111	SUB
src1 *src2* *dst*	scst5 xsint sint	.S1, .S2	010110	SUB

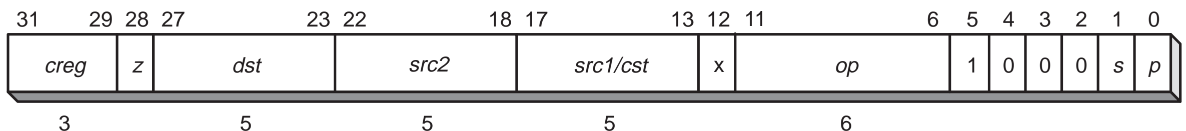
Opcode map field used...	For operand type...	Unit	Opfield	Mnemonic
src2 *src1* *dst*	sint sint sint	.D1, .D2	010001	SUB
src2 *src1* *dst*	sint ucst*5* sint	.D1, .D2	010011	SUB

Opcode

.L unit form:



.S unit form:



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

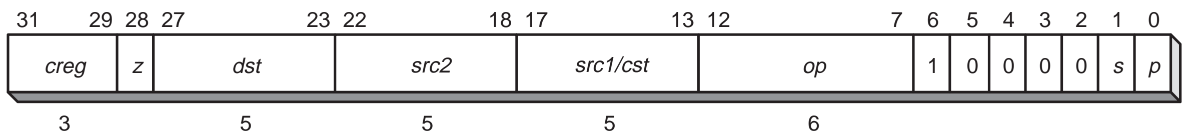
src2 is subtracted from *src1*. The result is placed in *dst*.

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

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

Opcode

.D unit form:



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 *src1*, *scst5*, *dst* is 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.

Pipeline

Pipeline Stage	E1
Read	*src1, src2*
Written	*dst*
Unit in use	.L, .S, or .D

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
```

	Before instruction				1 cycle after instruction		
A1	0000 325Ah		12810†	A1	0000 325Ah		
A2	FFFF FF12h		4294967058†	A2	FFFF FF12h		
A5:A4	XXXX XXXXh	XXXX XXXXh		A5:A4	0000 00FFh	0000 3348h	−4294954168‡

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

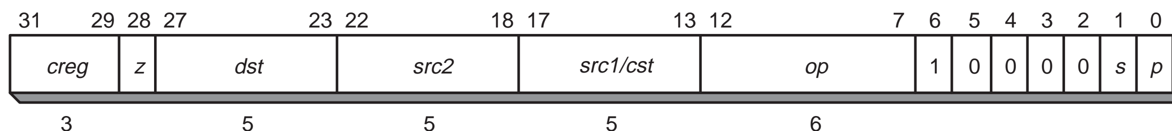
Syntax

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 ucst5 sint	.D1, .D2	Byte: 110011 Halfword: 110111 Word: 111011

Opcode



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 2.6.1 on page 2-9). *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

Pipeline

Pipeline Stage	E1
Read	*src1, src2*
Written	*dst*
Unit in use	.D

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

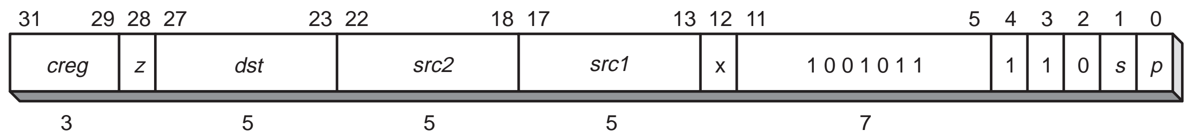
Syntax

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

.unit = .L1 or .L2

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

Opcode



Description

Subtract *src2* from *src1*. If result is greather than or equal to 0, left shift result by 1, add 1 to it, and place it in *dst*. If result is less than 0, left shift *scr1* by 1, and place it 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

Pipeline

Pipeline Stage	E1
Read	*src1, src2*
Written	*dst*
Unit in use	.L

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 47E5h	18405
A1	0001 F63Fh	128575	A1	0001 F63Fh	

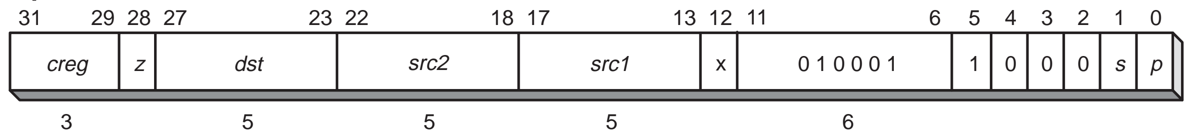
Syntax

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

.unit = .S1 or .S2

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

Opcode



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
```

Pipeline

Pipeline Stage	E1
Read	*src1, src2*
Written	*dst*
Unit in use	.S

Instruction Type

Single-cycle

Delay Slots

0

Example

```
SUB2 .S2X  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	0019 E8D7h	25†	−5929‡

† Signed 16-MSB integer
‡ Signed 16-LSB integer

Syntax

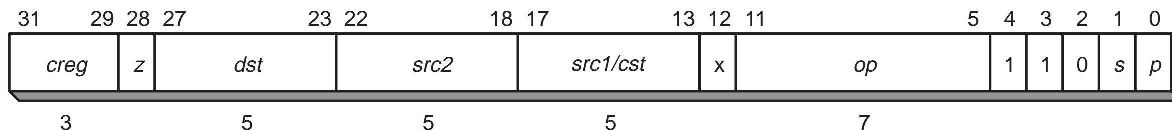
XOR (.unit) *src2*, *src1*, *dst*

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

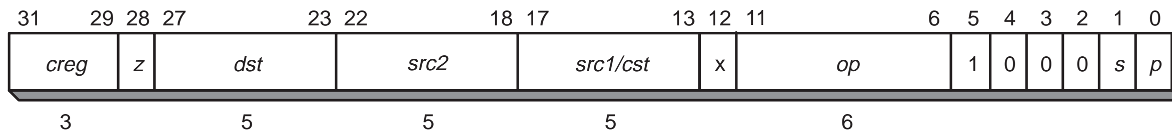
Opcode map field used...	For operand type...	Unit	Opfield
src1 *src2* *dst*	uint xuint uint	.L1, .L2	1101111
src1 *src2* *dst*	scst5 xuint uint	.L1, .L2	1101110
src1 *src2* *dst*	uint xuint uint	.S1, .S2	001011
src1 *src2* *dst*	scst5 xuint uint	.S1, .S2	001010

Opcode

.L unit form:



.S unit form:



Description

A bitwise exclusive-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* xor *src2* → *dst*
else nop

Pipeline

Pipeline Stage	E1
Read	*src1, src2*
Written	*dst*
Unit in use	.L or .S

Instruction Type Single-cycle

Delay Slots 0

Example 1

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

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

Example 2

```
XOR .L2    B1,0dh,B2
```

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

Syntax

ZERO (.unit) *dst*

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

Opcode map field used...	For operand type...	Unit	Opfield
dst	sint	.L1, .L2	0010111
dst	sint	.D1, .D2	010001
dst	sint	.S1, .S2	010111
dst	slong	.L1, .L2	0110111

Description

This is a pseudo operation used to fill the *dst* register with 0s 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

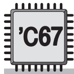
Instruction Type

Single-cycle

Delay Slots

0

Chapter 4



TMS320C67x Floating-Point Instruction Set

The ’C67x floating-point DSP uses all of the instructions available to the ’C62x, but it also uses other instructions that are specific to the ’C67x. These specific instructions are for 32-bit integer multiply, doubleword load, and floating-point operations, including addition, subtraction, and multiplication. This chapter describes these ’C67x-specific instructions.

Instructions that are common to both the ’C62x and ’C67x are described in Chapter 3.

Topic **Page**

4.1 Instruction Operation and Execution Notations 4-2

4.2 Mapping Between Instructions and Functional Units 4-4

4.3 Overview of IEEE Standard Single- and Double-Precision Formats 4-6

4.4 Delay Slots 4-11

4.5 TMS320C67x Instruction Constraints 4-12

4.6 Individual Instruction Descriptions 4-15

4.1 Instruction Operation and Execution Notations

Table 4–1 explains the symbols used in the floating-point instruction descriptions.

Table 4–1. Floating-Point Instruction Operation and Execution Notations

Symbol	Meaning
abs(x)	Absolute value of x
cond	Check for either *creg* equal to 0 or *creg* not equal to 0
creg	3-bit field specifying a conditional register
cstn	n-bit constant field (for example, cst5)
dp	Double-precision floating-point register value
dp(x)	Convert x to dp
dst_h	msb32 of dst
dst_l	lsb32 of dst
int	32-bit integer value
int(x)	Convert x to integer
lsbn or LSBn	n least significant bits (for example, lsb32)
msbn or MSBn	n most significant bits (for example, msb32)
nop	No operation
R	Any general-purpose register
rcp(x)	Reciprocal approximation of x
sdint	Signed 64-bit integer value (two registers)
sint	Signed 32-bit integer value
sp	Single-precision floating-point register value that can optionally use cross path
sp(x)	Convert x to sp
sqrcp(x)	Square root of reciprocal approximation of x
src1_h	msb32 of src1
src1_l	lsb32 of src1
src2_h	msb32 of src2
src2_l	lsb32 of src2

Table 4–1. Floating-Point Instruction Operation and Execution Notations (Continued)

Symbol	Meaning
ucstn	n-bit unsigned constant field (for example, ucstn5)
uint	Unsigned 32-bit integer value
dp	Double-precision floating-point register value
xsint	Signed 32-bit integer value that can optionally use cross path
sp	Single-precision floating-point register value
xsp	Single-precision floating-point register value that can optionally use cross path
xuint	Unsigned 32-bit integer value that can optionally use cross path
$\rightarrow$	Assignment
+	Addition
$\times$	Multiplication
–	Subtraction
<<	Shift left

4.2 Mapping Between Instructions and Functional Units

Table 4–2 shows the mapping between instructions and functional units and and Table 4–3 shows the mapping between functional units and instructions.

Table 4–2. Instruction to Functional Unit Mapping

.L Unit	.M Unit	.S Unit	.D Unit
ADDDP	MPYDP	ABSDP	ADDAD
ADDSP	MPYI	ABSSP	LDDW
DPINT	MPYID	CMPEQDP	
DPSP	MPYSP	CMPEQSP	
INTDP		CMPGTDP	
INTDPU		CMPGTSP	
INTSP		CMPLTDP	
INTSPU		CMPLTSP	
SPINT		RCPDP	
SPTRUNC		RCPSP	
SUBDP		RSQRDP	
SUBSP		RSQRSP	
		SPDP	

Table 4–3. Functional Unit to Instruction Mapping

	’C67x Functional Units				
Instruction	**.L Unit**	**.M Unit**	**.S Unit**	**.D Unit**	**Type**
ABSDP			✓		2-cycle DP
ABSSP			✓		Single cycle
ADDAD				✓	Single cycle
ADDDP	✓				ADDDP/ SUBDP
ADDSP	✓				Four cycle
CMPEQDP			✓		DP compare
CMPEQSP			✓		Single cycle
CMPGTDP			✓		DPcompare
CMPGTSP			✓		Single cycle

Table 4–3. Functional Unit to Instruction Mapping (Continued)

Instruction	'C67x Functional Units .L Unit	.M Unit	.S Unit	.D Unit	Type
CMPLTDP			✓		DP compare
CMPLTSP			✓		Single cycle
DPINT	✓				4-cycle
DPSP	✓				4-cycle
DPTRUNC	✓				4-cycle
INTDP	✓				INTDP
INTDPU	✓				INTDP
INTSP	✓				4-cycle
INTSPU	✓				4-cycle
LDDW				✓	Load
MPYDP		✓			MPYDP
MPYI		✓			MPYI
MPYID		✓			MPYID
MPYSP		✓			4-cycle
RCPDP			✓		2-cycle DP
RCPSP			✓		Single cycle
RSQRDP			✓		2-cycle DP
RSQRSP			✓		Single cycle
SPDP			✓		2-cycle DP
SPINT	✓				4-cycle
SPTRUNC	✓				4-cycle
SUBDP	✓				ADDDP/ SUBDP
SUBSP	✓				4-cycle

4.3 Overview of IEEE Standard Single- and Double-Precision Formats

Floating-point operands are classified as single-precision (SP) and double-precision (DP). Single-precision floating-point values are 32-bit values stored in a single register. Double-precision floating-point values are 64-bit values stored in a register pair. The register pair consists of consecutive even and odd registers from the same register file. The least significant 32 bits are loaded into the even register. The most significant 32 bits containing the sign bit and exponent are loaded into the next register (which is always the odd register). The register pair syntax places the odd register first, followed by a colon, then the even register (that is, A1:A0, B1:B0, A3:A2, B3:B2, etc.).

Instructions that use DP sources fall in two categories: instructions that read the upper and lower 32-bit words on separate cycles, and instructions that read both 32-bit words on the same cycle. All instructions that produce a double-precision result write the low 32-bit word one cycle before writing the high 32-bit word. If an instruction that writes a DP result is followed by an instruction that uses the result as its DP source and it reads the upper and lower words on separate cycles, then the second instruction can be scheduled on the same cycle that the high 32-bit word of the result is written. The lower result is written on the previous cycle. This is because the second instruction reads the low word of the DP source one cycle before the high word of the DP source.

IEEE floating-point numbers consist of normal numbers, denormalized numbers, NaNs (not a number), and infinity numbers. Denormalized numbers are nonzero numbers that are smaller than the smallest nonzero normal number. Infinity is a value that represents an infinite floating-point number. NaN values represent results for invalid operations, such as (+infinity + (–infinity)).

Normal single-precision values are always accurate to at least six decimal places, sometimes up to nine decimal places. Normal double-precision values are always accurate to at least 15 decimal places, sometimes up to 17 decimal places.

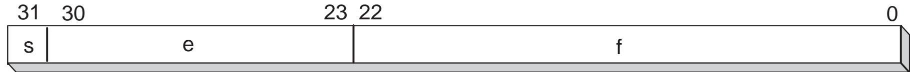
Table 4–4 shows notations used in discussing floating-point numbers.

Table 4–4. IEEE Floating-Point Notations

Symbol	Meaning
s	Sign bit
e	Exponent field
f	Fraction (mantissa) field
x	Can have value of 0 or 1 (don't care)
NaN	Not-a-Number (SNaN or QNaN)
SNaN	Signal NaN
QNaN	Quiet NaN
NaN_out	QNaN with all bits in the f field= 1
Inf	Infinity
LFPN	Largest floating-point number
SFPN	Smallest floating-point number
LDFPN	Largest denormalized floating-point number
SDFPN	Smallest denormalized floating-point number
signed Inf	+infinity or –infinity
signed NaN_out	NaN_out with s = 0 or 1

Figure 4–1 shows the fields of a single-precision floating-point number represented within a 32-bit register.

Figure 4–1. Single-Precision Floating-Point Fields



Legend: s — sign bit (0 positive, 1 negative)
e — 8-bit exponent ($0 < e < 255$)
f — 23-bit fraction
$0 < f < 1*2^{-1} + 1*2^{-2} + ... + 1*2^{-23}$ or
$0 < f < ((2^{23})-1)/(2^{23})$

The floating-point fields represent floating-point numbers within two ranges: normalized (e is between 0 and 255) and denormalized (e is 0). The following formulas define how to translate the s, e, and f fields into a single-precision floating-point number.

Normal

$-1^{s} * 2^{(e-127)} * 1.f \quad 0 < e < 255$

Denormalized (Subnormal)

$-1^{s} * 2^{-126} * 0.f \quad e = 0;\ f \text{ nonzero}$

Table 4–5 shows the s,e, and f values for special single-precision floating-point numbers.

Table 4–5. Special Single-Precision Values

Symbol	Sign (s)	Exponent (e)	Fraction (f)
+0	0	0	0
–0	1	0	0
+Inf	0	255	0
–Inf	1	255	0
NaN	x	255	nonzero
QNaN	x	255	1xx..x
SNaN	x	255	0xx..x and nonzero

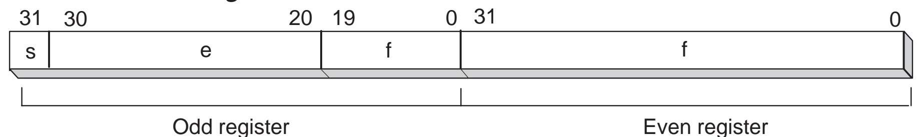
Table 4–6 shows hex and decimal values for some single-precision floating-point numbers.

Table 4–6. Hex and Decimal Representation for Selected Single-Precision Values

Symbol	Hex Value	Decimal Value
NaN_out	0x7FFF FFFF	QNaN
0	0x0000 0000	0.0
–0	0x8000 0000	–0.0
1	0x3F80 0000	1.0
2	0x4000 0000	2.0
LFPN	0x7F7F FFFF	3.40282347e+38
SFPN	0x0080 0000	1.17549435e–38
LDFPN	0x007F FFFF	1.17549421e–38
SDFPN	0x0000 0001	1.40129846e–45

Figure 4–2 shows the fields of a double-precision floating-point number represented within a pair of 32-bit registers.

Figure 4–2. Double-Precision Floating-Point Fields



Legend:
- s — sign bit (0 positive, 1 negative)
- e — 11-bit exponent (0 < e < 2047)
- f — 52-bit fraction
  $0 < f < 1*2^{-1} + 1*2^{-2} + ... + 1*2^{-52}$ or
  $0 < f < ((2^{52})-1)/(2^{52})$

The floating-point fields represent floating-point numbers within two ranges: normalized (e is between 0 and 2047) and denormalized (e is 0). The following formulas define how to translate the s, e, and f fields into a double-precision floating-point number.

Normal

$-1^{s} * 2^{(e-1023)} * 1.f \quad 0 < e < 2047$

Denormalized (Subnormal)

$-1^{s} * 2^{-1022} * 0.f \quad e = 0; \text{ f nonzero}$

Table 4–7 shows the s,e, and f values for special double-precision floating-point numbers.

Table 4–7. Special Double-Precision Values

Symbol	Sign (s)	Exponent (e)	Fraction (f)
+0	0	0	0
–0	1	0	0
+Inf	0	2047	0
–Inf	1	2047	0
NaN	x	2047	nonzero
QNaN	x	2047	1xx..x
SNaN	x	2047	0xx..x and nonzero

Table 4–8 shows hex and decimal values for some double-precision floating-point numbers.

Table 4–8. Hex and Decimal Representation for Selected Double-Precision Values

Symbol	Hex Value	Decimal Value
NaN_out	0x7FFF FFFF FFFF FFFF	QNaN
0	0x0000 0000 0000 0000	0.0
–0	0x8000 0000 0000 0000	–0.0
1	0x3FF0 0000 0000 0000	1.0
2	0x4000 0000 0000 0000	2.0
LFPN	0x7FEF FFFF FFFF FFFF	1.7976931348623157e+308
SFPN	0x0010 0000 0000 0000	2.2250738585072014e–308
LDFPN	0x000F FFFF FFFF FFFF	2.2250738585072009e–308
SDFPN	0x0000 0000 0000 0001	4.9406564584124654e–324

4.4 Delay Slots

The execution of floating-point instructions can be defined in terms of delay slots and functional unit latency. The number of delay slots is equivalent to the number of additional cycles required after the source operands are read for the result to be available for reading. For a single-cycle type instruction, operands are read on cycle *i* and produce a result that can be read on cycle *i + 1*. For a 4-cycle instruction, operands are read on cycle *i* and produce a result that can be read on cycle *i + 4*. Table 4–9 shows the number of delay slots associated with each type of instruction.

The double-precision floating-point addition, subtraction, multiplication, compare, and the 32-bit integer multiply instructions also have a functional unit latency that is greater than 1. The functional unit latency is equivalent to the number of cycles that the instruction uses the functional unit read ports. For example, the **ADDDP** instruction has a functional unit latency of 2. Operands are read on cycle *i* and cycle *i + 1*. Therefore, a new instruction cannot begin until cycle *i + 2*, rather than *i + 1*. **ADDDP** produces a result that can be read on cycle *i + 7*, because it has six delay slots.

Delay slots are equivalent to an execution or result latency. All of the instructions that are common to the 'C62x and 'C67x have a functional unit latency of 1. This means that a new instruction can be started on the functional unit each cycle. Single-cycle throughput is another term for single-cycle functional unit latency.

Table 4–9. Delay Slot and Functional Unit Latency Summary

Instruction Type	Delay Slots	Functional Unit Latency	Read Cycles†	Write Cycles†
Single cycle	0	1	i	i
2-cycle DP	1	1	i	i, i + 1
4-cycle	3	1	i	i + 3
INTDP	4	1	i	i + 3, i + 4
Load	4	1	i	i, i + 4‡
DP compare	1	2	i, i + 1	1 + 1
ADDDP/SUBDP	6	2	i, i + 1	i + 5, i + 6
MPYI	8	4	i, i + 1, 1 + 2, i + 3	i + 8
MPYID	9	4	i, i + 1, 1 + 2, i + 3	i + 8, i + 9
MPYDP	9	4	i, i + 1, 1 + 2, i + 3	i + 8, i + 9

† Cycle i is in the E1 pipeline phase.
‡ A write on cycle i + 4 uses a separate write port from other instructions on the .D unit.

4.5 TMS320C67x Instruction Constraints

If an instruction has a multicycle functional unit latency, it locks the functional unit for the necessary number of cycles. Any new instruction dispatched to that functional unit during this locking period causes undefined results. If an instruction with a multicycle functional unit latency has a condition that is evaluated as false during E1, it still locks the functional unit for subsequent cycles.

An instruction of the following types scheduled on cycle i has the following constraints:

DP compare	No other instruction can use the functional unit on cycles i and i + 1.
ADDDP/SUBDP	No other instruction can use the functional unit on cycles i and i + 1.
MPYI	No other instruction can use the functional unit on cycles i, i + 1, i + 2, and i + 3.
MPYID	No other instruction can use the functional unit on cycles i, i + 1, i + 2, and i + 3.
MPYDP	No other instruction can use the functional unit on cycles i, i + 1, i + 2, and i + 3.

If a cross path is used to read a source in an instruction with a multicycle functional unit latency, you must ensure that no other instructions executing on the same side uses the cross path.

An instruction of the following types scheduled on cycle i using a cross path to read a source, has the following constraints:

DP compare	No other instruction on the same side can used the cross path on cycles i and i + 1.
ADDDP/SUBDP	No other instruction on the same side can use the cross path on cycles i and i + 1.
MPYI	No other instruction on the same side can use the cross path on cycles i, i + 1, i + 2, and i + 3.
MPYID	No other instruction on the same side can use the cross path on cycles i, i + 1, i + 2, and i + 3.
MPYDP	No other instruction on the same side can use the cross path on cycles i, i + 1, i + 2, and i + 3.

Other hazards exist because instructions have varying numbers of delay slots, and need the functional unit read and write ports of varying numbers of cycles. A read or write hazard exists when two instructions on the same functional unit attempt to read or write, respectively, to the register file on the same cycle.

An instruction of the following types scheduled on cycle i has the following constraints:

2-cycle DP	A single-cycle instruction cannot be scheduled on that functional unit on cycle i + 1 due to a write hazard on cycle i + 1. Another 2-cycle DP instruction cannot be scheduled on that functional unit on cycle i + 1 due to a write hazard on cycle i + 1.
4-cycle	A single-cycle instruction cannot be scheduled on that functional unit on cycle i + 3 due to a write hazard on cycle i + 3. A multiply (16 × 16-bit) instruction cannot be scheduled on that functional unit on cycle i + 2 due to a write hazard on cycle i + 3.
INTDP	A single-cycle instruction cannot be scheduled on that functional unit on cycle i + 3 or i + 4 due to a write hazard on cycle i + 3 or i + 4, respectively. An INTDP instruction cannot be scheduled on that functional unit on cycle i + 1 due to a write hazard on cycle i + 1. A 4-cycle instruction cannot be scheduled on that functional unit on cycle i + 1 due to a write hazard on cycle i + 1.
MPYI	A 4-cycle instruction cannot be scheduled on that functional unit on cycle i + 4, i + 5, or i + 6. A MPYDP instruction cannot be scheduled on that functional unit on cycle i + 4, i + 5, or i + 6. A multiply (16 × 16-bit) instruction cannot be scheduled on that functional unit on cycle i + 6 due to a write hazard on cycle i + 7.
MPYID	A 4-cycle instruction cannot be scheduled on that functional unit on cycle i + 4, i + 5, or i + 6. A MPYDP instruction cannot be scheduled on that functional unit on cycles i + 4, i + 5, or i + 6. A multiply (16 × 16-bit) instruction cannot be scheduled on that functional unit on cycle i + 7 or i + 8 due to a write hazard on cycle i + 8 or i + 9, respectively.

MPYDP	A 4-cycle instruction cannot be scheduled on that functional unit on cycle i + 4, i + 5, or i + 6.
	A MPYI instruction cannot be scheduled on that functional unit on cycle i + 4, i + 5, or i + 6.
	A MPYID instruction cannot be scheduled on that functional unit on cycle i + 4, i + 5, or i + 6.
	A multiply (16×16-bit) instruction cannot be scheduled on that functional unit on cycle i + 7 or i + 8 due to a write hazard on cycle i + 8 or i + 9, respectively.
ADDDP/SUBDP	A single-cycle instruction cannot be scheduled on that functional unit on cycle i + 5 or i + 6 due to a write hazard on cycle i + 5 or i + 6, respectively.
	A 4-cycle instruction cannot be scheduled on that functional unit on cycle i + 2 or i + 3 due to a write hazard on cycle i + 5 or i + 6, respectively.
	An INTDP instruction cannot be scheduled on that functional unit on cycle i + 2 or i + 3 due to a write hazard on cycle i + 5 or i + 6, respectively.

All of the above cases deal with double-precision floating-point instructions or the **MPYI** or **MPYID** instructions except for the 4-cycle case. A 4-cycle instruction consists of both single- and double-precision floating-point instructions. Therefore, the 4-cycle case is important for the following single-precision floating-point instructions:

- ADDSP
- SUBSP
- SPINT
- SPTRUNC
- INTSP
- MPYSP

The .S and .L units share their long write port with the load port for the 32 most significant bits of an **LDDW** load. Therefore, the **LDDW** instruction and the .S or .L unit writing a long result cannot write to the same register file on the same cycle. The **LDDW** writes to the register file on pipeline phase E5. Instructions that use a long result and use the .L and .S unit write to the register file on pipeline phase E1. Therefore, the instruction with the long result must be scheduled later than four cycles following the **LDDW** instruction if both instructions use the same side.

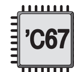
4.6 Individual Instruction Descriptions

This section gives detailed information on the floating-point instruction set for the 'C67x. Each instruction presents the following information:

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

Syntax

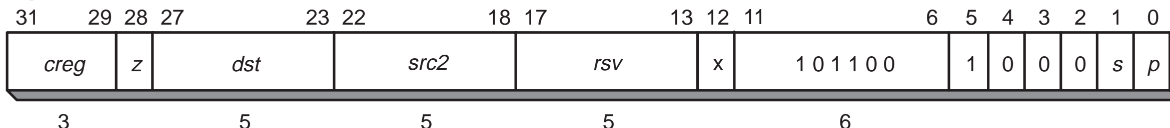
ABSDP (.unit) *src2*, *dst*



.unit = .S1 or .S2

Opcode map field used...	For operand type...	Unit
src2 *dst*	dp dp	.S1, .S2

Opcode



Description

The absolute value of *src2* is placed in *dst*. The 64-bit double-precision operand is read in one cycle by using the *src2* port for the 32 MSBs and the *src1* port for the 32 LSBs.

Execution

if (cond) abs(*src2*) $\rightarrow$ *dst*
else nop

The absolute value of *src2* is determined as follows:

1) If *src2* $\geq \times 0$, then *src2* $\rightarrow$ *dst*
2) If *src2* $<$ 0, then –*src2* $\rightarrow$ *dst*

Notes:

1) If *scr2* is SNaN, NaN_out is placed in *dst* and the INVAL and NAN2 bits are set.

2) If *src2* is QNaN, NaN_out is placed in *dst* and the NAN2 bit is set.

3) If *src2* is denormalized, +0 is placed in *dst* and the INEX and DEN2 bits are set.

4) If *src2* is +infinity or –infinity, +infinity is placed in *dst* and the INFO bit is set.

Pipeline

Pipeline Stage	E1	E2
Read	*src2_l* *src2_h*	
Written	*dst_l*	*dst_h*
Unit in use	.S	

If *dst* is used as the source for the **ADDDP**, **CMPEQDP**, **CMPLTDP**, **CMPGTDP**, **MPYDP**, or **SUBDP** instruction, the number of delay slots can be reduced by one, because these instructions read the lower word of the DP source one cycle before the upper word of the DP source.

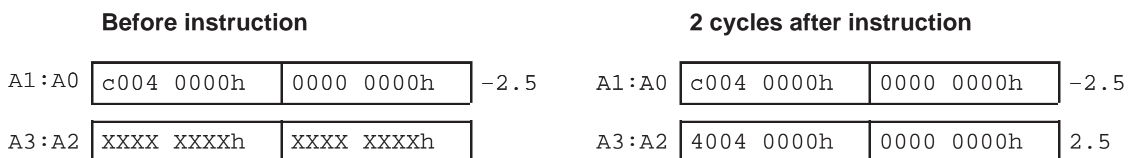
Instruction Type 2-cycle DP

Delay Slots 1

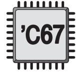
Functional Unit Latency 1

Example

```
ABSDP .S1  A1:A0,A3:A2
```



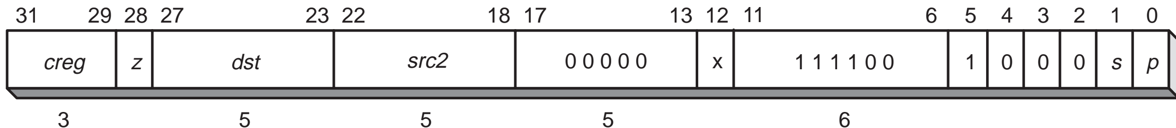
Syntax **ABSSP** (.unit) *src2*, *dst*



.unit = . S1 or .S2

Opcode map field used...	For operand type...	Unit
src2 *dst*	xsp sp	.S1, .S2

Opcode



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

Execution

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

The absolute value of *src2* is determined as follows:

1) If *src2* $\geq$ 0, then *src2* → *dst*
2) If *src2* $< \times$0, then –*src2* → *dst*

Notes:

1) If *scr2* is SNaN, NaN_out is placed in *dst* and the INVAL and NAN2 bits are set.
2) If *src2* is QNaN, NaN_out is placed in *dst* and the NAN2 bit is set.
3) If *src2* is denormalized, +0 is placed in *dst* and the INEX and DEN2 bits are set.
4) If *src2* is +infinity or –infinity, +infinity is placed in *dst* and the INFO bit is set.

Pipeline

Pipeline Stage	E1
Read	*src2*
Written	*dst*
Unit in use	.S

Instruction Type Single-cycle

Delay Slots 0

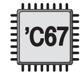
Functional Unit Latency 1

Example

```
ABSSP .S1X B1,A5
```

	Before instruction			1 cycle after instruction	
B1	c020 0000h	−2.5	B1	c020 0000h	−2.5
A5	XXXX XXXXh		A5	4020 0000h	2.5

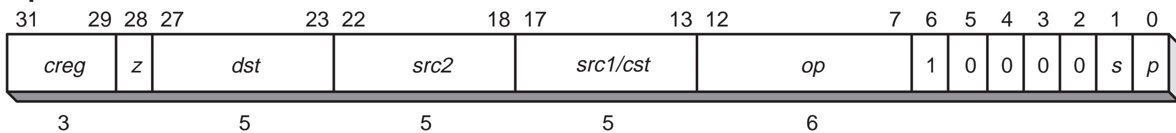
Syntax **ADDAD** (.unit) *src2*, *src1*, *dst*



.unit = . D1 or .D2

Opcode map field used...	For operand type...	Unit	Opfield
src2 *src1* *dst*	sint sint sint	.D1, .D2	111100
src2 *src1* *dst*	sint ucst5 sint	.D1, .D2	111101

Opcode



Description *src1* is added to *src2* using the doubleword 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 2.6.1 on page 2-9). *src1* is left shifted by 3 due to doubleword data sizes. The result is placed in *dst*. (See the **ADDAB/ADDAH/ADDAW** instruction, page 3-34, for byte, halfword, and word versions.)

Note:

There is no SUBAD instruction.

Execution

if (cond) *src2* +(src1 $<<$ 3) $\rightarrow$ *dst*
else nop

Pipeline

Pipeline Stage	E1
Read	*src1* *src2*
Written	*dst*
Unit in use	.D

Instruction Type Single-cycle

Delay Slots 0

Functional Unit Latency 1

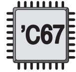
Example

```
ADDAD .D1  A1,A2,A3
```

	Before instruction			1 cycle after instruction	
A1	0000 1234h	4660	A1	0000 1234h	4660
A2	0000 0002h	2	A2	0000 0002h	2
A3	XXXX XXXXh		A3	0000 1244h	4676

Syntax

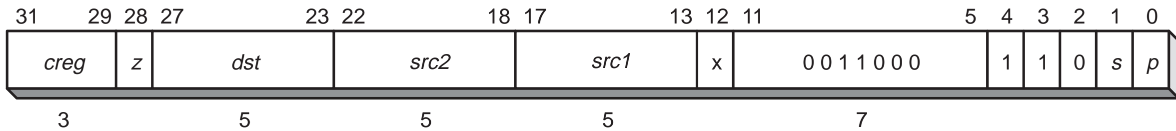
ADDDP (.unit) *src1*, *src2*, *dst*



.unit = .L1 or .L2

Opcode map field used...	For operand type...	Unit
src1	dp	.L1, .L2
src2	xdp	
dst	dp	

Opcode



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

Execution

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

Notes:

1) If rounding is performed, the INEX bit is set.
2) If one source is SNaN or QNaN, the result is NaN_out. If either source is SNaN, the INVAL bit is set, also.
3) If one source is +infinity and the other is –infinity, the result is NaN_out and the INVAL bit is set.
4) If one source is signed infinity and the other source is anything except NaN or signed infinity of the opposite sign, the result is signed infinity and the INFO bit is set.
5) If overflow occurs, the INEX and OVER bits are set and the results are rounded as follows (LFPN is the largest floating-point number):

	Overflow Output Rounding Mode			
Result Sign	**Nearest Even**	**Zero**	**+Infinity**	**–Infinity**
+	+infinity	+LFPN	+infinity	+LFPN
–	–infinity	–LFPN	–LFPN	–infinity

6) If underflow occurs, the INEX and UNDER bits are set and the results are rounded as follows (SPFN is the smallest floating-point number):

	Underflow Output Rounding Mode			
Result Sign	**Nearest Even**	**Zero**	**+Infinity**	**–Infinity**
+	+0	+0	+SFPN	+0
–	–0	–0	–0	–SFPN

7) If the sources are equal numbers of opposite sign, the result is +0 unless the rounding mode is –infinity, in which case the result is –0.
8) If the sources are both 0 with the same sign or both are denormalized with the same sign, the sign of the result is negative for negative sources and positive for positive sources.
9) A signed denormalized source is treated as a signed 0 and the DENn bit is set. If the other source is not NaN or signed infinity, the INEX bit is set.

Pipeline

Pipeline Stage	E1	E2	E3	E4	E5	E6	E7
Read	*src1_l* *src2_l*	*src1_h* *src2_h*					
Written						*dst_l*	*dst_h*
Unit in use	.L	.L					

If *dst* is used as the source for the **ADDDP**, **CMPEQDP**, **CMPLTDP**, **CMPGTDP**, **MPYDP**, or **SUBDP** instruction, the number of delay slots can be reduced by one, because these instructions read the lower word of the DP source one cycle before the upper word of the DP source.

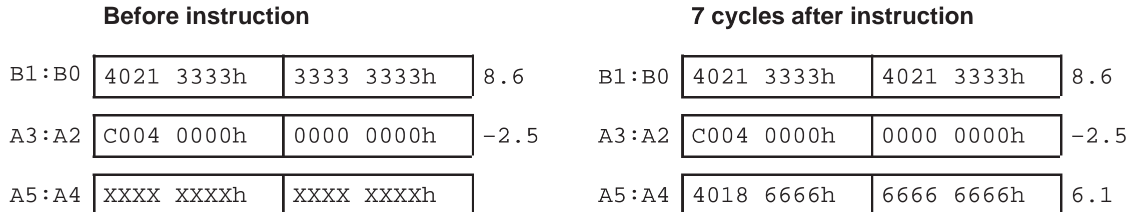
Instruction Type ADDDP/SUBDP

Delay Slots 6

Functional Unit Latency 2

Example

```
ADDDP .L1X B1:B0,A3:A2,A5:A4
```



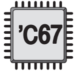
Before instruction

B1:B0	4021 3333h	3333 3333h	8.6
A3:A2	C004 0000h	0000 0000h	−2.5
A5:A4	XXXX XXXXh	XXXX XXXXh	

7 cycles after instruction

B1:B0	4021 3333h	4021 3333h	8.6
A3:A2	C004 0000h	0000 0000h	−2.5
A5:A4	4018 6666h	6666 6666h	6.1

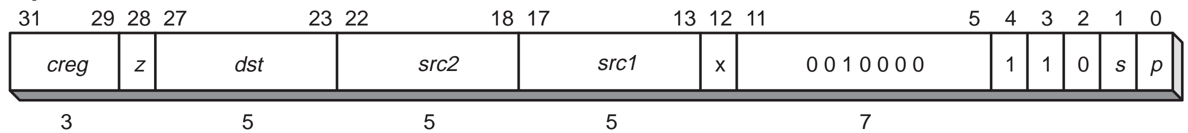
Syntax **ADDSP** (.unit) *src1*, *src2*, *dst*



.unit = .L1 or .L2

Opcode map field used...	For operand type...	Unit
src1	sp	.L1, .L2
src2	xsp	
dst	sp	

Opcode



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

Execution

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

Notes:

1) If rounding is performed, the INEX bit is set.
2) If one source is SNaN or QNaN, the result is NaN_out. If either source is SNaN, the INVAL bit is set also.
3) If one source is +infinity and the other is –infinity, the result is NaN_out and the INVAL bit is set.
4) If one source is signed infinity and the other source is anything except NaN or signed infinity of the opposite sign, the result is signed infinity and the INFO bit is set.
5) If overflow occurs, the INEX and OVER bits are set and the results are rounded as follows (LFPN is the largest floating-point number):

	Overflow Output Rounding Mode			
Result Sign	**Nearest Even**	**Zero**	**+Infinity**	**–Infinity**
+	+infinity	+LFPN	+infinity	+LFPN
–	–infinity	–LFPN	–LFPN	–infinity

6) If underflow occurs, the INEX and UNDER bits are set and the results are rounded as follows (SPFN is the smallest floating-point number):

	Underflow Output Rounding Mode			
Result Sign	**Nearest Even**	**Zero**	**+Infinity**	**–Infinity**
+	+0	+0	+SFPN	+0
–	–0	–0	–0	–SFPN

7) If the sources are equal numbers of opposite sign, the result is +0 unless the rounding mode is –infinity, in which case the result is –0.
8) If the sources are both 0 with the same sign or both are denormalized with the same sign, the sign of the result is negative for negative sources and positive for positive sources.
9) A signed denormalized source is treated as a signed 0 and the DENn bit is set. If the other source is not NaN or signed infinity, the INEX bit is also set.

Pipeline

Pipeline Stage	E1	E2	E3	E4
Read	*src1* *src2*			
Written				*dst*
Unit in use	.L			

Instruction Type 4-cycle

Delay Slots 3

Functional Unit Latency 1

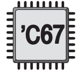
Example

```
ADDSP .L1  A1,A2,A3
```

	Before instruction			4 cycles after instruction	
A1	C020 0000h	−2.5	A1	C020 0000h	−2.5
A2	4109 999Ah	8.6	A2	4109 999Ah	8.6
A3	XXXX XXXXh		A3	40C3 3334h	6.1

Syntax

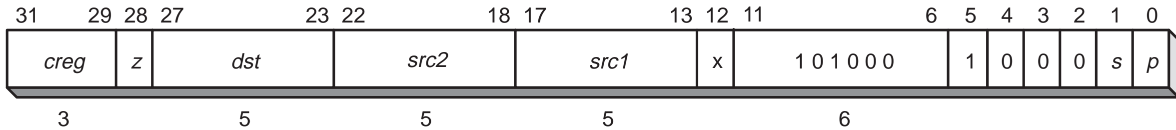
CMPEQDP (.unit) *src1*, *src2*, *dst*



.unit = .S1 or .S2

Opcode map field used...	For operand type...	Unit
src1 *src2* *dst*	dp xdp sint	.S1, .S2

Opcode



Description

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

Execution

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

Special cases of inputs:

Input			FAUCR Fields	
src1	***src2***	**Output**	**UNORD**	**INVAL**
NaN	don't care	0	1	0
don't care	NaN	0	1	0
NaN	NaN	0	1	0
+/–denormalized	+/–0	1	0	0
+/–0	+/–denormalized	1	0	0
+/–0	+/–0	1	0	0
+/–denormalized	+/–denormalized	1	0	0
+infinity	+infinity	1	0	0
+infinity	other	0	0	0
–infinity	–infinity	1	0	0
–infinity	other	0	0	0

Notes:

1) In the case of NaN compared with itself, the result is false.
2) No configuration bits besides those in the preceding table are set, except the NaNn and DENn bits when appropriate.

Pipeline

Pipeline Stage	E1	E2
Read	*src1_l* *src2_l*	*src1_h* *src2_h*
Written		*dst*
Unit in use	.S	.S

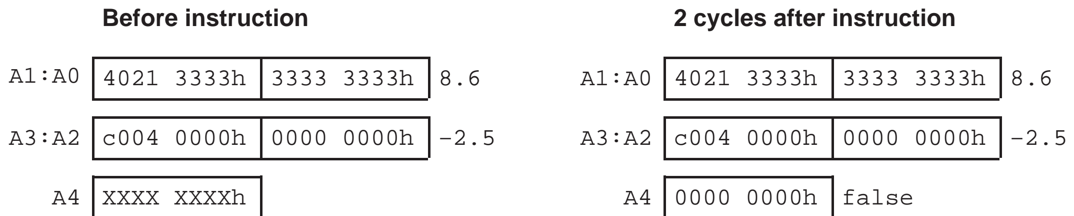
Instruction Type DP compare

Delay Slots 1

Functional Unit Latency 2

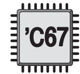
Example

```
CMPEQDP .S1 A1:A0,A3:A2,A4
```



Syntax

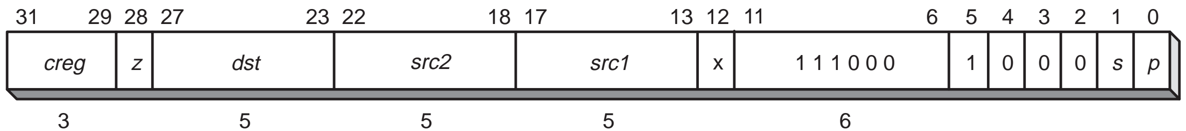
CMPEQSP (.unit) *src1*, *src2*, *dst*



.unit = .S1 or .S2

Opcode map field used...	For operand type...	Unit
src1 *src2* *dst*	sp xsp sint	.S1, .S2

Opcode



Description

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

Execution

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

Special cases of inputs:

Input		Output	Configuration Register	
src1	***src2***		**UNORD**	**INVAL**
NaN	don't care	0	1	0
don't care	NaN	0	1	0
NaN	NaN	0	1	0
+/–denormalized	+/–0	1	0	0
+/–0	+/–denormalized	1	0	0
+/–0	+/–0	1	0	0
+/–denormalized	+/–denormalized	1	0	0
+infinity	+infinity	1	0	0
+infinity	other	0	0	0
–infinity	–infinity	1	0	0
–infinity	other	0	0	0

Notes:

1) In the case of NaN compared with itself, the result is false.
2) No configuration bits besides those shown in the preceding table are set, except for the NaNn and DENn bits when appropriate.

Pipeline

Pipeline Stage	E1
Read	*src1* *src2*
Written	*dst*
Unit in use	.S

Instruction Type Single-cycle

Delay Slots 0

Functional Unit Latency 1

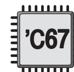
Example

```
CMPEQSP .S1 A1,A2,A3
```

	Before instruction			1 cycle after instruction	
A1	C020 0000h	−2.5	A1	C020 0000h	−2.5
A2	4109 999Ah	8.6	A2	4109 999Ah	8.6
A3	XXXX XXXXh		A3	0000 0000h	false

Syntax

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



.unit = .S1 or .S2

Opcode map field used...	For operand type...	Unit
src1 *src2* *dst*	dp xdp sint	.S1, .S2

Opcode

Description

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

Execution

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

Special cases of inputs:

Input		Output	Configuration Register	
src1	***src2***		**UNORD**	**INVAL**
NaN	don't care	0	1	1
don't care	NaN	0	1	1
NaN	NaN	0	1	1
+/–denormalized	+/–0	0	0	0
+/–0	+/–denormalized	0	0	0
+/–0	+/–0	0	0	0
+/–denormalized	+/–denormalized	0	0	0
+infinity	+infinity	0	0	0
+infinity	other	1	0	0
–infinity	–infinity	0	0	0
–infinity	other	0	0	0

> **Note:**
>
> No configuration bits besides those shown in the preceding table are set, except the NaNn and DENn bits when appropriate.

Pipeline

Pipeline Stage	E1	E2
Read	*src1_l* *src2_l*	*src1_h* *src2_h*
Written		*dst*
Unit in use	.S	.S

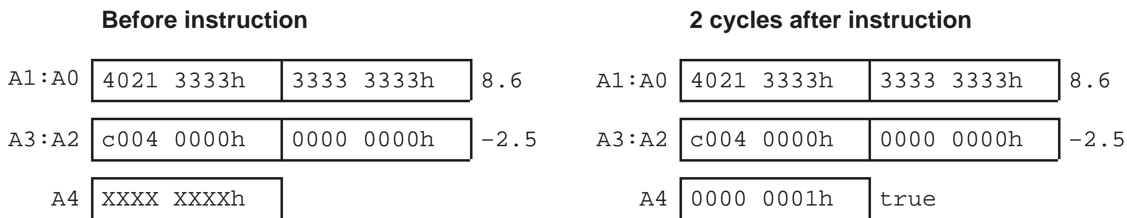
Instruction Type DP compare

Delay Slots 1

Functional Unit Latency 2

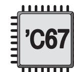
Example

```
CMPGTDP .S1 A1:A0,A3:A2,A4
```



Syntax

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



.unit = .S1 or .S2

Opcode map field used...	For operand type...	Unit
src1 *src2* *dst*	sp xsp sint	.S1, .S2

Opcode

Description

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

Execution

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

Special cases of inputs:

Input		Output	Configuration Register	
src1	***src2***		**UNORD**	**INVAL**
NaN	don't care	0	1	1
don't care	NaN	0	1	1
NaN	NaN	0	1	1
+/–denormalized	+/–0	0	0	0
+/–0	+/–denormalized	0	0	0
+/–0	+/–0	0	0	0
+/–denormalized	+/–denormalized	0	0	0
+infinity	+infinity	0	0	0
+infinity	other	1	0	0
–infinity	–infinity	0	0	0
–infinity	other	0	0	0

> **Note:**
>
> No configuration bits besides those shown in the preceding table are set, except for the NaNn and DENn bits when appropriate.

Pipeline

Pipeline Stage	E1
Read	*src1* *src2*
Written	*dst*
Unit in use	.S

Instruction Type Single-cycle

Delay Slots 0

Functional Unit Latency 1

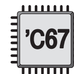
Example

```
CMPGTSP .S1X A1,B2,A3
```

	Before instruction			1 cycle after instruction	
A1	C020 0000h	−2.5	A1	C020 0000h	−2.5
B2	4109 999Ah	8.6	B2	4109 999Ah	8.6
A3	XXXX XXXXh		A3	0000 0000h	false

Syntax

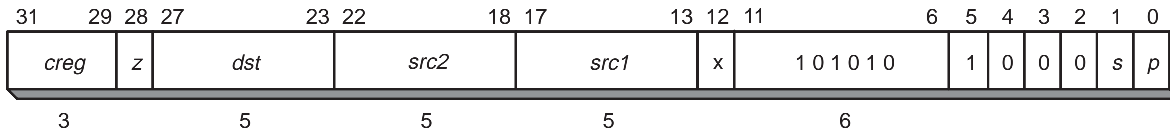
CMPLTDP (.unit) *src1*, *src2*, *dst*



.unit = .S1 or .S2

Opcode map field used...	For operand type...	Unit
src1 *src2* *dst*	dp xdp sint	.S1, .S2

Opcode



Description

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

Execution

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

Special cases of inputs:

Input		Output	Configuration Register	
src1	***src2***		**UNORD**	**INVAL**
NaN	don't care	0	1	1
don't care	NaN	0	1	1
NaN	NaN	0	1	1
+/–denormalized	+/–0	0	0	0
+/–0	+/–denormalized	0	0	0
+/–0	+/–0	0	0	0
+/–denormalized	+/–denormalized	0	0	0
+infinity	+infinity	0	0	0
+infinity	other	0	0	0
–infinity	–infinity	0	0	0
–infinity	other	1	0	0

> **Note:**
>
> No configuration bits besides those shown in the preceding table are set, except for the NaNn and DENn bits when appropriate.

Pipeline

Pipeline Stage	E1	E2
Read	*src1_l* *src2_l*	*src1_h* *src2_h*
Written		*dst*
Unit in use	.S	.S

Instruction Type DP compare

Delay Slots 1

Functional Unit Latency 2

Example

```
CMPLTDP    .S1X       A1:A0,B3:B2,A4
```

Before instruction

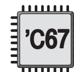
Register	High	Low	Value
A1:A0	4021 3333h	3333 3333h	8.6
B3:B2	c004 0000h	0000 0000h	−2.5
A4	XXXX XXXXh		

2 cycles after instruction

Register	High	Low	Value
A1:A0	4021 3333h	4021 3333h	8.6
B3:B2	c004 0000h	0000 0000h	−2.5
A4	0000 0000h		false

Syntax

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



.unit = .S1 or .S2

Opcode map field used...	For operand type...	Unit
src1	sp	.S1, .S2
src2	xsp	
dst	sint	

Opcode

Description

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

Execution

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

Special cases of inputs:

Input			Configuration Register	
src1	***src2***	**Output**	**UNORD**	**INVAL**
NaN	don't care	0	1	1
don't care	NaN	0	1	1
NaN	NaN	0	1	1
+/–denormalized	+/–0	0	0	0
+/–0	+/–denormalized	0	0	0
+/–0	+/–0	0	0	0
+/–denormalized	+/–denormalized	0	0	0
+infinity	+infinity	0	0	0
+infinity	other	0	0	0
–infinity	–infinity	0	0	0
–infinity	other	1	0	0

Note:

No configuration bits besides those shown in the preceding table are set, except for the NaNn and DENn bits when appropriate.

Pipeline

Pipeline Stage	E1
Read	*src1* *src2*
Written	*dst*
Unit in use	.S

Instruction Type Single-cycle

Delay Slots 0

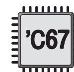
Functional Unit Latency 1

Example

```
CMPGTSP .S1 A1,A2,A3
```

	Before instruction			1 cycle after instruction	
A1	C020 0000h	−2.5	A1	C020 0000h	−2.5
A2	4109 999Ah	8.6	A2	4109 999Ah	8.6
A3	XXXX XXXXh		A3	0000 0001h	true

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



.unit = .L1 or .L2

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

Opcode

Description The 64-bit double-precision value in *src2* is converted to an integer and placed in *dst*. The operand is read in one cycle by using the *src2* port for the 32 MSBs and the *src1* port for the 32 LSBs.

Execution

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

Notes:

1) If *src2* is NaN, the maximum signed integer (7FFF FFFFh or 8000 0000h) is placed in *dst* and the INVAL bit is set.

2) If *src2* is signed infinity or if overflow occurs, the maximum signed integer (7FFF FFFFh or 8000 0000h) is placed in *dst* and the INEX and OVER bits are set. Overflow occurs if *src2* is greater than $2^{31} - 1$ or less than -2^{31}.

3) If *src2* is denormalized, 0000 0000h is placed in *dst* and the INEX and DEN2 bits are set.

4) If rounding is performed, the INEX bit is set.

Pipeline

Pipeline Stage	E1	E2	E3	E4
Read	*src2_l* *src2_h*			
Written				*dst*
Unit in use	.L			

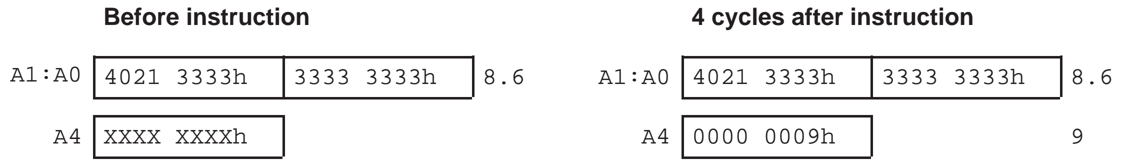
Instruction Type 4-cycle

Delay Slots 3

Functional Unit Latency 1

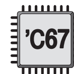
Example

```
DPINT   .L1   A1:A0,A4
```



Syntax

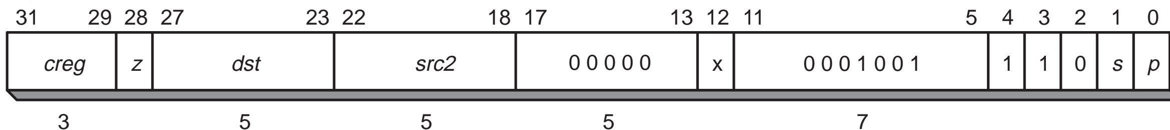
DPSP (.unit) *src2*, *dst*



.unit = .L1 or .L2

Opcode map field used...	For operand type...	Unit
src2 *dst*	dp sp	.L1, .L2

Opcode



Description

The double-precision 64-bit value in *src2* is converted to a single-precision value and placed in *dst*. The operand is read in one cycle by using the *src2* port for the 32 MSBs and the *src1* port for the 32 LSBs.

Execution

if (cond)	sp(*src2*) → *dst*
else	nop

Notes:

1) If rounding is performed, the INEX bit is set.
2) If *src2* is SNaN, NaN_out is placed in *dst* and the INVAL and NAN2 bits are set.
3) If *src2* is QNaN, NaN_out is placed in *dst* and the NAN2 bit is set.
4) If *src2* is a signed denormalized number, signed 0 is placed in *dst* and the INEX and DEN2 bits are set.
5) If *src2* is signed infinity, the result is signed infinity and the INFO bit is set.
6) If overflow occurs, the INEX and OVER bits are set and the results are set as follows (LFPN is the largest floating-point number):

	Overflow Output Rounding Mode			
Result Sign	**Nearest Even**	**Zero**	**+Infinity**	**–Infinity**
+	+infinity	+LFPN	+infinity	+LFPN
–	–infinity	–LFPN	–LFPN	–infinity

7) If underflow occurs, the INEX and UNDER bits are set and the results are set as follows (SPFN is the smallest floating-point number):

	Underflow Output Rounding Mode			
Result Sign	**Nearest Even**	**Zero**	**+Infinity**	**–Infinity**
+	+0	+0	+SFPN	+0
–	–0	–0	–0	–SFPN

Pipeline

Pipeline Stage	E1	E2	E3	E4
Read	*src2_l* *src2_h*			
Written				*dst*
Unit in use	.L			

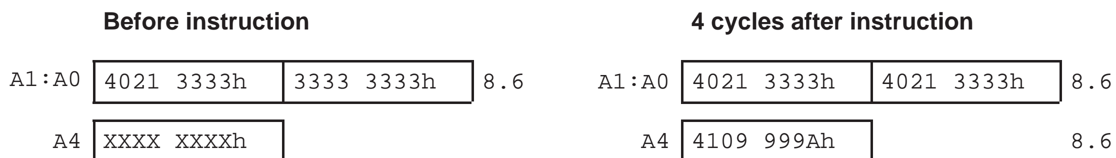
Instruction Type 4-cycle

Delay Slots 3

Functional Unit Latency 1

Example

```
DPSP      .L1       A1:A0,A4
```



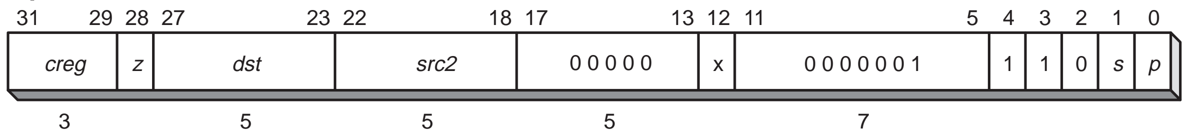
Syntax

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

.unit = .L1 or .L2

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

Opcode



Description

The 64-bit double-precision value in *src2* is converted to an integer and placed in *dst*. This instruction operates like **DPINT** except that the rounding modes in the FADCR are ignored; round toward zero (truncate) is always used. The 64-bit operand is read in one cycle by using the *src2* port for the 32 MSBs and the src1 port for the 32 LSBs.

Execution

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

Notes:

1) If *src2* is NaN, the maximum signed integer (7FFF FFFFh or 8000 0000h) is placed in *dst* and the INVAL bit is set.

2) If *src2* is signed infinity or if overflow occurs, the maximum signed integer (7FFF FFFFh or 8000 0000h) is placed in *dst* and the INEX and OVER bits are set. Overflow occurs if *src2* is greater than $2^{31} - 1$ or less than -2^{31}.

3) If *src2* is denormalized, 0000 0000h is placed in *dst* and the INEX and DEN2 bits are set.

4) If rounding is performed, the INEX bit is set.

Pipeline

Pipeline Stage	E1	E2	E3	E4
Read	*src2_l* *src2_h*			
Written				*dst*
Unit in use	.L			

Instruction Type

4-cycle

Delay Slots 3

Functional Unit Latency 1

Example

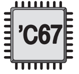
```
DPTRUNC    .L1        A1:A0,A4
```

Before instruction

A1:A0	4021 3333h	3333 3333h	8.6
A4	XXXX XXXXh		

4 cycles after instruction

A1:A0	4021 3333h	3333 3333h	8.6
A4	0000 0008h		8



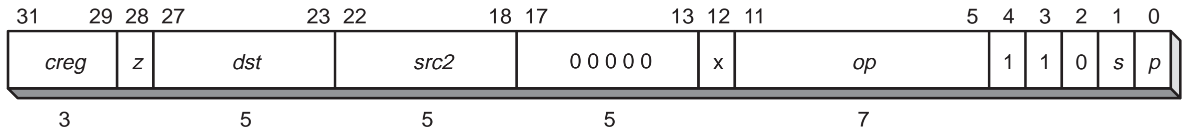
Syntax

INTDP (.unit) *src2*, *dst*
or
INTDPU (.unit) *src2*, *dst*

.unit = .L1 or .L2

Opcode map field used...	For operand type...	Unit	Opfield
src2 *dst*	xsint dp	.L1, .L2	0111001
src2 *dst*	xuint dp	.L1, .L2	0111011

Opcode



Description

The integer value in *src2* is converted to a double-precision value and placed in *dst*.

Execution

if (cond) dp(*src2*) → *dst*
else nop

You cannot set configuration bits with this instruction.

Pipeline

Pipeline Stage	E1	E2	E3	E4	E5
Read	*src2*				
Written				*dst_l*	*dst_h*
Unit in use	.L				

If *dst* is used as the source for the **ADDDP**, **CMPEQDP**, **CMPLTDP**, **CMPGTDP**, **MPYDP**, or **SUBDP** instruction, the number of delay slots can be reduced by one, because these instructions read the lower word of the DP source one cycle before the upper word of the DP source.

Instruction Type

INTDP

Delay Slots

4

Functional Unit Latency

1

Example 1

```
INTDP    .L1x  B4,A1:A0
```

	Before instruction				5 cycles after instruction		
B4	1965 1127h	426053927		B4	1965 1127h	426053927	
A1:A0	XXXX XXXXh	XXXX XXXXh		A1:A0	41B9 6511h	2700 0000h	4.2605393 E08

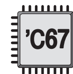
Example 2

```
INTDPU   .L1  A4,A1:A0
```

	Before instruction				5 cycles after instruction		
A4	FFFF FFDEh	4294967262		A4	FFFF FFDEh	4294967262	
A1:A0	XXXX XXXXh	XXXX XXXXh		A1:A0	41EF FFFFh	FBC0 0000h	4.2949673 E09

Syntax

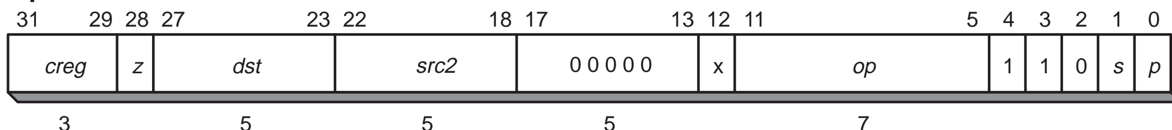
INTSP (.unit) *src2*, *dst*
or
INTSPU (.unit) *src2*, *dst*



.unit = .L1 or .L2

Opcode map field used...	For operand type...	Unit	Opfield
src2 *dst*	xsint sp	.L1, .L2	1001010
src2 *dst*	xuint sp	.L1, .L2	1001001

Opcode



Description The integer value in *src2* is converted to single-precision value and placed in *dst*.

Execution

if (cond) sp(*src2*) → *dst*
else nop

The only configuration bit that can be set is the INEX bit and only if the mantissa is rounded.

Pipeline

Pipeline Stage	E1	E2	E3	E4
Read	*src2*			
Written				*dst*
Unit in use	.L			

Instruction Type 4-cycle

Delay Slots 3

Functional Unit Latency 1

Example 1

```
INTSP .L1 A1,A2
```

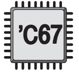
	Before instruction			4 cycles after instruction	
A1	1965 1127h	426053927	A1	1965 1127h	426053927
A2	XXXX XXXXh		A2	4DCB 2889h	4.2605393 E08

Example 2

```
INTSPU .L1X B1,A2
```

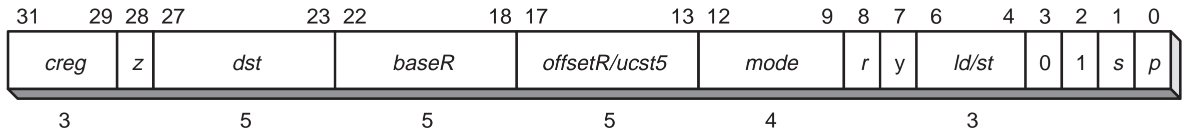
	Before instruction			4 cycles after instruction	
B1	FFFF FFDEh	4294967262	B1	C020 0000h	4294967262
A2	XXXX XXXXh		A2	4F80 0000h	4.2949673 E09

Syntax **LDDW** (.unit) *+*baseR[offsetR/ucst5]*, *dst*



.unit = .D1 or .D2

Opcode



Description This instruction loads a doubleword to a pair of general-purpose registers (*dst*). Table 4–10 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 the register file used: y = 0 selects the .D1 unit and the *baseR* and *offsetR* from the A register file, and y = 1 selects the .D2 unit and *baseR* and *offsetR* from the B register file. The *s* bit determines the register file into which the *dst* is loaded: s = 0 indicates that *dst* is in the A register file, and s = 1 indicates that *dst* is in the B register file. The *r* bit has a value of 1 for the **LDDW** instruction and a value of 0 for all other load and store instructions. The *dst* field must always be an even value because LDDW loads register pairs. Therefore, bit 23 is always zero. Furthermore, the value of the *ld/st* field is110.

The bracketed *offsetR*/*ucst5* is scaled by a left-shift of 3 to correctly represent doublewords. 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 shifted value of *baseR* before the addition or subtraction is the address to be accessed in memory.

Increments and decrements default to 1 and offsets default to 0 when no bracketed register, bracketed constant, or constant enclosed in parentheses is specified. Square brackets, [ ], indicate that *ucst5* is left shifted by 3. Parentheses, ( ), indicate that *ucst5* is not left shifted. In other words, parentheses indicate a byte offset rather than a doubleword offset. You must type either brackets or parathesis around the specified offset if you use the optional offset parameter.

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 2.6.1 on page 2-9).

The destination register pair must consist of a consecutive even and odd register pair from the same register file. The instruction can be used to load a double-precision floating-point value (64 bits), a pair of single-precision floating-point words (32 bits), or a pair of 32-bit integers. The least significant 32 bits are loaded into the even register and the most significant 32 bits (containing the sign bit and exponent) are loaded into the next register (which is always the odd register). The register pair syntax places the odd register first, followed by a colon, then the even register (that is, A1:A0, B1:B0, A3:A2, B3:B2, etc.).

All 64 bits of the double-precision floating point value are stored in big- or little-endian byte order, depending on the mode selected. When **LDDW** is used to load two 32-bit single-precision floating-point values or two 32-bit integer values, the order is dependent on the endian mode used. In little-endian mode, the first 32-bit word in memory is loaded into the even register. In big-endian mode, the first 32-bit word in memory is loaded into the odd register. Regardless of the endian mode, the double word address must be on a doubleword boundary (the three LSBs are zero).

Table 4–10 summarizes the address generation options supported.

Table 4–10. 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

Execution

if (cond) mem → *dst*
else nop

Pipeline

Pipeline Stage	E1	E2	E3	E4	E5
Read	baseR offsetR				
Written	baseR				*dst*
Unit in use	.D				

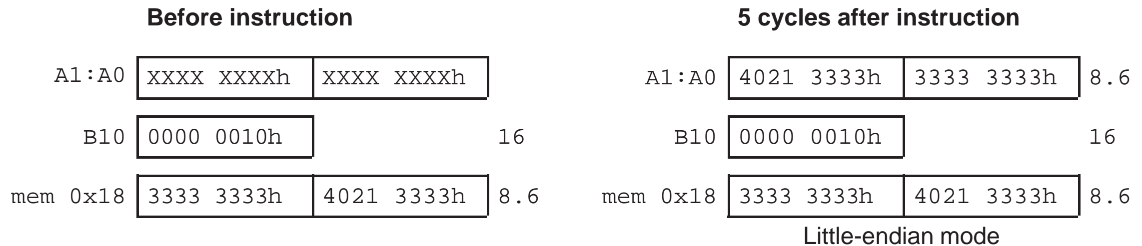
Instruction Type Load

Delay Slots 4

Functional Unit Latency 1

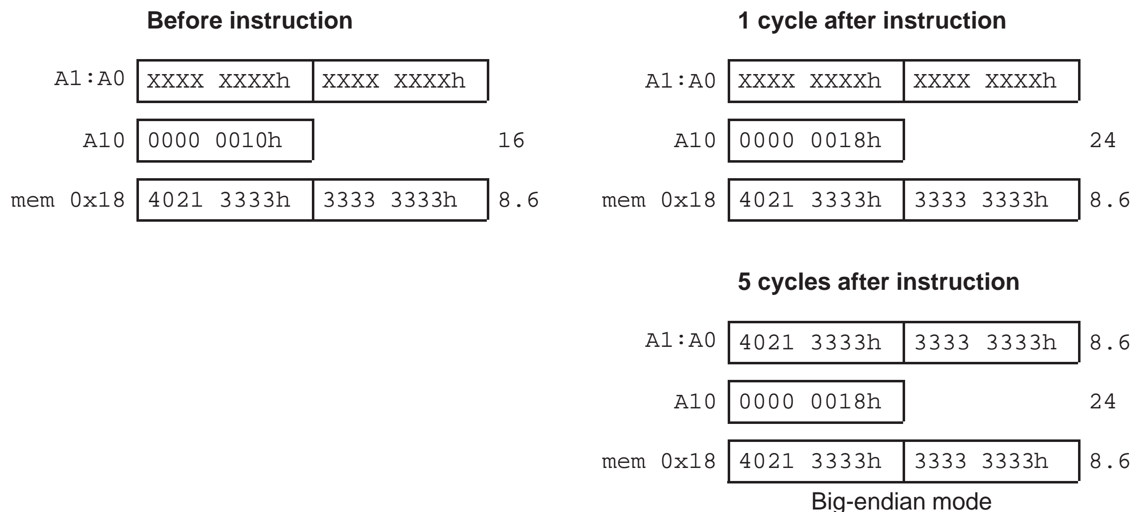
Example 1

```
LDDW .D2   *+B10[1],A1:A0
```

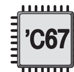


Example 2

```
LDDW .D1   *++A10[1],A1:A0
```



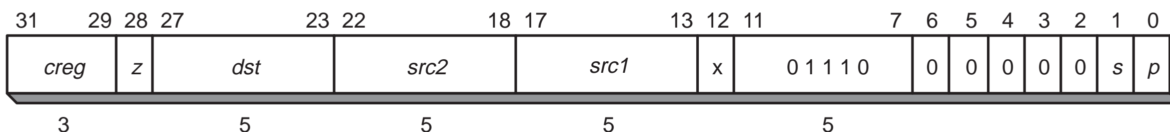
Syntax **MPYDP** (.unit) *src1*, *src2*, *dst*



.unit = .M1 or .M2

Opcode map field used...	For operand type...	Unit
src1	dp	.M1, .M2
src2	dp	
dst	dp	

Opcode



Description The *src1* operand is multiplied by the *src2* operand. The result is placed in *dst*.

Execution

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

Notes:

1) If one source is SNaN or QNaN, the result is a signed NaN_out. If either source is SNaN, the INVAL bit is set also. The sign of NaN_out is the exclusive-or of the input signs.

2) Signed infinity multiplied by signed infinity or a normalized number (other than signed 0) returns signed infinity. Signed infinity multiplied by signed 0 returns a signed NaN_out and sets the INVAL bit.

3) If one or both sources are signed 0, the result is signed 0 unless the other source is NaN or signed infinity, in which case the result is signed NaN_out.

4) If signed 0 is multiplied by signed infinity, the result is signed NaN_out and the INVAL bit is set.

5) A denormalized source is treated as signed 0 and the DENn bit is set. The INEX bit is set except when the other source is signed infinity, signed NaN, or signed 0. Therefore, a signed infinity multiplied by a denormalized number gives a signed NaN_out and sets the INVAL bit.

6) If rounding is performed, the INEX bit is set.

Pipeline

Pipeline Stage	E1	E2	E3	E4	E5	E6	E7	E8	E9	E10
Read	*src1_l* *src2_l*	*src1_l* *src2_h*	*src1_h* *src2_l*	*src1_h* *src2_h*						
Written									*dst_l*	*dst_h*
Unit in use	.M	.M	.M	.M						

If *dst* is used as the source for the **ADDDP**, **CMPEQDP**, **CMPLTDP**, **CMPGTDP**, **MPYDP**, or **SUBDP** instruction, the number of delay slots can be reduced by one, because these instructions read the lower word of the DP source one cycle before the upper word of the DP source.

Instruction Type MPYDP

Delay Slots 9

Functional Unit Latency 4

Example

```
MPYDP .M1  A1:A0,A3:A2,A5:A4
```

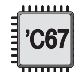
Before instruction

A1:A0	4021 3333h	3333 3333h	8.6
A3:A2	C004 0000h	0000 0000	−2.5
A5:A4	XXXX XXXXh	XXXX XXXXh	

10 cycles after instruction

A1:A0	4021 3333h	4021 3333h	8.6
A3:A2	C004 0000h	0000 0000h	−2.5
A5:A4	C035 8000h	0000 0000h	−21.5

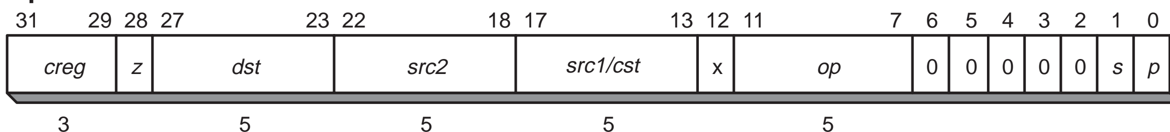
Syntax **MPYI** (.unit) *src1*, *src2*, *dst*



.unit = .M1 or .M2

Opcode map field used...	For operand type...	Unit	Opfield
src1 *src2* *dst*	sint xsint sint	.M1, .M2	00100
src1 *src2* *dst*	cst5 xsint sint	.M1, .M2	00110

Opcode



Description The *src1* operand is multiplied by the *src2* operand. The lower 32 bits of the result are placed in *dst*.

Execution

if (cond) lsb32(*src1* × *src2*) → *dst*
else nop

Pipeline

Pipeline Stage	E1	E2	E3	E4	E5	E6	E7	E8	E9
Read	*src1* *src2*	*src1* *src2*	*src1* *src2*	*src1* *src2*					
Written									*dst*
Unit in use	.M	.M	.M	.M					

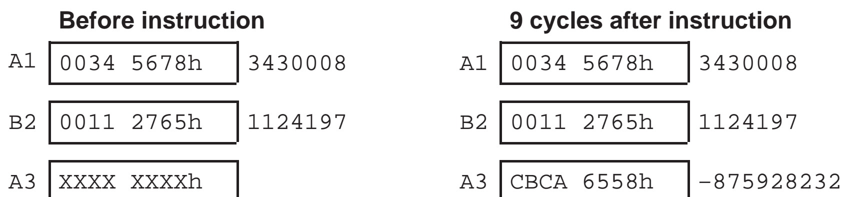
Instruction Type MPYI

Delay Slots 8

Functional Unit Latency 4

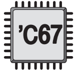
Example

```
MPYI       .M1X       A1,B2,A3
```



	Before instruction		9 cycles after instruction	
A1	0034 5678h	3430008	0034 5678h	3430008
B2	0011 2765h	1124197	0011 2765h	1124197
A3	XXXX XXXXh		CBCA 6558h	−875928232

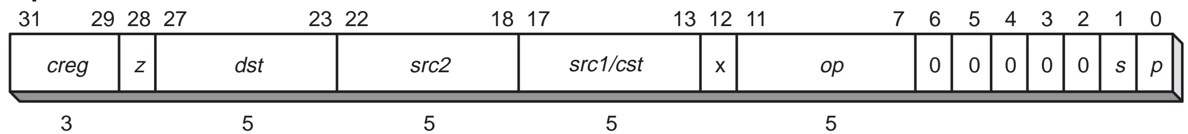
Syntax **MPYID** (.unit) *src1*, *src2*, *dst*



.unit = .M1 or .M2

Opcode map field used...	For operand type...	Unit	Opfield
src1 *src2* *dst*	sint xsint sdint	.M1, .M2	01000
src1 *src2* *dst*	cst5 xsint sdint	.M1, .M2	01100

Opcode



Description The *src1* operand is multiplied by the *src2* operand. The 64-bit result is placed in the *dst* register pair.

Execution

if (cond) lsb32(*src1* × *src2*) → *dst_l*
msb32(*src1* × *src2*) → *dst_h*
else nop

Pipeline

Pipeline Stage	E1	E2	E3	E4	E5	E6	E7	E8	E9	E10
Read	*src1* *src2*	*src1* *src2*	*src1* *src2*	*src1* *src2*						
Written									*dst_l*	*dst_h*
Unit in use	.M	.M	.M	.M						

Instruction Type MPYID

Delay Slots 9 (8 if *dst_l* is *src* of next instruction)

Functional Unit Latency 4

Example

```
MPYID .M1  A1,A2,A5:A4
```

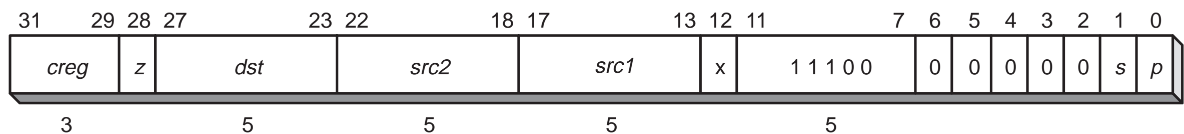
	Before instruction				10 cycles after instruction		
A1	0034 5678h		3430008	A1	0034 5678h		3430008
A2	0011 2765h		1124197	A2	0011 2765h		1124197
A5:A4	XXXX XXXXh	XXXX XXXXh		A5:A4	0000 0381h	CBCA 6558h	3856004703576

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

.unit = .M1 or .M2

Opcode map field used...	For operand type...	Unit
src1	sp	.M1, .M2
src2	xsp	
dst	sp	

Opcode



Description The *src1* operand is multiplied by the *src2* operand. The result is placed in *dst*.

Execution

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

Notes:

1) If one source is SNaN or QNaN, the result is a signed NaN_out. If either source is SNaN, the INVAL bit is set also. The sign of NaN_out is the exclusive-or of the input signs.

2) Signed infinity multiplied by signed infinity or a normalized number (other than signed 0) returns signed infinity. Signed infinity multiplied by signed 0 returns a signed NaN_out and sets the INVAL bit.

3) If one or both sources are signed 0, the result is signed 0 unless the other source is NaN or signed infinity, in which case the result is signed NaN_out.

4) If signed 0 is multiplied by signed infinity, the result is signed NaN_out and the INVAL bit is set.

5) A denormalized source is treated as signed 0 and the DENn bit is set. The INEX bit is set except when the other source is signed infinity, signed NaN, or signed 0. Therefore, a signed infinity multiplied by a denormalized number gives a signed NaN_out and sets the INVAL bit.

6) If rounding is performed, the INEX bit is set.

Pipeline

Pipeline Stage	E1	E2	E3	E4
Read	*src1* *src2*			
Written				*dst*
Unit in use	.M			

If *dst* is used as the source for the **ADDDP**, **CMPEQDP**, **CMPLTDP**, **CMPGTDP**, **MPYDP**, or **SUBDP** instruction, the number of delay slots can be reduced by one, because these instructions read the lower word of the DP source one cycle before the upper word of the DP source.

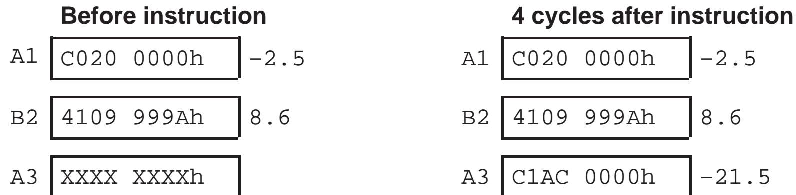
Instruction Type 4-cycle

Delay Slots 3

Functional Unit Latency 1

Example

```
MPYSP .M1X A1,B2,A3
```

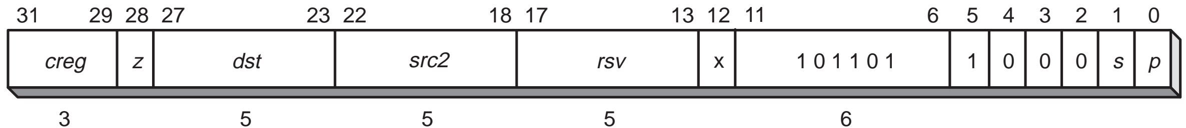


Syntax **RCPDP** (.unit) *src2*, *dst*

.unit = .S1 or .S2

Opcode map field used...	For operand type...	Unit
src2 *dst*	dp dp	.S1, .S2

Opcode



Description The 64-bit double-precision floating-point reciprocal approximation value of *src2* is placed in *dst*. The operand is read in one cycle by using the *src1* port for the 32 LSBs and the *src2* port for the 32 MSBs.

The **RCPDP** instruction provides the correct exponent, and the mantissa is accurate to the eighth binary position (therefore, mantissa error is less than 2^{-8}). This estimate can be used as a seed value for an algorithm to compute the reciprocal to greater accuracy. The Newton-Rhapson algorithm can further extend the mantissa’s precision:

$$x[n+1] = x[n](2 - v*x[n])$$

where v = the number whose reciprocal is to be found.

x[0], the seed value for the algorithm, is given by **RCPDP**. For each iteration, the accuracy doubles. Thus, with one iteration, accuracy is 16 bits in the mantissa; with the second iteration, the accuracy is 32 bits; with the third iteration, the accuracy is the full 52 bits.

Execution

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

Notes:

1) If *src2* is SNaN, NaN_out is placed in *dst* and the INVAL and NAN2 bits are set.
2) If *src2* is QNaN, NaN_out is placed in *dst* and the NAN2 bit is set.
3) If *src2* is a signed denormalized number, signed infinity is placed in *dst* and the DIV0, INFO, OVER, INEX, and DEN2 bits are set.
4) If *src2* is signed 0, signed infinity is placed in *dst* and the DIV0 and INFO bits are set.
5) If *src2* is signed infinity, signed 0 is placed in *dst*.
6) If the result underflows, signed 0 is placed in *dst* and the INEX and UNDER bits are set. Underflow occurs when $2^{1022} < src2 <$ infinity.

Pipeline

Pipeline Stage	E1	E2
Read	*src2_l* *src2_h*	
Written	*dst_l*	*dst_h*
Unit in use	.S	

If *dst* is used as the source for the **ADDDP**, **CMPEQDP**, **CMPLTDP**, **CMPGTDP**, **MPYDP**, or **SUBDP** instruction, the number of delay slots can be reduced by one, because these instructions read the lower word of the DP source one cycle before the upper word of the DP source.

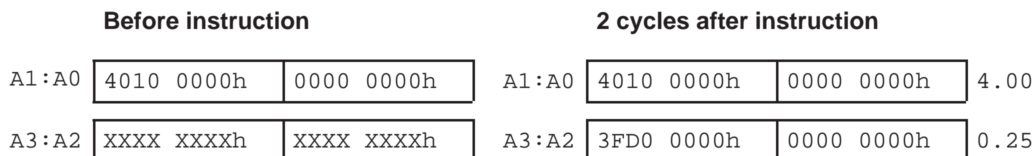
Instruction Type 2-cycle DP

Delay Slots 1

Functional Unit Latency 1

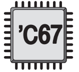
Example

```
RCPDP    .S1    A1:A0,A3:A2
```



Syntax

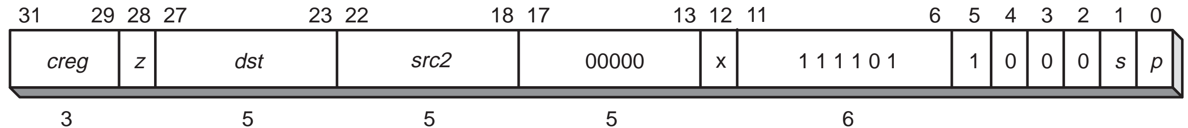
RCPSP (.unit) *src2*, *dst*



.unit = .S1 or .S2

Opcode map field used...	For operand type...	Unit
src2 *dst*	xsp sp	.S1, .S2

Opcode



Description

The single-precision floating-point reciprocal approximation value of *src2* is placed in *dst*.

The **RCPSP** instruction provides the correct exponent, and the mantissa is accurate to the eighth binary position (therefore, mantissa error is less than 2^{-8}). This estimate can be used as a seed value for an algorithm to compute the reciprocal to greater accuracy. The Newton-Rhapson algorithm can further extend the mantissa's precision:

x[n+1] = x[n](2 – v*x[n])

where v = the number whose reciprocal is to be found.

x[0], the seed value for the algorithm, is given by **RCPSP**. For each iteration, the accuracy doubles. Thus, with one iteration, accuracy is 16 bits in the mantissa; with the second iteration, the accuracy is the full 23 bits.

Execution

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

Notes:

1) If *src2* is SNaN, NaN_out is placed in *dst* and the INVAL and NAN2 bits are set.
2) If *src2* is QNaN, NaN_out is placed in *dst* and the NAN2 bit is set.
3) If *src2* is a signed denormalized number, signed infinity is placed in *dst* and the DIV0, INFO, OVER, INEX, and DEN2 bits are set.
4) If *src2* is signed 0, signed infinity is placed in *dst* and the DIV0 and INFO bits are set.
5) If *src2* is signed infinity, signed 0 is placed in *dst*.
6) If the result underflows, signed 0 is placed in *dst* and the INEX and UNDER bits are set. Underflow occurs when $2^{126} < src2 <$ infinity.

Pipeline

Pipeline Stage	E1
Read	*src2*
Written	*dst*
Unit in use	.S

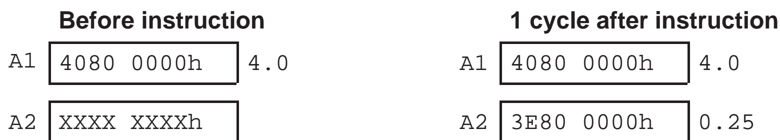
Instruction Type Single-cycle

Delay Slots 0

Functional Unit Latency 1

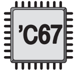
Example

```
MPYSP .S1  A1,A2
```



	Before instruction			1 cycle after instruction	
A1	4080 0000h	4.0	A1	4080 0000h	4.0
A2	XXXX XXXXh		A2	3E80 0000h	0.25

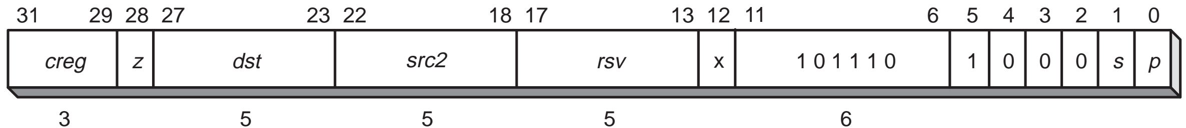
Syntax **RSQRDP** (.unit) *src2*, *dst*



.unit = .S1 or .S2

Opcode map field used...	For operand type...	Unit
src2 *dst*	dp dp	.S1, .S2

Opcode



Description The 64-bit double-precision floating-point square-root reciprocal approximation value of *src2* is placed in *dst*. The operand is read in one cycle by using the src1 port for the 32 LSBs and the *src2* port for the 32 MSBs.

The **RSQRDP** instruction provides the correct exponent, and the mantissa is accurate to the eighth binary position (therefore, mantissa error is less than 2^{-8}). This estimate can be used as a seed value for an algorithm to compute the reciprocal square root to greater accuracy.

The Newton-Rhapson algorithm can further extend the mantissa's precision:

x[n+1] = x[n](1.5 – (v/2)*x[n]*x[n])

where v = the number whose reciprocal square root is to be found.

x[0], the seed value for the algorithm is given by **RSQRDP**. For each iteration the accuracy doubles. Thus, with one iteration, the accuracy is 16 bits in the mantissa; with the second iteration, the accuracy is 32 bits; with the third iteration, the accuracy is the full 52 bits.

Execution

```
if (cond)    sqrcp(src2) → dst
else         nop
```

Notes:

1) If *src2* is SNaN, NaN_out is placed in *dst* and the INVAL and NAN2 bits are set.
2) If *src2* is QNaN, NaN_out is placed in *dst* and the NAN2 bit is set.
3) If *src2* is a negative, nonzero, nondenormalized number, NaN_out is placed in *dst* and the INVAL bit is set.
4) If *src2* is a signed denormalized number, signed infinity is placed in *dst* and the DIV0, INEX, and DEN2 bits are set.
5) If *src2* is signed 0, signed infinity is placed in *dst* and the DIV0 and INFO bits are set. The Newton-Rhapson approximation cannot be used to calculate the square root of 0 because infinity multiplied by 0 is invalid.
6) If *src2* is positive infinity, positive 0 is placed in *dst*.

Pipeline

Pipeline Stage	E1	E2
Read	*src2_l* *src2_h*	
Written	*dst_l*	*dst_h*
Unit in use	.S	

If *dst* is used as the source for the **ADDDP**, **CMPEQDP**, **CMPLTDP**, **CMPGTDP**, **MPYDP**, or **SUBDP** instruction, the number of delay slots can be reduced by one, because these instructions read the lower word of the DP source one cycle before the upper word of the DP source.

Instruction Type 2-cycle DP

Delay Slots 1

Functional Unit Latency 1

Example

```
RCPDP   .S1   A1:A0,A3:A2
```

Before instruction

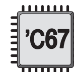
A1:A0	4010 0000h	0000 0000h	4.0
A3:A2	XXXX XXXXh	XXXX XXXXh	

2 cycles after instruction

A1:A0	4010 0000h	0000 0000h	4.0
A3:A2	3FE0 0000h	0000 0000h	0.5

Syntax

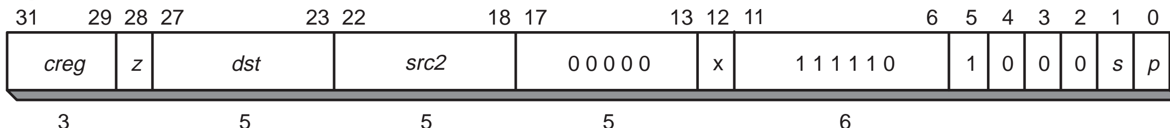
RSQRSP (.unit) *src2*, *dst*



.unit = .S1 or .S2

Opcode map field used...	For operand type...	Unit
src2	xsp	.S1, .S2
dst	sp	

Opcode



Description

The single-precision floating-point square-root reciprocal approximation value of *src2* is placed in *dst*.

The **RSQRSP** instruction provides the correct exponent, and the mantissa is accurate to the eighth binary position (therefore, mantissa error is less than 2^{-8}). This estimate can be used as a seed value for an algorithm to compute the reciprocal square root to greater accuracy.

The Newton-Rhapson algorithm can further extend the mantissa's precision:

x[n+1] = x[n](1.5 – (v/2)*x[n]*x[n])

where v = the number whose reciprocal square root is to be found.

x[0], the seed value for the algorithm, is given by **RSQRSP**. For each iteration, the accuracy doubles. Thus, with one iteration, accuracy is 16 bits in the mantissa; with the second iteration, the accuracy is the full 23 bits.

Execution

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

Notes:

1) If *src2* is SNaN, NaN_out is placed in *dst* and the INVAL and NAN2 bits are set.
2) If *src2* is QNaN, NaN_out is placed in *dst* and the NAN2 bit is set.
3) If *src2* is a negative, nonzero, nondenormalized number, NaN_out is placed in *dst* and the INVAL bit is set.
4) If *src2* is a signed denormalized number, signed infinity is placed in *dst* and the DIV0, INEX, and DEN2 bits are set.
5) If *src2* is signed 0, signed infinity is placed in *dst* and the DIV0 and INFO bits are set. The Newton-Rhapson approximation cannot be used to calculate the square root of 0 because infinity multiplied by 0 is invalid.
6) If *src2* is positive infinity, positive 0 is placed in *dst*.

Pipeline

Pipeline Stage	E1
Read	*src2*
Written	*dst*
Unit in use	.S

Instruction Type Single-cycle

Delay Slots 0

Functional Unit Latency 1

Example 1

```
RSQRSP .S1 A1,A2
```

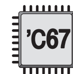
	Before instruction			1 cycle after instruction	
A1	4080 0000h	4.0	A1	4080 0000h	4.0
A2	XXXX XXXXh		A2	3F00 0000h	0.5

Example 2

```
RSQRSP .S2X A1,B2
```

	Before instruction			1 cycle after instruction	
A1	4109 999Ah	8.6	A1	4109 999Ah	8.6
B2	XXXX XXXXh		B2	3EAE 8000h	0.34082031

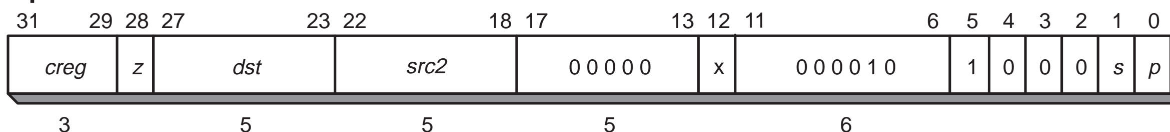
Syntax **SPDP** (.unit) *src2*, *dst*



.unit = .S1 or .S2

Opcode map field used...	For operand type...	Unit
src2 *dst*	xsp dp	.S1, .S2

Opcode



Description The single-precision value in *src2* is converted to a double-precision value and placed in *dst*.

Execution

if (cond) dp(*src2*) → *dst*
else nop

Notes:

1) If *src2* is SNaN, NaN_out is placed in *dst* and the INVAL and NAN2 bits are set.
2) If *src2* is QNaN, NaN_out is placed in *dst* and the NAN2 bit is set.
3) If *src2* is a signed denormalized number, signed 0 is placed in *dst* and the INEX and DEN2 bits are set.
4) If *src2* is signed infinity, INFO bit is set.
5) No overflow or underflow can occur.

Pipeline

Pipeline Stage	E1	E2
Read	*src2*	
Written	*dst_l*	*dst_h*
Unit in use	.S	

If *dst* is used as the source for the **ADDDP**, **CMPEQDP**, **CMPLTDP**, **CMPGTDP**, **MPYDP**, or **SUBDP** instruction, the number of delay slots can be reduced by one, because these instructions read the lower word of the DP source one cycle before the upper word of the DP source.

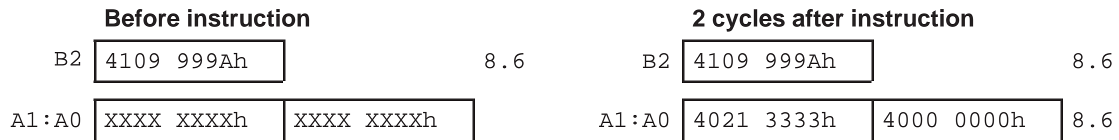
Instruction Type 2-cycle DP

Delay Slots 1

Functional Unit Latency 1

Example

```
SPDP      .S1X      B2,A1:A0
```



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

.unit = .L1 or .L2

'C67

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

Opcode

31–29	28	27–23	22–18	17–13	12	11–5	4	3	2	1	0
creg	*z*	*dst*	*src2*	0 0 0 0 0	x	0 0 0 1 0 1 0	1	1	0	*s*	*p*
3		5	5	5		7					

Description The single-precision value in *src2* is converted to an integer and placed in *dst*.

Execution

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

Notes:

1) If *src2* is NaN, the maximum signed integer (7FFF FFFFh or 8000 0000h) is placed in *dst* and the INVAL bit is set.

2) If *src2* is signed infinity or if overflow occurs, the maximum signed integer (7FFF FFFFh or 8000 0000h) is placed in *dst* and the INEX and OVER bits are set. Overflow occurs if *src2* is greater than $2^{31} - 1$ or less than -2^{31}.

3) If *src2* is denormalized, 0000 0000h is placed in *dst* and INEX and DEN2 bits are set.

4) If rounding is performed, the INEX bit is set.

Pipeline

Pipeline Stage	E1	E2	E3	E4
Read	*src2*			
Written				*dst*
Unit in use	.L			

Instruction Type 4-cycle

Delay Slots 3

Functional Unit Latency 1

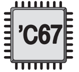
Example

```
SPINT .L1 A1,A2
```

	Before instruction			4 cycles after instruction	
A1	4109 9999Ah	8.6	A1	4109 999Ah	8.6
A2	XXXX XXXXh		A2	0000 0009h	9

Syntax

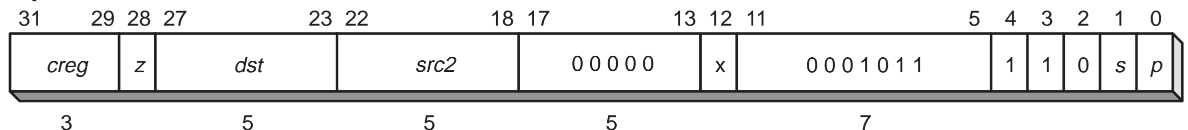
SPTRUNC (.unit) *src2*, *dst*



.unit = .L1 or .L2

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

Opcode



Description

The single-precision value in *src2* is converted to an integer and placed in *dst*. This instruction operates like **SPINT** except that the rounding modes in the FADCR are ignored, and round toward zero (truncate) is always used.

Execution

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

Notes:

1) If *src2* is NaN, the maximum signed integer (7FFF FFFFh or 8000 0000h) is placed in *dst* and the INVAL bit is set.
2) If *src2* is signed infinity or if overflow occurs, the maximum signed integer (7FFF FFFFh or 8000 0000h) is placed in *dst* and the INEX and OVER bits are set. Overflow occurs if *src2* is greater than $2^{31} - 1$ or less than -2^{31}.
3) If *src2* is denormalized, 0000 0000h is placed in *dst* and INEX and DEN2 bits are set.
4) If rounding is performed, the INEX bit is set.

Pipeline

Pipeline Stage	E1	E2	E3	E4
Read	*src2*			
Written				*dst*
Unit in use	.L			

Instruction Type 4-cycle

Delay Slots 3

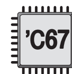
Functional Unit Latency 1

Example

```
SPTRUNC .L1X B1,A2
```

	Before instruction			4 cycles after instruction	
B1	4109 9999Ah	8.6	B1	4109 999Ah	8.6
A2	XXXX XXXXh		A2	0000 0008h	8

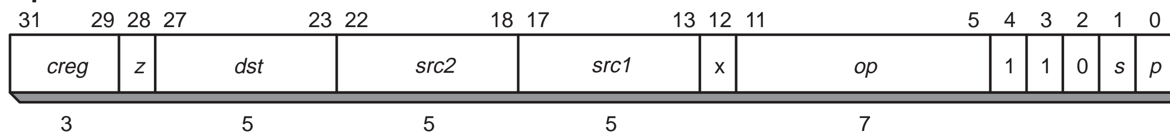
Syntax **SUBDP** (.unit) *src1*, *src2*, *dst*



.unit = .L1 or .L2

Opcode map field used...	For operand type...	Unit	Opfield
src1 *src2* *dst*	dp xdp dp	.L1, .L2	0011001
src1 *src2* *dst*	xdp dp dp	.L1, .L2	0011101

Opcode



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

Notes:

1) If rounding is performed, the INEX bit is set.
2) If one source is SNaN or QNaN, the result is NaN_out. If either source is SNaN, the INVAL bit is set also.
3) If both sources are +infinity or –infinity, the result is NaN_out and the INVAL bit is set.
4) If one source is signed infinity and the other source is anything except NaN or signed infinity of the same sign, the result is signed infinity and the INFO bit is set.
5) If overflow occurs, the INEX and OVER bits are set and the results are set as follows (LFPN is the largest floating-point number):

	Overflow Output Rounding Mode			
Result Sign	**Nearest Even**	**Zero**	**+Infinity**	**–Infinity**
+	+infinity	+LFPN	+infinity	+LFPN
–	–infinity	–LFPN	–LFPN	–infinity

6) If underflow occurs, the INEX and UNDER bits are set and the results are set as follows (SPFN is the smallest floating-point number):

	Underflow Output Rounding Mode			
Result Sign	**Nearest Even**	**Zero**	**+Infinity**	**–Infinity**
+	+0	+0	+SFPN	+0
–	–0	–0	–0	–SFPN

7) If the sources are equal numbers of the same sign, the result is +0 unless the rounding mode is –infinity, in which case the result is –0.
8) If the sources are both 0 with opposite signs or both denormalized with opposite signs, the sign of the result is the same as the sign of *src1*.
9) A signed denormalized source is treated as a signed 0 and the DENn bit is set. If the other source is not NaN or signed infinity, the INEX bit is also set.

Pipeline

Pipeline Stage	E1	E2	E3	E4	E5	E6	E7
Read	*src1_l* *src2_l*	*src1_h* *src2_h*					
Written						*dst_l*	*dst_h*
Unit in use	.L	.L					

If *dst* is used as the source for the **ADDDP**, **CMPEQDP**, **CMPLTDP**, **CMPGTDP**, **MPYDP**, or **SUBDP** instruction, the number of delay slots can be reduced by one, because these instructions read the lower word of the DP source one cycle before the upper word of the DP source.

Instruction Type ADDDP/SUBDP

Delay Slots 6

Functional Unit Latency 2

Example

```
SUBDP .L1X B1:B0,A3:A2,A5:A4
```

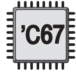
Before instruction

B1:B0	4021 3333h	3333 3333h	8.6
A3:A2	C004 0000h	0000 0000h	−2.5
A5:A4	XXXX XXXXh	XXXX XXXXh	

7 cycles after instruction

B1:B0	4021 3333h	3333 3333h	8.6
A3:A2	C004 0000h	0000 0000h	−2.5
A5:A4	4026 3333h	3333 3333h	11.1

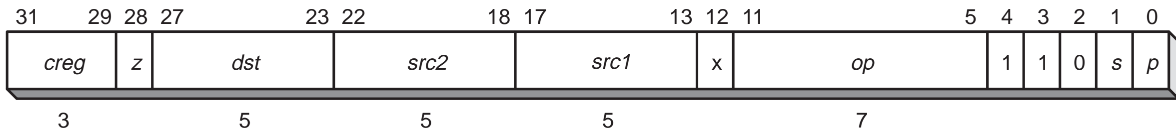
Syntax **SUBSP** (.unit) *src1*, *src2*, *dst*



.unit = .L1 or .L2

Opcode map field used...	For operand type...	Unit	Opfield
src1 *src2* *dst*	sp xsp sp	.L1, .L2	0010001
src1 *src2* *dst*	xsp sp sp	.L1, .L2	0010101

Opcode



Execution

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

Notes:

1) If rounding is performed, the INEX bit is set.
2) If one source is SNaN or QNaN, the result is NaN_out. If either source is SNaN, the INVAL bit is set also.
3) If both sources are +infinity or –infinity, the result is NaN_out and the INVAL bit is set.
4) If one source is signed infinity and the other source is anything except NaN or signed infinity of the same sign, the result is signed infinity and the INFO bit is set.
5) If overflow occurs, the INEX and OVER bits are set and the results are set as follows (LFPN is the largest floating-point number):

	Overflow Output Rounding Mode			
Result Sign	**Nearest Even**	**Zero**	**+Infinity**	**–Infinity**
+	+infinity	+LFPN	+infinity	+LFPN
–	–infinity	–LFPN	–LFPN	–infinity

6) If underflow occurs, the INEX and UNDER bits are set and the results are set as follows (SPFN is the smallest floating-point number):

	Underflow Output Rounding Mode			
Result Sign	**Nearest Even**	**Zero**	**+Infinity**	**–Infinity**
+	+0	+0	+SFPN	+0
–	–0	–0	–0	–SFPN

7) If the sources are equal numbers of the same sign, the result is +0 unless the rounding mode is –infinity, in which case the result is –0.
8) If the sources are both 0 with opposite signs or both denormalized with opposite signs, the sign of the result is the same as the sign of *src1*.
9) A signed denormalized source is treated as a signed 0 and the DENn bit is set. If the other source is not NaN or signed infinity, the INEX bit is also set.

Pipeline

Pipeline Stage	E1	E2	E3	E4
Read	*src1* *src2*			
Written				*dst*
Unit in use	.L			

Instruction Type

4-cycle

Delay Slots

3

Functional Unit Latency

1

Example

```
SUBSP .L1X A2,B1,A3
```

Before instruction			**4 cycles after instruction**		
A2	4109 999Ah		A2	4109 999Ah	8.6
B1	C020 0000h		B1	C020 0000h	−2.5
A3	XXXX XXXXh		A3	4131 999Ah	11.1

Chapter 5

TMS320C62x Pipeline

The 'C62x 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. The 'C62x generally requires fewer execution phases than the 'C67x because the 'C62x executes only fixed-point instructions.

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 *TMS320C62x/C67x Programmer's Guide*.

Topic **Page**

5.1 Pipeline Operation Overview 5-2

5.2 Pipeline Execution of Instruction Types 5-11

5.3 Performance Considerations 5-18

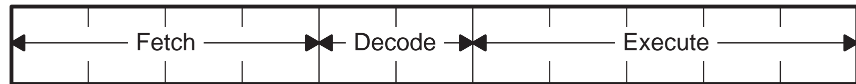
5.1 Pipeline Operation Overview

The pipeline phases are divided into three stages:

- Fetch
- Decode
- Execute

All instructions in the ’C62x 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 ’C62x pipeline are shown in Figure 5–1.

Figure 5–1. Fixed-Point Pipeline Stages



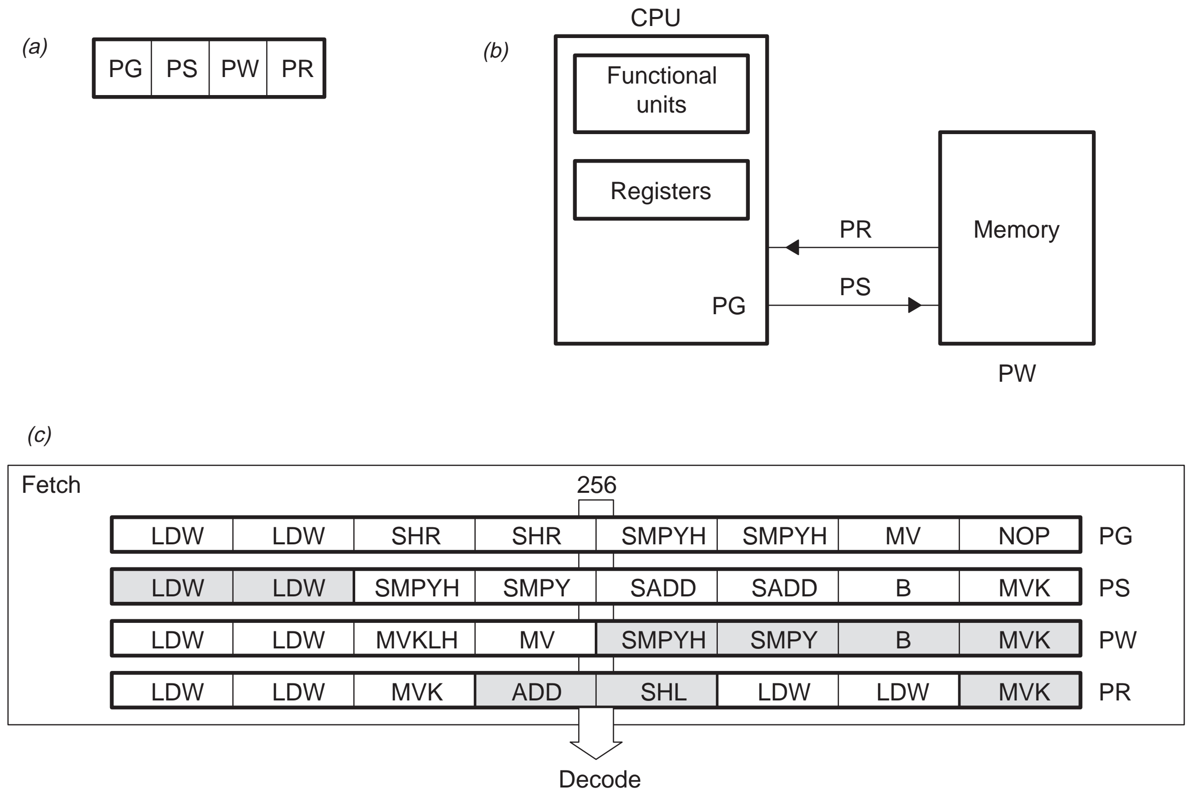
5.1.1 Fetch

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 ’C62x 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 5–2(a) shows the fetch phases in sequential order from left to right. Figure 5–2(b) is 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 5–2(c) shows fetch packets flowing through the phases of the fetch stage of the pipeline. In Figure 5–2(c), the first fetch packet (in PR) is made up of four execute packets, and the second and third fetch packets (in PW and PS) contain two execute packets each. The last fetch packet (in PG) contains a single execute packet of eight instructions.

Figure 5–2. Fetch Phases of the Pipeline



5.1.2 Decode

The decode phases of the pipeline are:

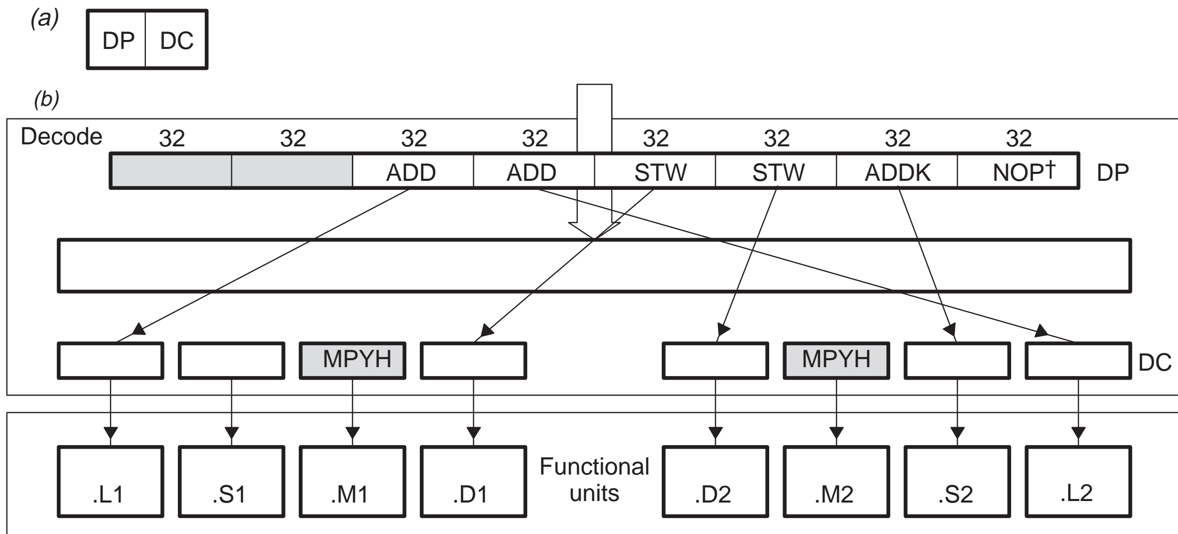
- **DP:** Instruction dispatch
- **DC:** Instruction decode

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 5–3(a) shows the decode phases in sequential order from left to right. Figure 5–3(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 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 5–3. Decode Phases of the Pipeline



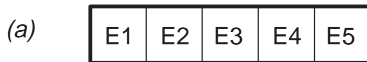
† NOP is not dispatched to a functional unit.

5.1.3 Execute

The execute portion of the fixed-point 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 5.2, *Pipeline Execution of Instruction Types*. Figure 5–4(a) shows the execute phases of the pipeline in sequential order from left to right. Figure 5–4(b) shows the portion of the functional block diagram in which execution occurs.

Figure 5–4. Execute Phases of the Pipeline and Functional Block Diagram of the TMS320C62x

(a)



(b)

5.1.4 Summary of Pipeline Operation

Figure 5–5 shows all the phases in each stage of the ’C62x pipeline in sequential order, from left to right.

Figure 5–5. Fixed-Point Pipeline Phases

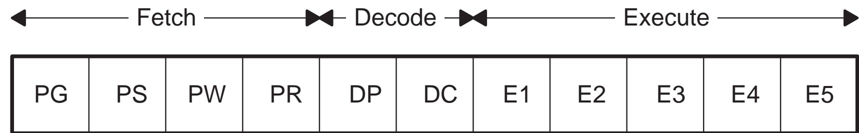
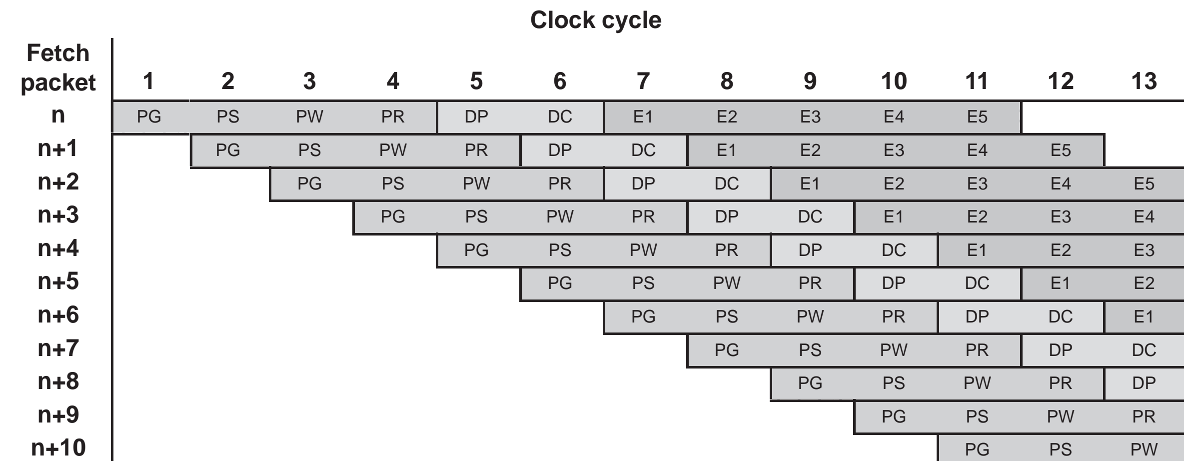


Figure 5–6 shows an example of the pipeline flow of consecutive fetch packets that contain eight parallel instructions. In this case, where the pipeline is full, all instructions in a fetch packet are in parallel and split into one execute packet per fetch packet. The fetch packets flow in lockstep fashion through each phase of the pipeline.

For example, examine cycle 7 in Figure 5–6. When the instructions from FP n reach E1, the instructions in the execute packet from FPn +1 are being decoded. FP n + 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 5.3, *Performance Considerations*, on page 5-18 for additional detail on code flowing through the pipeline.

Figure 5–6. 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 5–1 summarizes the pipeline phases and what happens in each.

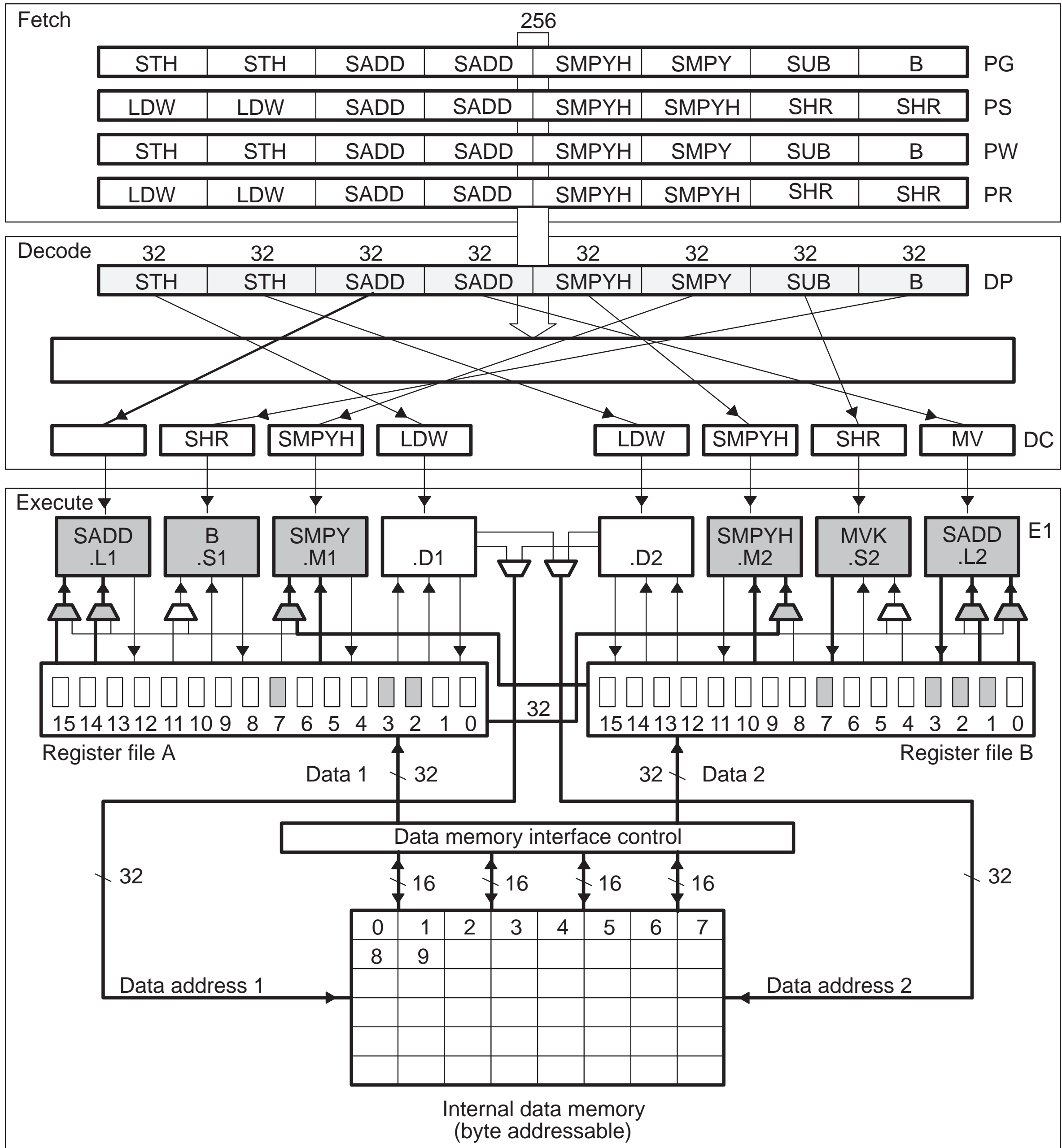
Table 5–1. Operations Occurring During Fixed-Point Pipeline Phases

Stage	Phase	Symbol	During This Phase	Instruction Type 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 5–7 shows a ’C62x functional block diagram laid out vertically by stages of the pipeline.

Figure 5–7. Functional Block Diagram of TMS320C62x 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 ’C62x. Figure 5–7 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 that 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 5–1.

Example 5–1. Execute Packet in Figure 5–7

```
        SADD    .L1     A2,A7,A2        ; E1 Phase
||      SADD    .L2     B2,B7,B2
||      SMPYH   .M2X    B3,A3,B2
||      SMPY    .M1X    B3,A3,A2
||      B       .S1     LOOP1
||      MVK     .S2     117,B1

        LDW     .D2     *B4++,B3        ; DC Phase
||      LDW     .D1     *A4++,A3
||      MV      .L2X    A1,B0
||      SMPYH   .M1     A2,A2,A0
||      SMPYH   .M2     B2,B2,B10
||      SHR     .S1     A2,16,A5
||      SHR     .S2     B2,16,B5

LOOP1:

        STH     .D1     A5,*A8++[2]     ; DP, PW, and PG
Phases
||      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] B       .S1     LOOP1
|| [B1] SUB     .S2     B1,1,B1

        LDW     .D2     *B4++,B3        : PR and PS Phases
||      LDW     .D1     *A4++,A3
||      SADD    .L1     A0,A1,A1
||      SADD    .L2     B10,B0,B0
||      SMPYH   .M1     A2,A2,A0
||      SMPYH   .M2     B2,B2,B10
||      SHR     .S1     A2,16,A5
||      SHR     .S2     B2,16,B5
```

In the DC phase portion of Figure 5–7, one box is empty because a **NOP** was the eighth instruction in the fetch packet in DC and no functional unit is needed for a **NOP**. Finally, the figure shows six functional units processing code during the same cycle of the pipeline.

Registers used by the instructions in E1 are shaded in Figure 5–7. The multiplexers used for the input operands to the functional units are also shaded in the figure. The bold crosspaths are used by the **MPY** instructions.

Most 'C62x 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 5.2, *Pipeline Execution of Instruction Types*.

5.2 Pipeline Execution of Instruction Types

The pipeline operation of the ’C62x instructions can be categorized into six instruction types. Five of these are shown in Table 5–2 (**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 5–2. 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‡

† See section 5.2.3 and 5.2.4 for more information on execution and delay slots for stores and loads.
‡ See section 5.2.5 for more information on branches.

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.

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.

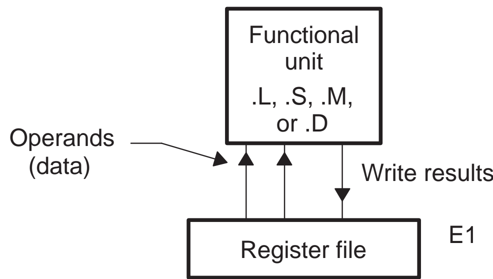
5.2.1 Single-Cycle Instructions

Single-cycle instructions complete execution during the E1 phase of the pipeline. Figure 5–8 shows the fetch, decode, and execute phases of the pipeline that single-cycle instructions use.

Figure 5–8. Single-Cycle Instruction Phases

Figure 5–9 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 5–9. Single-Cycle Execution Block Diagram



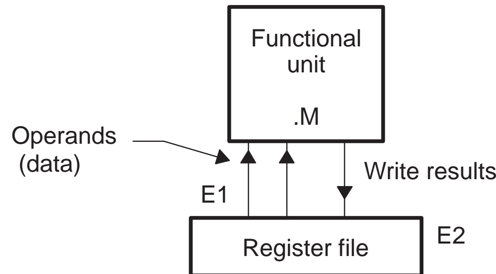
5.2.2 Multiply Instructions

Multiply instructions use both the E1 and E2 phases of the pipeline to complete their operations. Figure 5–10 shows the pipeline phases the multiply instructions use.

Figure 5–10. Multiply Instruction Phases

Figure 5–11 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 5–11. Multiply Execution Block Diagram



5.2.3 Store Instructions

Store instructions require phases E1 through E3 to complete their operations. Figure 5–12 shows the pipeline phases the store instructions use.

Figure 5–12. Store Instruction Phases

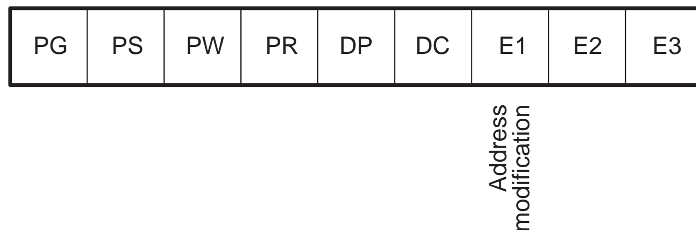
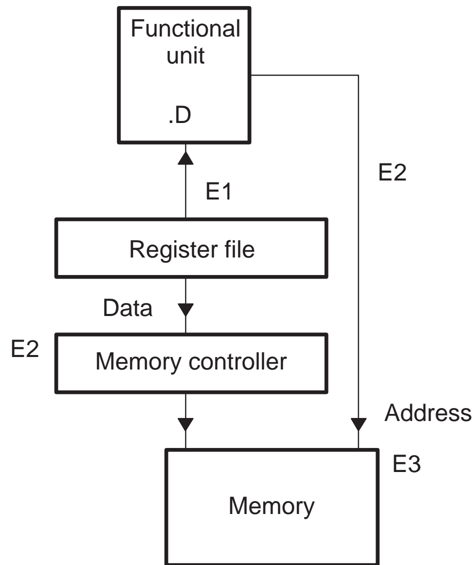


Figure 5–13 shows the operations occurring in the pipeline phases for a store. In the E1 phase, the address of the data to be stored is computed. In the E2 phase, the data and destination addresses are sent to data memory. In the E3 phase, a memory write is performed. The address modification is performed 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 5–13. 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 5.2.4.

5.2.4 Load Instructions

Data loads require all five of the pipeline execute phases to complete their operations. Figure 5–14 shows the pipeline phases the load instructions use.

Figure 5–14. Load Instruction Phases

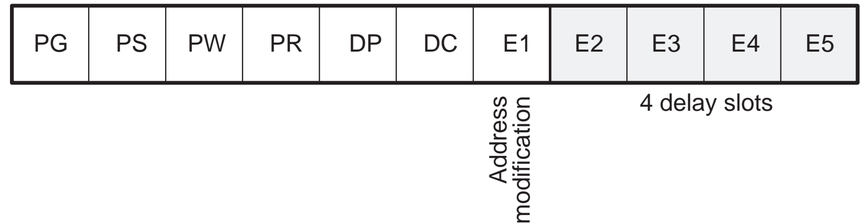
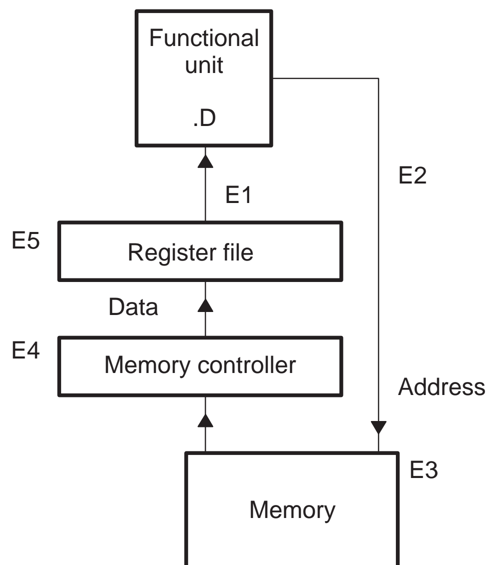


Figure 5–15 shows the operations occurring in the pipeline phases for a load. In the E1 phase, the data address pointer is modified in its register. In the E2 phase, the data address is sent to data memory. In the E3 phase, a memory read at that address is performed.

Figure 5–15. 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.

5.2.5 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 5–16 shows the pipeline phases used by the branch instruction and branch target code. The delay slots are shaded.

Figure 5–16. Branch Instruction Phases

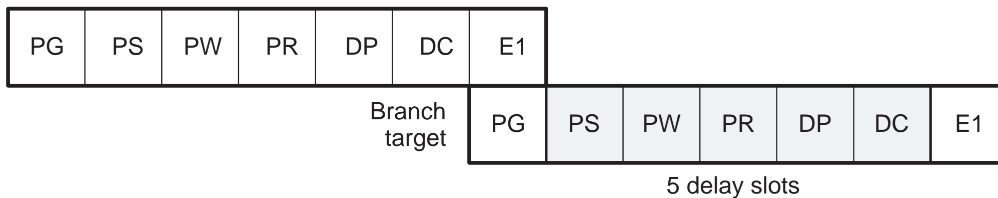
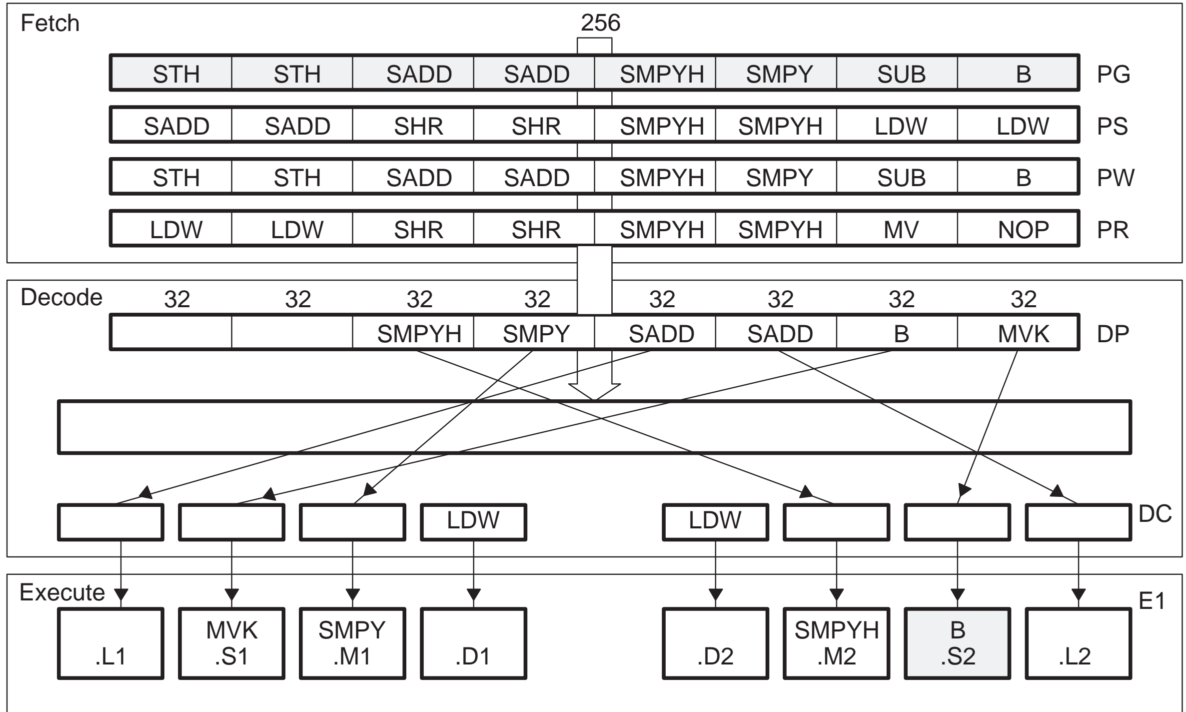


Figure 5–17 shows a branch execution block diagram. If a branch is in the E1 phase of the pipeline (in the .S2 unit in the figure), its branch target is in the fetch packet that is in PG during that same cycle (shaded in the figure). Because the branch target has to wait until it reaches the E1 phase to begin execution, the branch takes five delay slots before the branch target code executes.

Figure 5–17. Branch Execution Block Diagram



5.3 Performance Considerations

The ’C62x 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 5.3.2 covers the effect of including a multicycle **NOP** in an individual EP.

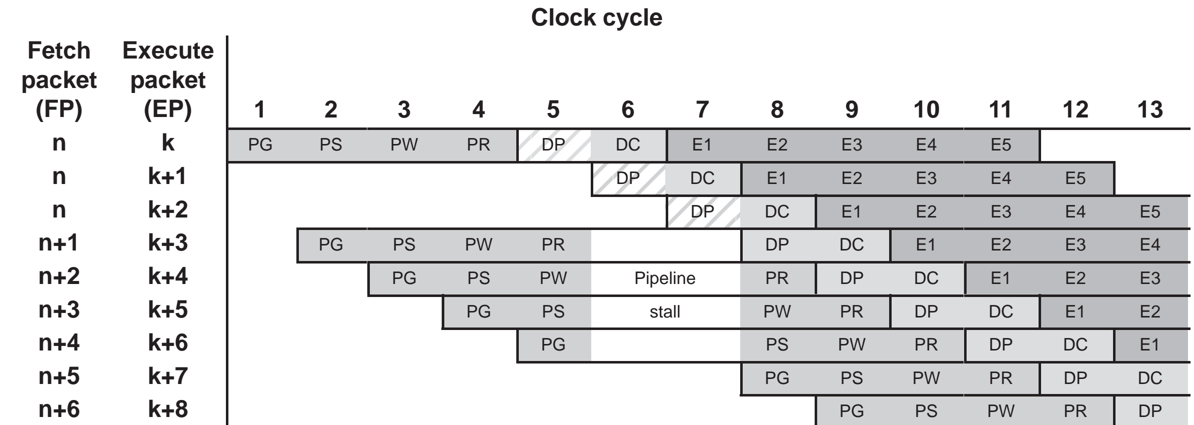
Finally, the effect of the memory system on the operation of the pipeline is considered. The access of program and data memory is discussed, along with memory stalls.

5.3.1 Pipeline Operation With Multiple Execute Packets in a Fetch Packet

Again referring to Figure 5–6 on page 5-6, pipeline operation is shown with eight instructions in every fetch packet. Figure 5–18, however, shows the pipeline operation with a fetch packet that contains multiple execute packets. Code for Figure 5–18 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 5–18. Pipeline Operation: Fetch Packets With Different Numbers of Execute Packets



In Figure 5–18, 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.

5.3.2 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 5–19 shows how a multicycle **NOP** can drive the execution of other instructions in the same execute packet. Figure 5–19(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 5–19(b) shows the replacement of the single-cycle **NOP** with a multicycle **NOP** (**NOP** 5) in the same execute packet. 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 5–19. Multicycle NOP in an Execute Packet

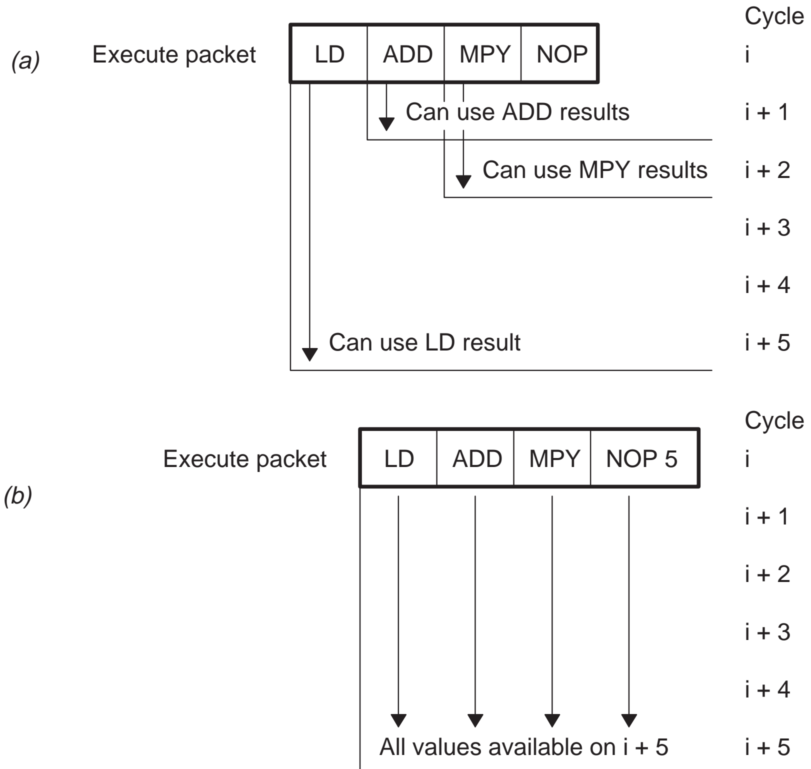
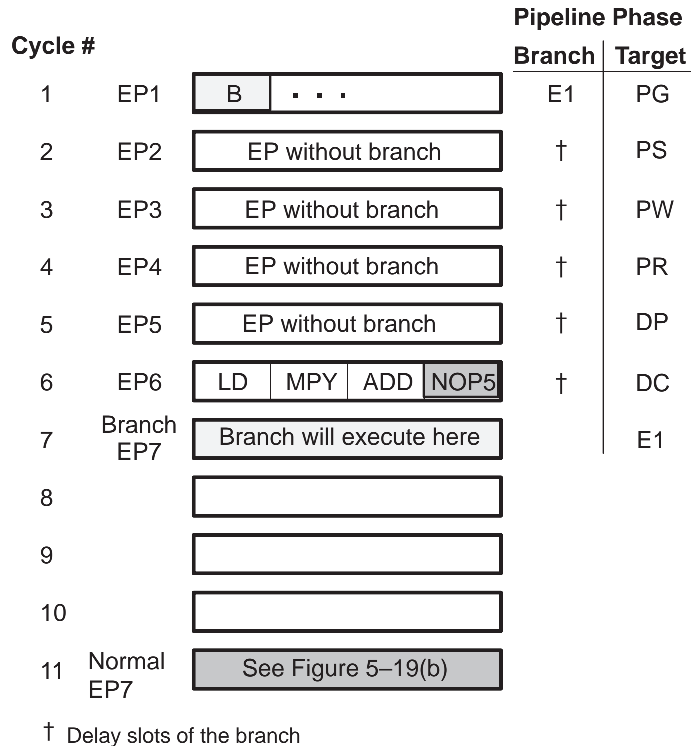


Figure 5–20 shows how a multicycle **NOP** can be affected by a branch. If the delay slots of a branch finish while a multicycle **NOP** is still dispatching **NOP**s into the pipeline, the branch overrides the multicycle **NOP** and the branch target begins execution five delay slots after the branch was issued.

*Figure 5–20. Branching and Multicycle **NOP**s*



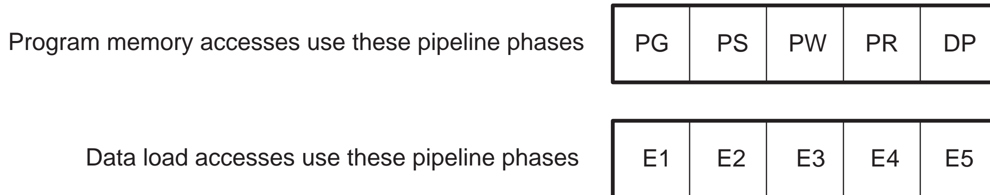
In one case, execute packet 1 (EP1) does not have a branch. The **NOP** 5 in EP6 will force the CPU to wait until cycle 11 to execute EP7.

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.

5.3.3 Memory Considerations

The ’C62x 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 into multiple phases. This enables the ’C62x to access memory at a high speed. These phases are shown in Figure 5–21.

Figure 5–21. 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 ’C62x and perform the same types of operations (listed in Table 5–3) to accommodate those memories. Table 5–3 shows the operation of program fetches pipeline versus the operation of a data load.

Table 5–3. 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

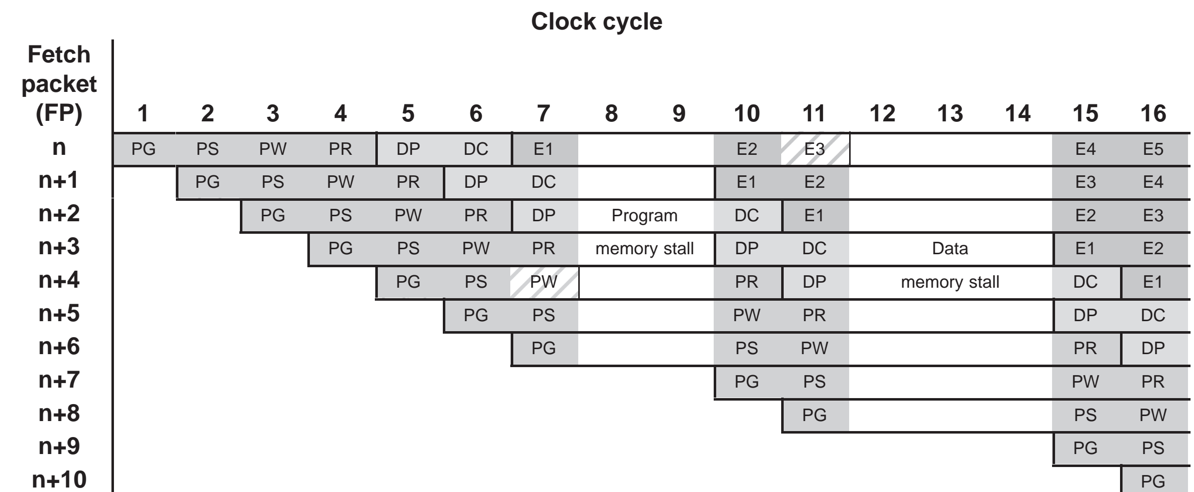
Depending on the type of memory and the time required to complete an access, the pipeline may stall to ensure proper coordination of data and instructions. This is discussed in section 5.3.3.1, *Memory Stalls.*

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 5.3.3.2, *Memory Bank Hits*.

5.3.3.1 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 5–22 illustrates this point.

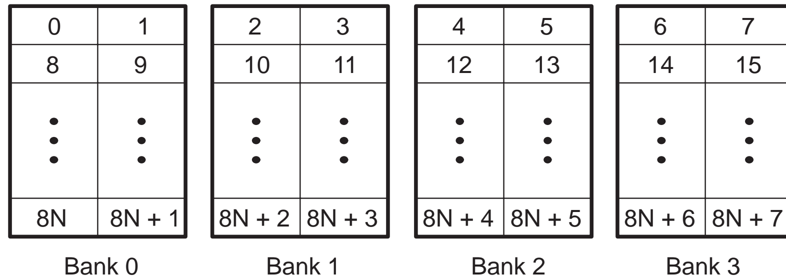
Figure 5–22. Program and Data Memory Stalls



5.3.3.2 Memory Bank Hits

Most 'C62x devices use an interleaved memory bank scheme, as shown in Figure 5–23. 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 5–23. 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 5–2. 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 5–4 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 *TMS320C62x/C67x Programmer's Guide.*

Example 5–2. 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
```

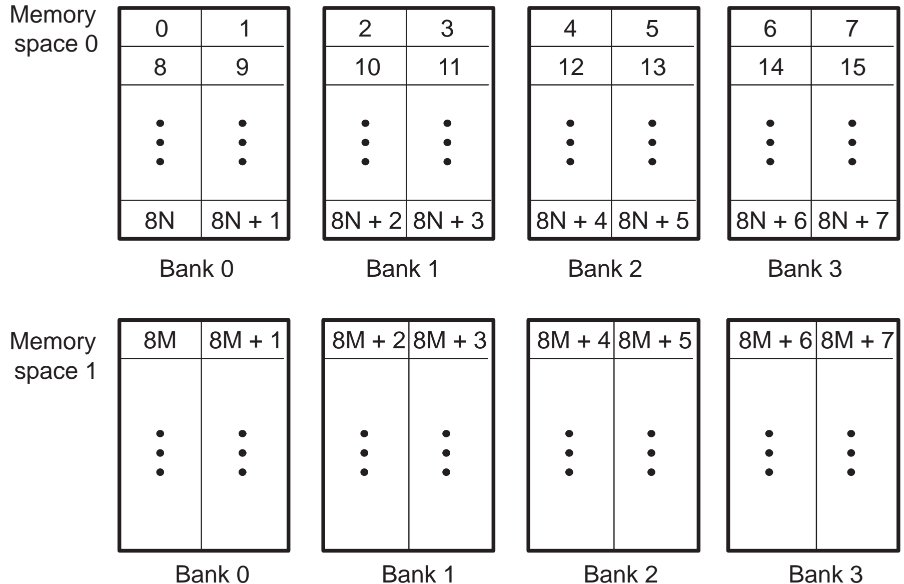
Table 5–4. Loads in Pipeline From Example 5–2

	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

† Stall due to memory bank hit

For devices that have more than one memory space (see Figure 5–24), 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 5–24. 4-Bank Interleaved Memory With Two Memory Spaces



The internal memory of the ’C62x family varies from device to device. See the *TMS320C6201/C6701 Peripherals Reference Guide* to determine the memory spaces in your particular device.

Chapter 6

TMS320C67x Pipeline

The 'C67x 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. The TMS320C67x generally has more execution phases than the 'C62x because it processes floating-point instructions.

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 *TMS320C62x/C67x Programmer's Guide*.

Topic **Page**

6.1 Pipeline Operation Overview 6-2

6.2 Pipeline Execution of Instruction Types 6-13

6.3 Functional Unit Hazards 6-20

6.4 Performance Considerations 6-52

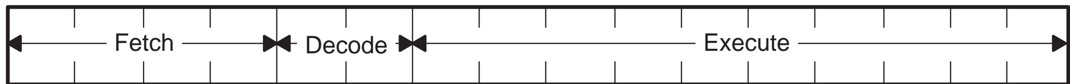
6.1 Pipeline Operation Overview

The pipeline phases are divided into three stages:

- Fetch
- Decode
- Execute

All instructions in the ’C67x 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 ’C67x pipeline are shown in Figure 6–1.

Figure 6–1. Floating-Point Pipeline Stages



6.1.1 Fetch

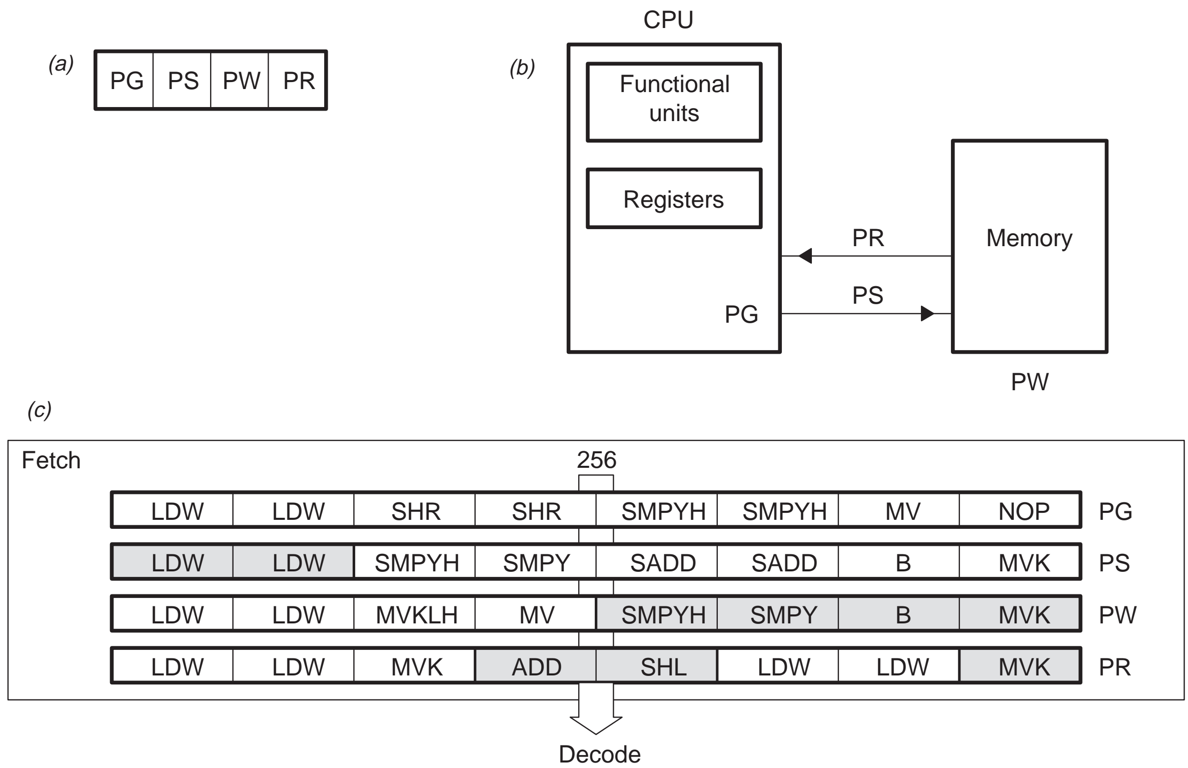
The fetch phases of the pipeline are:

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

The ’C67x 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 6–2(a) shows the fetch phases in sequential order from left to right. Figure 6–2(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 6–2(c) shows fetch packets flowing through the phases of the fetch stage of the pipeline. In Figure 6–2(c), the first fetch packet (in PR) is made up of four execute packets, and the second and third fetch packets (in PW and PS) contain two execute packets each. The last fetch packet (in PG) contains a single execute packet of eight single-cycle instructions.

Figure 6–2. Fetch Phases of the Pipeline



6.1.2 Decode

The decode phases of the pipeline are:

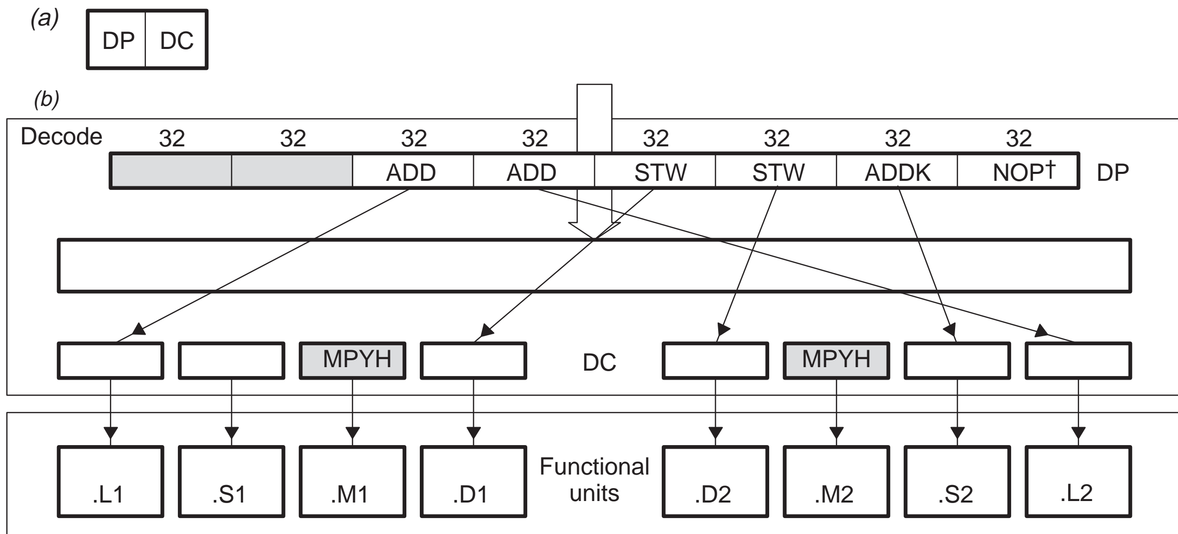
- **DP:** Instruction dispatch
- **DC:** Instruction decode

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 6–3(a) shows the decode phases in sequential order from left to right. Figure 6–3(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 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 6–3. Decode Phases of the Pipeline

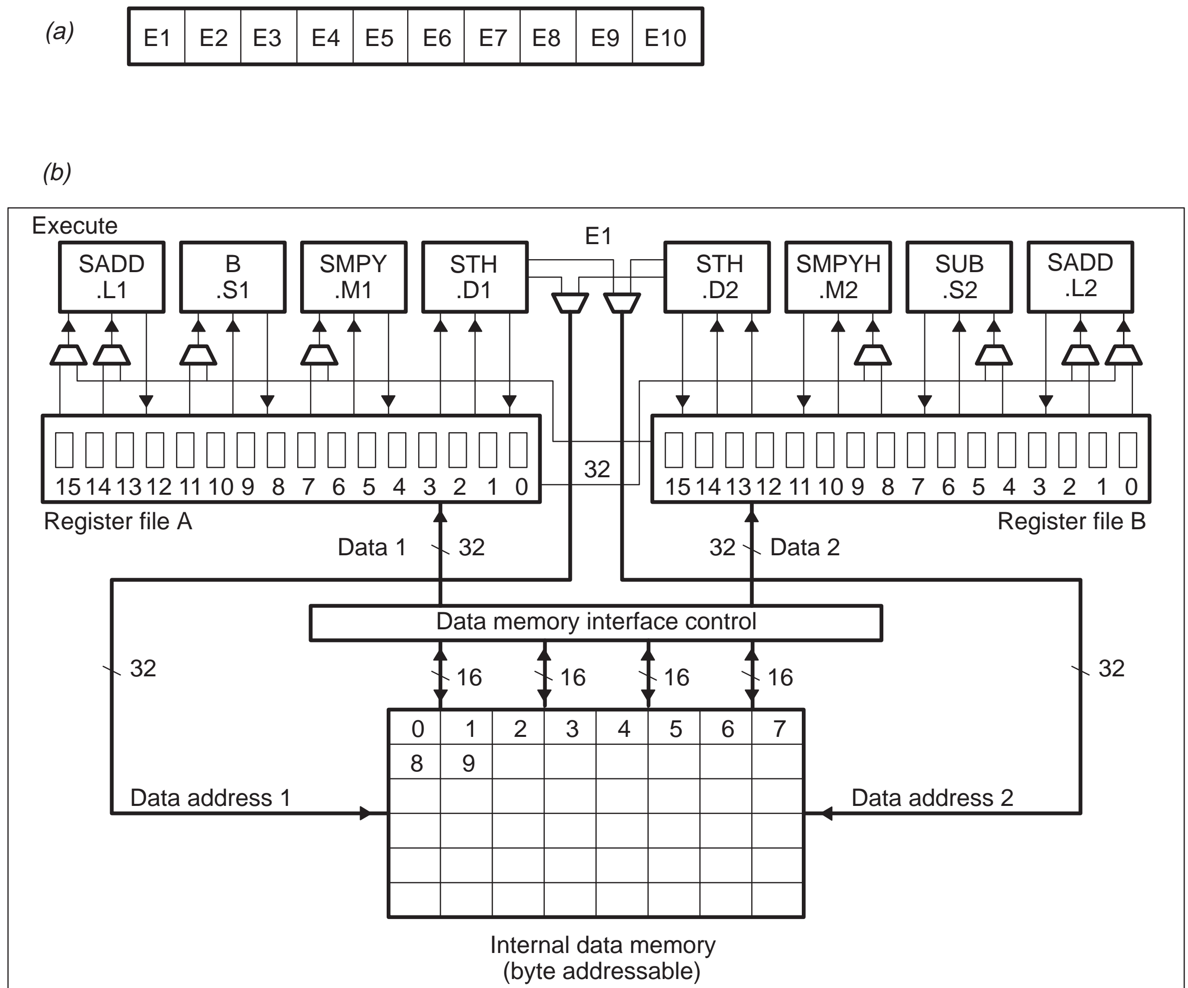


† NOP is not dispatched to a functional unit.

6.1.3 Execute

The execute portion of the floating-point pipeline is subdivided into ten phases (E1–E10), as compared to the fixed-point pipeline's five phases. 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 6.2, *Pipeline Execution of Instruction Types*. Figure 6–4(a) shows the execute phases of the pipeline in sequential order from left to right. Figure 6–4(b) shows the portion of the functional block diagram in which execution occurs.

Figure 6–4. Execute Phases of the Pipeline and Functional Block Diagram of the TMS320C67x



6.1.4 Summary of Pipeline Operation

Figure 6–5 shows all the phases in each stage of the ’C67x pipeline in sequential order, from left to right.

Figure 6–5. Floating-Point Pipeline Phases

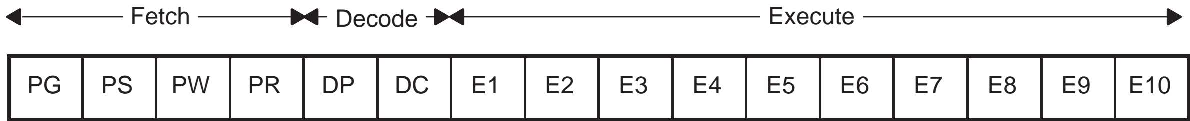
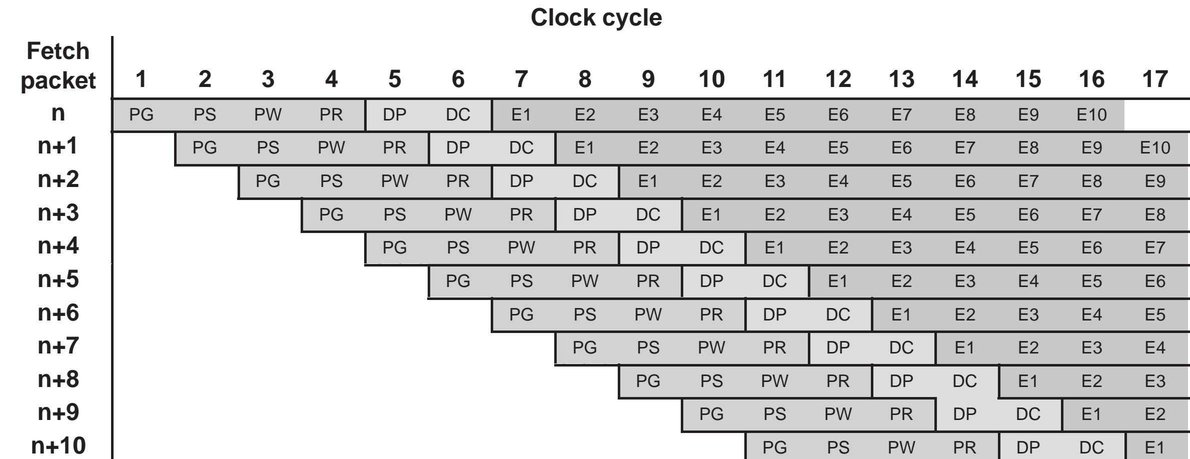


Figure 6–6 shows an example of the pipeline flow of consecutive fetch packets that contain eight parallel instructions. In this case, where the pipeline is full, all instructions in a fetch packet are in parallel and split into one execute packet per fetch packet. The fetch packets flow in lockstep fashion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

For example, examine cycle 7 in Figure 6–6. When the instructions from FP n reach E1, the instructions in the execute packet from FPn +1 are being decoded. FP n + 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 6.4, *Performance Considerations*, on page 6-52 for additional detail on code flowing through the pipeline.

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



Fetch packet	Clock cycle 1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
n	PG	PS	PW	PR	DP	DC	E1	E2	E3	E4	E5	E6	E7	E8	E9	E10	
n+1		PG	PS	PW	PR	DP	DC	E1	E2	E3	E4	E5	E6	E7	E8	E9	E10
n+2			PG	PS	PW	PR	DP	DC	E1	E2	E3	E4	E5	E6	E7	E8	E9
n+3				PG	PS	PW	PR	DP	DC	E1	E2	E3	E4	E5	E6	E7	E8
n+4					PG	PS	PW	PR	DP	DC	E1	E2	E3	E4	E5	E6	E7
n+5						PG	PS	PW	PR	DP	DC	E1	E2	E3	E4	E5	E6
n+6							PG	PS	PW	PR	DP	DC	E1	E2	E3	E4	E5
n+7								PG	PS	PW	PR	DP	DC	E1	E2	E3	E4
n+8									PG	PS	PW	PR	DP	DC	E1	E2	E3
n+9										PG	PS	PW	PR	DP	DC	E1	E2
n+10											PG	PS	PW	PR	DP	DC	E1

Table 6–1 summarizes the pipeline phases and what happens in each.

Table 6–1. Operations Occurring During Floating-Point Pipeline Phases

Stage	Phase	Symbol	During This Phase	Instruction Type Completed
Program fetch	Program address generation	PG	The address of the fetch packet is determined.	
	Program address sent	PS	The address of the fetch packet is sent to the 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 of the fetch packet is determined and sent to the appropriate functional unit 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 the register file.† For branch instructions, branch fetch packet in PG phase is affected.† For single-cycle instructions, results are written to a register file.† For DP compare, ADDDP/SUBDP, and MPYDP instructions, the lower 32-bits of the sources are read. For all other instructions, the sources are read.† For 2-cycle DP instructions, the lower 32 bits of the result are written to a register file.†	Single-cycle

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

Table 6–1. Operations Occurring During Floating-Point Pipeline Phases (Continued)

Stage	Phase	Symbol	During This Phase	Instruction Type Completed
	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 SCR if saturation occurs.† For multiply, 2-cycle DP, and DP compare instructions, results are written to a register file.† For DP compare and ADDDP/SUBDP instructions, the upper 32 bits of the source are read.† For the MPYDP instruction, the lower 32 bits of *src1* and the upper 32 bits of *src2* are read.† For MPYI and MPYID instructions, the sources are read.†	Multiply 2-cycle DP DP compare
	Execute 3	E3	Data memory accesses are performed. Any multiply instruction that saturates results sets the SAT bit in the CSR if saturation occurs.† For MPYDP instruction, the upper 32 bits of *src1* and the lower 32 bits of *src2* are read.† For MPYI and MPYID instructions, the sources are read.†	Store
	Execute 4	E4	For load instructions, data is brought to the CPU boundary For the MPYI and MPYID instructions, the sources are read.† For the MPYDP instruction, the upper 32 bits of the sources are read.† For MPYI and MPYID instructions, the sources are read.† For 4-cycle instructions, results are written to a register file.† For INTDP instruction, the lower 32 bits of the result are written to a register file.†	4-cycle
	Execute 5	E5	For load instructions, data is written into a register file.† For the INTDP instruction, the upper 32 bits of the result are written to a register file.†	Load INTDP

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

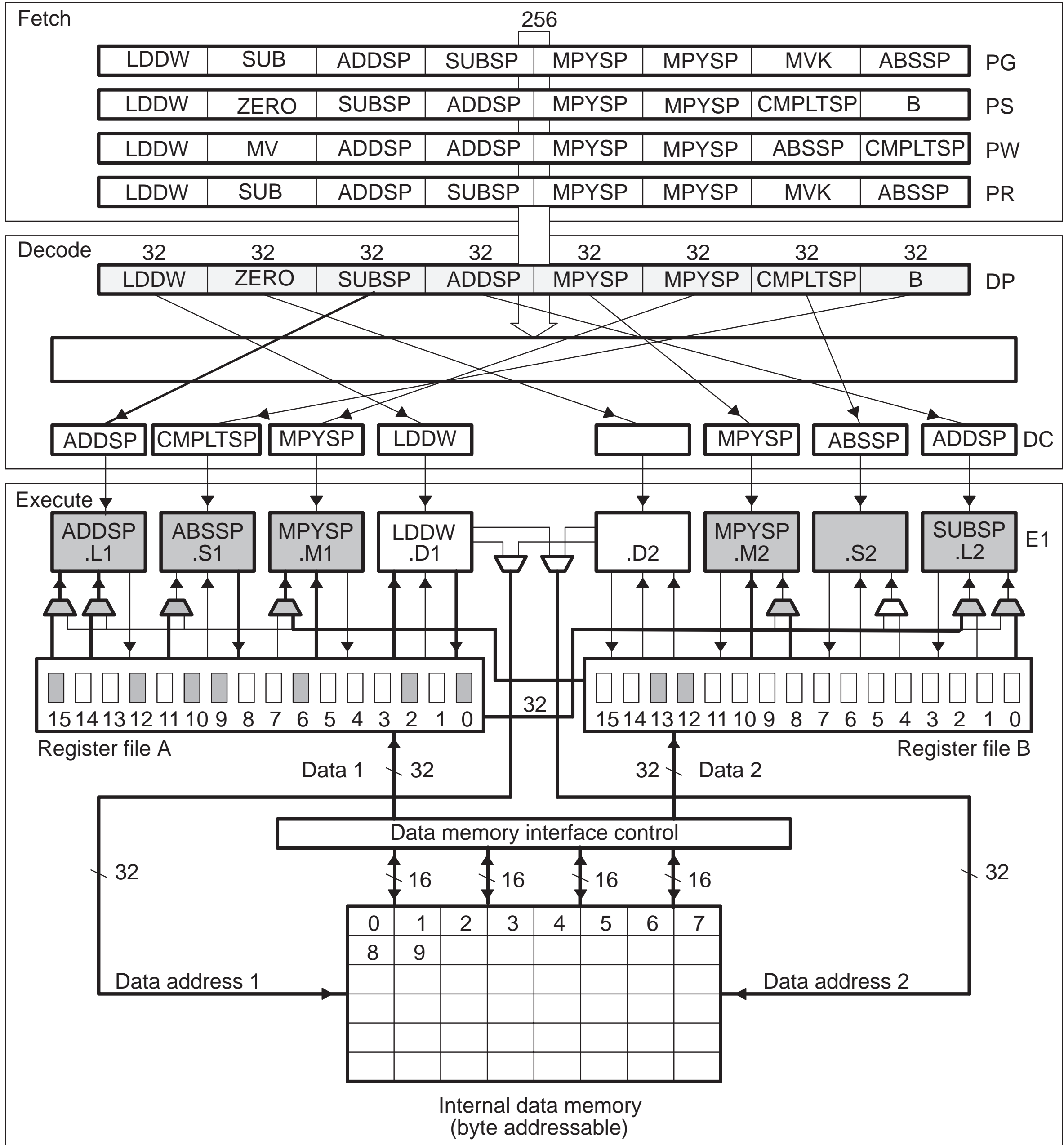
Table 6–1. Operations Occurring During Floating-Point Pipeline Phases (Continued)

Stage	Phase	Symbol	During This Phase	Instruction Type Completed
	Execute 6	E6	For ADDDP/SUBDP instructions, the lower 32 bits of the result are written to a register file.†	
	Execute 7	E7	For ADDDP/SUBDP instructions, the upper 32 bits of the result are written to a register file.†	ADDDP/ SUBDP
	Execute 8	E8	Nothing is read or written.	
	Execute 9	E9	For the MPYI instruction, the result is written to a register file.† For MPYDP and MPYID instructions, the lower 32 bits of the result are written to a register file.†	MPYI
	Execute 10	E10	For MPYDP and MPYID instructions, the upper 32 bits of the result are written to a register file.	MPYDP MPYID

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

Figure 6–7 shows a ’C67x functional block diagram laid out vertically by stages of the pipeline.

Figure 6–7. Functional Block Diagram of TMS320C67x 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 ’C67x. Figure 6–7 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 that 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 6–1.

In the DC phase portion of Figure 6–7, one box is empty because a **NOP** was the eighth instruction in the fetch packet in DC, and no functional unit is needed for a **NOP**. Finally, the figure shows six functional units processing code during the same cycle of the pipeline.

Registers used by the instructions in E1 are shaded in Figure 6–7. The multiplexers used for the input operands to the functional units are also shaded in the figure. The bold crosspaths are used by the **MPY** and **SUBSP** instructions.

Many ’C67x instructions are single-cycle instructions, which means they have only one execution phase (E1). The other instructions require more than one execute phase. The types of instructions, each of which require different numbers of execute phases, are described in section 6.2, *Pipeline Execution of Instruction Types*.

Example 6–1. Execute Packet in Figure 6–7

```
              LDDW    .D1     *A0--[4],B5:B4  ; E1 Phase
||            ADDSP   .L1     A9,A10,A12
||            SUBSP   .L2X    B12,A2,B12
||            MPYSP   .M1X    A6,B13,A11
||            MPYSP   .M2     B5,B13,B11
||            ABSSP   .S1     A12,A15

              LDDW    .D1     *A0++[5],A7:A6  ; DC Phase
||            ADDSP   .L1     A12,A11,A12
||            ADDSP   .L2     B10,B11,B12
||            MPYSP   .M1X    A4,B6,A9
||            MPYSP   .M2X    A7,B6,B9
||            CMPLTSP .S1     A15,A8,A1
||            ABSSP   .S2     B12,B15

LOOP:
    [!B2]     LDDW    .D1     *A0++[2],A5:A4  ; DP and PS Phases
||  [B2]      ZERO    .D2     B0
||            SUBSP   .L1     A12,A2,A12
||            ADDSP   .L2     B9,B12,B12
||            MPYSP   .M1X    A5,B7,A10
||            MPYSP   .M2     B4,B7,B10
||  [B0]      B       .S1     LOOP
||  [!B1]     CMPLTSP .S2     B15,B8,B1
    [!B2]     LDDW    .D1     *A0--[4],B5:B4  ; PR and PG Phases
||  [B0]      SUB     .D2     B0,2,B0
||            ADDSP   .L1     A9,A10,A12
||            SUBSP   .L2X    B12,A2,B12
||            MPYSP   .M1X    A6,B13,A11
||            MPYSP   .M2     B5,B13,B11
||            ABSSP   .S1     A12,A15
||  [A1]      MVK     .S2     1,B2

    [!B2]     LDDW    .D1     *A0++[5],A7:A6  ; PW Phase
||  [B1]      MV      .D2     B1,B2
||            ADDSP   .L1     A12,A11,A12
||            ADDSP   .L2     B10,B11,B12
||            MPYSP   .M1X    A4,B6,A9
||  [!A1]     CMPLTSP .S1     A15,A8,A1
||            ABSSP   .S2     B12,B15
```

6.2 Pipeline Execution of Instruction Types

The pipeline operation of the 'C67x instructions can be categorized into fourteen instruction types. Thirteen of these are shown in Table 6–2 (**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 and functional unit latency associated with each instruction type are listed in the bottom row.

Table 6–2. Execution Stage Length Description for Each Instruction Type

		Instruction Type				
		Single Cycle	**16 × 16 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	
	E6					
	E7					
	E8					
	E9					
	E10					
Delay slots		0	1	0†	4†	5‡
Functional unit latency		1	1	1	1	1

† See sections 6.3.7 (page 6-40) and 6.3.8 (page 6-42) for more information on execution and delay slots for stores and loads.
‡ See section 6.3.9 (page 6-44) for more information on branches.

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.

Table 6–2. Execution Stage Length Description for Each Instruction Type (Continued)

		Instruction Type			
		2-Cycle DP	**4-Cycle**	**INTDP**	**DP Compare**
Execution phases	E1	Compute the lower results and write to register	Read sources and start computation	Read sources and start computation	Read lower sources and start computation
	E2	Compute the upper results and write to register	Continue computation	Continue computation	Read upper sources, finish computation, and write results to register
	E3		Continue computation	Continue computation	
	E4		Complete computation and write results to register	Continue computation and write lower results to register	
	E5			Complete computation and write upper results to register	
	E6				
	E7				
	E8				
	E9				
	E10				
Delay slots		1	3	4	1
Functional unit latency		1	1	1	2

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.

Table 6–2. Execution Stage Length Description for Each Instruction Type (Continued)

		Instruction Type			
		ADDDP/SUBDP	**MPYI**	**MPYID**	**MPYDP**
Execution phases	E1	Read lower sources and start computation	Read sources and start computation	Read sources and start computation	Read lower sources and start computation
	E2	Read upper sources and continue computation	Read sources and continue computation	Read sources and continue computation	Read lower *src1* and upper *src2* and continue computation
	E3	Continue computation	Read sources and continue computation	Read sources and continue computation	Read lower *src2* and upper *src1* and continue computation
	E4	Continue computation	Read sources and continue computation	Read sources and continue computation	Read upper sources and continue computation
	E5	Continue computation	Continue computation	Continue computation	Continue computation
	E6	Compute the lower results and write to register	Continue computation	Continue computation	Continue computation
	E7	Compute the upper results and write to register	Continue computation	Continue computation	Continue computation
	E8		Continue computation	Continue computation	Continue computation
	E9		Complete computation and write results to register	Continue computation and write lower results to register	Continue computation and write lower results to register
	E10			Complete computation and write upper results to register	Complete computation and write upper results to register
Delay slots		6	8	9	9
Functional unit latency		2	4	4	4

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.

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.

If an instruction has a multicycle functional unit latency, it locks the functional unit for the necessary number of cycles. Any new instruction dispatched to that functional unit during this locking period causes undefined results. If an instruction with a multicycle functional unit latency has a condition that is evaluated as false during E1, it still locks the functional unit for subsequent cycles.

An instruction of the following types scheduled on cycle i has the following constraints:

DP compare	No other instruction can use the functional unit on cycles i and i + 1.
ADDDP/SUBDP	No other instruction can use the functional unit on cycles i and i + 1.
MPYI	No other instruction can use the functional unit on cycles i, i + 1, i + 2, and i + 3.
MPYID	No other instruction can use the functional unit on cycles i, i + 1, i + 2, and i + 3.
MPYDP	No other instruction can use the functional unit on cycles i, i + 1, i + 2, and i + 3.

If a cross path is used to read a source using an instruction with multicycle functional unit latency, ensure that no other instructions executing on the same side use the cross path.

An instruction of the following types scheduled on cycle i, using a cross path to read a source, has the following constraints:

DP compare	No other instruction on the same side can use the cross path on cycles i and i + 1.
ADDDP/SUBDP	No other instruction on the same side can use the cross path on cycles i and i + 1.
MPYI	No other instruction on the same side can use the cross path on cycles i, i + 1, i + 2, and i + 3.
MPYID	No other instruction on the same side can use the cross path on cycles i, i + 1, i + 2, and i + 3.
MPYDP	No other instruction on the same side can use the cross path on cycles i, i + 1, i + 2, and i + 3.

Other hazards exist because instructions have varying numbers of delay slots, and the instructions need the functional unit read and write ports for varying numbers of cycles. A read or write hazard occurs when two instructions on the same functional unit attempt to read or write, respectively, to the register file on the came cycle.

An instruction scheduled on cycle i has the following constraints:

2-cycle DP	A single-cycle instruction cannot be scheduled on the same functional unit on cycle i + 1 due to a write hazard on cycle i + 1.
	Another 2-cycle DP instruction cannot be scheduled on the same functional unit on cycle i + 1 due to a write hazard on cycle i + 1.
4-cycle	A single-cycle instruction cannot be scheduled on the same functional unit on cycle i + 3 due to a write hazard on cycle i + 3.
	A multiply (16 $\times$ 16-bit) instruction cannot be scheduled on the same functional unit on cycle i + 2 due to a write hazard on cycle i + 3.
INTDP	A single-cycle instruction cannot be scheduled on the same functional unit on cycle i + 3 or i + 4 due to a write hazard on cycle i + 3 or i + 4, respectively.
	An INTDP instruction cannot be scheduled on the same functional unit on cycle i + 1 due to a write hazard on cycle i + 1.
	A 4-cycle instruction cannot be scheduled on the same functional unit on cycle i + 1 due to a write hazard on cycle i + 1.

MPYI	A 4-cycle instruction cannot be scheduled on the same functional unit on cycle i + 4, i + 5, or i + 6.
	A MPYDP instruction cannot be scheduled on the same functional unit on cycle i + 4, i + 5, or i + 6.
	A multiply (16 × 16-bit) instruction cannot be scheduled on the same functional unit on cycle i + 6 due to a write hazard on cycle i + 7.
MPYID	A 4-cycle instruction cannot be scheduled on the same functional unit on cycle i + 4, i + 5, or i + 6.
	A MPYDP instruction cannot be scheduled on the same functional unit on cycle i + 4, i + 5, or i + 6.
	A multiply (16 × 16-bit) instruction cannot be scheduled on the same functional unit on cycle i + 7 or i + 8 due to a write hazard on cycle i + 8 or i + 9, respectively.
MPYDP	A 4-cycle instruction cannot be scheduled on the same functional unit on cycle i + 4, i + 5, or i + 6.
	A MPYI instruction cannot be scheduled on the same functional unit on cycle i + 4, i + 5, or i + 6.
	A MPYID instruction cannot be scheduled on the same functional unit on cycle i + 4, i + 5, or i + 6.
	A multiply (16 × 16-bit) instruction cannot be scheduled on the same functional unit on cycle i + 7 or i + 8 due to a write hazard on cycle i + 8 or i + 9, respectively.
ADDDP/SUBDP	A single-cycle instruction cannot be scheduled on the same functional unit on cycle i + 5 or i + 6 due to a write hazard on cycle i + 5 or i + 6, respectively.
	A 4-cycle instruction cannot be scheduled on the same functional unit on cycle i + 2 or i + 3 due to a write hazard on cycle i + 5 or i + 6, respectively.
	An INTDP instruction cannot be scheduled on the same functional unit on cycle i + 2 or i + 3 due to a write hazard on cycle i + 5 or i + 6, respectively.

The 4-cycle case is important for the following single-precision floating-point instructions:

- ADDSP
- SUBSP
- SPINT
- SPTRUNC
- INTSP
- MPYSP

All of the preceding cases deal with double-precision floating-point instructions or the **MPYI** or **MPYID** instructions except for the 4-cycle case. A 4-cycle instruction consists of both single- and double-precision floating-point instructions. Therefore, the 4-cycle case is important for the following single-precision floating-point instructions:

The .S and .L units share their long write port with the load port for the 32 most significant bits of an **LDDW** load. Therefore, the **LDDW** instruction and the .S or .L unit writing a long result cannot write to the same register file on the same cycle. The **LDDW** writes to the register file on pipeline phase E5. Instructions that use a long result and use the .L and .S unit write to the register file on pipeline phase E1. Therefore, the instruction with the long result must be scheduled later than four cycles following the **LDDW** instruction if both instructions use the same side.

6.3 Functional Unit Hazards

If you wish to optimize your instruction pipeline, consider the instructions that are executed on each unit. Sources and destinations are read and written differently for each instruction. If you analyze these differences, you can make further optimization improvements by considering what happens during the execution phases of instructions that use the same functional unit in each execution packet.

The following sections provide information about what happens during each execute phase of the instructions within a category for each of the functional units.

6.3.1 .S-Unit Hazards

Table 6–3 shows the instruction hazards for single-cycle instructions executing on the .S unit.

Table 6–3. Single-Cycle .S-Unit Instruction Hazards

	Instruction Execution	
Cycle	1	2
Single-cycle	RW	
Instruction Type	**Subsequent Same-Unit Instruction Executable**	
Single-cycle		✓
DP compare		✓
2-cycle DP		✓
Branch		✓
Instruction Type	**Same Side, Different Unit, Both Using Cross Path Executable**	
Single-cycle		✓
Load		✓
Store		✓
INTDP		✓
ADDDP/SUBDP		✓
16 × 16 multiply		✓
4-cycle		✓
MPYI		✓
MPYID		✓
MPYDP		✓

Legend: (shaded) E1 phase of the single-cyle instruction
R Sources read for the instruction
W Destinations written for the instruction
✓ Next instruction can enter E1 during cycle

Table 6–4 shows the instruction hazards for DP compare instructions executing on the .S unit.

Table 6–4. DP Compare .S-Unit Instruction Hazards

	Instruction Execution		
Cycle	1	2	3
DP compare	R	RW	

Instruction Type	Subsequent Same-Unit Instruction Executable		
Single-cycle		Xrw	✓
DP compare		Xr	✓
2-cycle DP		Xrw	✓
Branch†		Xr	✓

Instruction Type	Same Side, Different Unit, Both Using Cross Path Executable		
Single-cycle		Xr	✓
Load		Xr	✓
Store		Xr	✓
INTDP		Xr	✓
ADDDP/SUBDP		Xr	✓
16 × 16 multiply		Xr	✓
4-cycle		Xr	✓
MPYI		Xr	✓
MPYID		Xr	✓
MPYDP		Xr	✓

Legend:
- ▒ E1 phase of the single-cyle instruction
- R Sources read for the instruction
- W Destinations written for the instruction
- ✓ Next instruction can enter E1 during cycle
- Xr Next instruction cannot enter E1 during cycle–read/decode hazard
- Xrw Next instruction cannot enter E1 during cycle–read/decode/write hazard
- † The branch on register instruction is the only branch instruction that reads a general-purpose register

Table 6–5 shows the instruction hazards for 2-cycle DP instructions executing on the .S unit.

Table 6–5. 2-Cycle DP .S-Unit Instruction Hazards

	Instruction Execution		
Cycle	1	2	3
2-cycle	RW	W	
Instruction Type	**Subsequent Same-Unit Instruction Executable**		
Single-cycle		Xw	✓
DP compare		✓	✓
2-cycle DP		Xw	✓
Branch		✓	✓
Instruction Type	**Same Side, Different Unit, Both Using Cross Path Executable**		
Single cycle		✓	✓
Load		✓	✓
Store		✓	✓
INTDP		✓	✓
ADDDP/SUBDP		✓	✓
16 × 16 multiply		✓	✓
4-cycle		✓	✓
MPYI		✓	✓
MPYID		✓	✓
MPYDP		✓	✓

Legend:
- ▒ E1 phase of the single-cyle instruction
- R Sources read for the instruction
- W Destinations written for the instruction
- ✓ Next instruction can enter E1 during cycle
- Xw Next instruction cannot enter E1 during cycle–write hazard

Table 6–6 shows the instruction hazards for branch instructions executing on the .S unit.

Table 6–6. Branch .S-Unit Instruction Hazards

	Instruction Execution							
Cycle	1	2	3	4	5	6	7	8
Branch†	R							
Instruction Type	**Subsequent Same-Unit Instruction Executable**							
Single-cycle		✓	✓	✓	✓	✓	✓	✓
DP compare		✓	✓	✓	✓	✓	✓	✓
2-cycle DP		✓	✓	✓	✓	✓	✓	✓
Branch		✓	✓	✓	✓	✓	✓	✓
Instruction Type	**Same Side, Different Unit, Both Using Cross Path Executable**							
Single-cycle		✓	✓	✓	✓	✓	✓	✓
Load		✓	✓	✓	✓	✓	✓	✓
Store		✓	✓	✓	✓	✓	✓	✓
INTDP		✓	✓	✓	✓	✓	✓	✓
ADDDP/SUBDP		✓	✓	✓	✓	✓	✓	✓
16 × 16 multiply		✓	✓	✓	✓	✓	✓	✓
4-cycle		✓	✓	✓	✓	✓	✓	✓
MPYI		✓	✓	✓	✓	✓	✓	✓
MPYID		✓	✓	✓	✓	✓	✓	✓
MPYDP		✓	✓	✓	✓	✓	✓	✓

Legend: (shaded) E1 phase of the single-cyle instruction
R Sources read for the instruction
✓ Next instruction can enter E1 during cycle
† The branch on register instruction is the only branch instruction that reads a general-purpose register

6.3.2 .M-Unit Hazards

Table 6–7 shows the instruction hazards for 16×16 multiply instructions executing on the .M unit.

Table 6–7. 16×16 Multiply .M-Unit Instruction Hazards

	Instruction Execution		
Cycle	1	2	3
16×16 multiply	R	W	
Instruction Type	**Subsequent Same-Unit Instruction Executable**		
16×16 multiply		✓	✓
4-cycle		✓	✓
MPYI		✓	✓
MPYID		✓	✓
MPYDP		✓	✓
Instruction Type	**Same Side, Different Unit, Both Using Cross Path Executable**		
Single-cycle		✓	✓
Load		✓	✓
Store		✓	✓
DP compare		✓	✓
2-cycle DP		✓	✓
Branch		✓	✓
4-cycle		✓	✓
INTDP		✓	✓
ADDDP/SUBDP		✓	✓

Legend:
- (shaded box) E1 phase of the single-cyle instruction
- R Sources read for the instruction
- W Destinations written for the instruction
- ✓ Next instruction can enter E1 during cycle

Table 6–8 shows the instruction hazards for 4-cycle instructions executing on the .M unit.

Table 6–8. 4-Cycle .M-Unit Instruction Hazards

	Instruction Execution				
Cycle	1	2	3	4	5
4-cycle	R			W	
Instruction Type	**Subsequent Same-Unit Instruction Executable**				
16 × 16 multiply		✓	Xw	✓	✓
4-cycle		✓	✓	✓	✓
MPYI		✓	✓	✓	✓
MPYID		✓	✓	✓	✓
MPYDP		✓	✓	✓	✓
Instruction Type	**Same Side, Different Unit, Both Using Cross Path Executable**				
Single-cycle		✓	✓	✓	✓
Load		✓	✓	✓	✓
Store		✓	✓	✓	✓
DP compare		✓	✓	✓	✓
2-cycle DP		✓	✓	✓	✓
Branch		✓	✓	✓	✓
4-cycle		✓	✓	✓	✓
INTDP		✓	✓	✓	✓
ADDDP/SUBDP		✓	✓	✓	✓

Legend:
- (shaded) E1 phase of the single-cyle instruction
- R Sources read for the instruction
- W Destinations written for the instruction
- ✓ Next instruction can enter E1 during cycle
- Xw Next instruction cannot enter E1 during cycle–write hazard

Table 6–9 shows the instruction hazards for **MPYI** instructions executing on the .M unit.

Table 6–9. MPYI .M-Unit Instruction Hazards

	Instruction Execution									
Cycle	1	2	3	4	5	6	7	8	9	10
MPYI	R	R	R	R					W	
Instruction Type	**Subsequent Same-Unit Instruction Executable**									
16 × 16 multiply		Xr	Xr	Xr	✓	✓	✓	Xw	✓	✓
4-cycle		Xr	Xr	Xr	Xu	Xw	Xu	✓	✓	✓
MPYI		Xr	Xr	Xr	✓	✓	✓	✓	✓	✓
MPYID		Xr	Xr	Xr	✓	✓	✓	✓	✓	✓
MPYDP		Xr	Xr	Xr	Xu	Xu	Xu	✓	✓	✓
Instruction Type	**Same Side, Different Unit, Both Using Cross Path Executable**									
Single-cycle		Xr	Xr	Xr	✓	✓	✓	✓	✓	✓
Load		Xr	Xr	Xr	✓	✓	✓	✓	✓	✓
Store		Xr	Xr	Xr	✓	✓	✓	✓	✓	✓
DP compare		Xr	Xr	Xr	✓	✓	✓	✓	✓	✓
2-cycle DP		Xr	Xr	Xr	✓	✓	✓	✓	✓	✓
Branch		Xr	Xr	Xr	✓	✓	✓	✓	✓	✓
4-cycle		Xr	Xr	Xr	✓	✓	✓	✓	✓	✓
INTDP		Xr	Xr	Xr	✓	✓	✓	✓	✓	✓
ADDDP/SUBDP		Xr	Xr	Xr	✓	✓	✓	✓	✓	✓

Legend:
- (shaded) E1 phase of the single-cyle instruction
- R Sources read for the instruction
- W Destinations written for the instruction
- ✓ Next instruction can enter E1 during cycle
- Xr Next instruction cannot enter E1 during cycle–read/decode hazard
- Xw Next instruction cannot enter E1 during cycle–write hazard
- Xu Next instruction cannot enter E1 during cycle–other resource conflict

Table 6–10 shows the instruction hazards for **MPYID** instructions executing on the .M unit.

Table 6–10. MPYID .M-Unit Instruction Hazards

	Instruction Execution										
Cycle	1	2	3	4	5	6	7	8	9	10	11
MPYID	R	R	R	R					W	W	
Instruction Type	**Subsequent Same-Unit Instruction Executable**										
16 × 16 multiply		Xr	Xr	Xr	✓	✓	✓	Xw	Xw	✓	✓
4-cycle		Xr	Xr	Xr	Xu	Xw	Xw	✓	✓	✓	✓
MPYI		Xr	Xr	Xr	✓	✓	✓	✓	✓	✓	✓
MPYID		Xr	Xr	Xr	✓	✓	✓	✓	✓	✓	✓
MPYDP		Xr	Xr	Xr	Xu	Xu	Xu	✓	✓	✓	✓
Instruction Type	**Same Side, Different Unit, Both Using Cross Path Executable**										
Single-cycle		Xr	Xr	Xr	✓	✓	✓	✓	✓	✓	✓
Load		Xr	Xr	Xr	✓	✓	✓	✓	✓	✓	✓
Store		Xr	Xr	Xr	✓	✓	✓	✓	✓	✓	✓
DP compare		Xr	Xr	Xr	✓	✓	✓	✓	✓	✓	✓
2-cycle DP		Xr	Xr	Xr	✓	✓	✓	✓	✓	✓	✓
Branch		Xr	Xr	Xr	✓	✓	✓	✓	✓	✓	✓
4-cycle		Xr	Xr	Xr	✓	✓	✓	✓	✓	✓	✓
INTDP		Xr	Xr	Xr	✓	✓	✓	✓	✓	✓	✓
ADDDP/SUBDP		Xr	Xr	Xr	✓	✓	✓	✓	✓	✓	✓

Legend:
- (shaded) E1 phase of the single-cyle instruction
- R Sources read for the instruction
- W Destinations written for the instruction
- ✓ Next instruction can enter E1 during cycle
- Xr Next instruction cannot enter E1 during cycle–read/decode hazard
- Xw Next instruction cannot enter E1 during cycle–write hazard
- Xu Next instruction cannot enter E1 during cycle–other resource conflict

Table 6–11 shows the instruction hazards for **MPYDP** instructions executing on the .M unit.

Table 6–11. MPYDP .M-Unit Instruction Hazards

	Instruction Execution										
Cycle	1	2	3	4	5	6	7	8	9	10	11
MPYDP	R	R	R	R					W	W	
Instruction Type	**Subsequent Same-Unit Instruction Executable**										
16 × 16 multiply		Xr	Xr	Xr	✓	✓	✓	Xw	Xw	✓	✓
4-cycle		Xr	Xr	Xr	Xu	Xw	Xw	✓	✓	✓	✓
MPYI		Xr	Xr	Xr	Xu	Xu	Xu	✓	✓	✓	✓
MPYID		Xr	Xr	Xr	Xu	Xu	Xu	✓	✓	✓	✓
MPYDP		Xr	Xr	Xr	✓	✓	✓	✓	✓	✓	✓
Instruction Type	**Same Side, Different Unit, Both Using Cross Path Executable**										
Single-cycle		Xr	Xr	Xr	✓	✓	✓	✓	✓	✓	✓
Load		Xr	Xr	Xr	✓	✓	✓	✓	✓	✓	✓
Store		Xr	Xr	Xr	✓	✓	✓	✓	✓	✓	✓
DP compare		Xr	Xr	Xr	✓	✓	✓	✓	✓	✓	✓
2-cycle DP		Xr	Xr	Xr	✓	✓	✓	✓	✓	✓	✓
Branch		Xr	Xr	Xr	✓	✓	✓	✓	✓	✓	✓
4-cycle		Xr	Xr	Xr	✓	✓	✓	✓	✓	✓	✓
INTDP		Xr	Xr	Xr	✓	✓	✓	✓	✓	✓	✓
ADDDP/SUBDP		Xr	Xr	Xr	✓	✓	✓	✓	✓	✓	✓

Legend:

(shaded)	E1 phase of the single-cyle instruction
R	Sources read for the instruction
W	Destinations written for the instruction
✓	Next instruction can enter E1 during cycle
Xr	Next instruction cannot enter E1 during cycle–read/decode hazard
Xw	Next instruction cannot enter E1 during cycle–write hazard
Xu	Next instruction cannot enter E1 during cycle–other resource conflict

6.3.3 .L-Unit Hazards

Table 6–12 shows the instruction hazards for single-cycle instructions executing on the .L unit.

Table 6–12. Single-Cycle .L-Unit Instruction Hazards

	Instruction Execution	
Cycle	1	2
Single-cycle	RW	
Instruction Type	**Subsequent Same-Unit Instruction Executable**	
Single-cycle		✓
4-cycle		✓
INTDP		✓
ADDDP/SUBDP		✓
Instruction Type	**Same Side, Different Unit, Both Using Cross Path Executable**	
Single-cycle		✓
DP compare		✓
2-cycle DP		✓
4-cycle		✓
Load		✓
Store		✓
Branch		✓
16 × 16 multiply		✓
MPYI		✓
MPYID		✓
MPYDP		✓

Legend: (shaded) E1 phase of the single-cyle instruction
R Sources read for the instruction
W Destinations written for the instruction
✓ Next instruction can enter E1 during cycle

Table 6–13 shows the instruction hazards for 4-cycle instructions executing on the .L unit.

Table 6–13. 4-Cycle .L-Unit Instruction Hazards

	Instruction Execution				
Cycle	1	2	3	4	5
4-cycle	R			W	
Instruction Type	**Subsequent Same-Unit Instruction Executable**				
Single-cycle		✓	✓	Xw	✓
4-cycle		✓	✓	✓	✓
INTDP		✓	✓	✓	✓
ADDDP/SUBDP		✓	✓	✓	✓
Instruction Type	**Same Side, Different Unit, Both Using Cross Path Executable**				
Single-cycle		✓	✓	✓	✓
DP compare		✓	✓	✓	✓
2-cycle DP		✓	✓	✓	✓
4-cycle		✓	✓	✓	✓
Load		✓	✓	✓	✓
Store		✓	✓	✓	✓
Branch		✓	✓	✓	✓
16 × 16 multiply		✓	✓	✓	✓
MPYI		✓	✓	✓	✓
MPYID		✓	✓	✓	✓
MPYDP		✓	✓	✓	✓

Legend:
- (shaded) E1 phase of the single-cyle instruction
- R Sources read for the instruction
- W Destinations written for the instruction
- ✓ Next instruction can enter E1 during cycle
- Xw Next instruction cannot enter E1 during cycle–write hazard

Table 6–14 shows the instruction hazards for **INTDP** instructions executing on the .L unit.

Table 6–14. INTDP .L-Unit Instruction Hazards

	Instruction Execution					
Cycle	1	2	3	4	5	6
INTDP	R			W	W	
Instruction Type	**Subsequent Same-Unit Instruction Executable**					
Single-cycle		✓	✓	Xw	Xw	✓
4-cycle		Xw	✓	✓	✓	✓
INTDP		Xw	✓	✓	✓	✓
ADDDP/SUBDP		✓	✓	✓	✓	✓
Instruction Type	**Same Side, Different Unit, Both Using Cross Path Executable**					
Single-cycle		✓	✓	✓	✓	✓
DP compare		✓	✓	✓	✓	✓
2-cycle DP		✓	✓	✓	✓	✓
4-cycle		✓	✓	✓	✓	✓
Load		✓	✓	✓	✓	✓
Store		✓	✓	✓	✓	✓
Branch		✓	✓	✓	✓	✓
16 × 16 multiply		✓	✓	✓	✓	✓
MPYI		✓	✓	✓	✓	✓
MPYID		✓	✓	✓	✓	✓
MPYDP		✓	✓	✓	✓	✓

Legend: (shaded) E1 phase of the single-cyle instruction
R Sources read for the instruction
W Destinations written for the instruction
✓ Next instruction can enter E1 during cycle
Xw Next instruction cannot enter E1 during cycle–write hazard

Table 6–15 shows the instruction hazards for **ADDDP/SUBDP** instructions executing on the .L unit.

Table 6–15. ADDDP/SUBDP .L-Unit Instruction Hazards

	Instruction Execution							
Cycle	1	2	3	4	5	6	7	8
ADDDP/SUBDP	R	R				W	W	
Instruction Type	**Subsequent Same-Unit Instruction Executable**							
Single-cycle		Xr	✓	✓	✓	Xw	Xw	✓
4-cycle		Xr	Xw	Xw	✓	✓	✓	✓
INTDP		Xrw	Xw	Xw	✓	✓	✓	✓
ADDDP/SUBDP		Xr	✓	✓	✓	✓	✓	✓
Instruction Type	**Same Side, Different Unit, Both Using Cross Path Executable**							
Single-cycle		Xr	✓	✓	✓	✓	✓	✓
DP compare		Xr	✓	✓	✓	✓	✓	✓
2-cycle DP		Xr	✓	✓	✓	✓	✓	✓
4-cycle		Xr	✓	✓	✓	✓	✓	✓
Load		Xr	✓	✓	✓	✓	✓	✓
Store		Xr	✓	✓	✓	✓	✓	✓
Branch		Xr	✓	✓	✓	✓	✓	✓
16 $\times$ 16 multiply		Xr	✓	✓	✓	✓	✓	✓
MPYI		Xr	✓	✓	✓	✓	✓	✓
MPYID		Xr	✓	✓	✓	✓	✓	✓
MPYDP		Xr	✓	✓	✓	✓	✓	✓

Legend:
- (shaded) E1 phase of the single-cyle instruction
- R Sources read for the instruction
- W Destinations written for the instruction
- ✓ Next instruction can enter E1 during cycle
- Xr Next instruction cannot enter E1 during cycle–read/decode hazard
- Xw Next instruction cannot enter E1 during cycle–write hazard
- Xrw Next instruction cannot enter E1 during cycle–read/decode/write hazard

6.3.4 D-Unit Instruction Hazards

Table 6–16 shows the instruction hazards for load instructions executing on the .D unit.

Table 6–16. Load .D-Unit Instruction Hazards

	Instruction Execution					
Cycle	1	2	3	4	5	6
Load	RW				W	
Instruction Type	**Subsequent Same-Unit Instruction Executable**					
Single-cycle		✓	✓	✓	✓	✓
Load		✓	✓	✓	✓	✓
Store		✓	✓	✓	✓	✓
Instruction Type	**Same Side, Different Unit, Both Using Cross Path Executable**					
16 × 16 multiply		✓	✓	✓	✓	✓
MPYI		✓	✓	✓	✓	✓
MPYID		✓	✓	✓	✓	✓
MPYDP		✓	✓	✓	✓	✓
Single-cycle		✓	✓	✓	✓	✓
DP compare		✓	✓	✓	✓	✓
2-cycle DP		✓	✓	✓	✓	✓
Branch		✓	✓	✓	✓	✓
4-cycle		✓	✓	✓	✓	✓
INTDP		✓	✓	✓	✓	✓
ADDDP/SUBDP		✓	✓	✓	✓	✓

Legend:
- (shaded) E1 phase of the single-cyle instruction
- R Sources read for the instruction
- W Destinations written for the instruction
- ✓ Next instruction can enter E1 during cycle

Table 6–17 shows the instruction hazards for store instructions executing on the .D unit.

Table 6–17. Store .D-Unit Instruction Hazards

	Instruction Execution			
Cycle	1	2	3	4
Store	RW			
Instruction Type	**Subsequent Same-Unit Instruction Executable**			
Single-cycle		✓	✓	✓
Load		✓	✓	✓
Store		✓	✓	✓
Instruction Type	**Same Side, Different Unit, Both Using Cross Path Executable**			
16 × 16 multiply		✓	✓	✓
MPYI		✓	✓	✓
MPYID		✓	✓	✓
MPYDP		✓	✓	✓
Single-cycle		✓	✓	✓
DP compare		✓	✓	✓
2-cycle DP		✓	✓	✓
Branch		✓	✓	✓
4-cycle		✓	✓	✓
INTDP		✓	✓	✓
ADDDP/SUBDP		✓	✓	✓

Legend:
- (shaded) E1 phase of the single-cyle instruction
- R Sources read for the instruction
- W Destinations written for the instruction
- ✓ Next instruction can enter E1 during cycle

Table 6–18 shows the instruction hazards for single-cycle instructions executing on the .D unit.

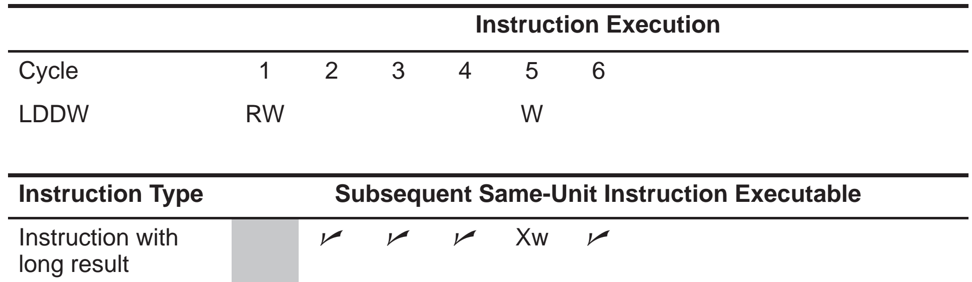
Table 6–18. Single-Cycle .D-Unit Instruction Hazards

	Instruction Execution	
Cycle	1	2
Single-cycle	RW	
Instruction Type	**Subsequent Same-Unit Instruction Executable**	
Single-cycle		✓
Load		✓
Store		✓
Instruction Type	**Same Side, Different Unit, Both Using Cross Path Executable**	
16 × 16 multiply		✓
MPYI		✓
MPYID		✓
MPYDP		✓
Single-cycle		✓
DP compare		✓
2-cycle DP		✓
Branch		✓
4-cycle		✓
INTDP		✓
ADDDP/SUBDP		✓

Legend: ▒ E1 phase of the single-cyle instruction
R Sources read for the instruction
W Destinations written for the instruction
✓ Next instruction can enter E1 during cycle

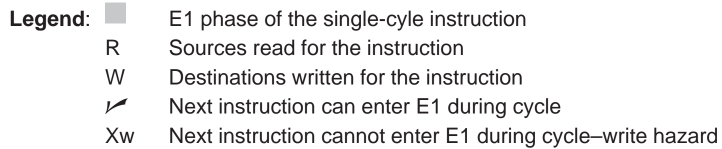
Table 6–19 shows the instruction hazards for **LDDW** instructions executing on the .D unit.

Table 6–19. LDDW Instruction With Long Write Instruction Hazards



	Instruction Execution					
Cycle	1	2	3	4	5	6
LDDW	RW				W	

Instruction Type	Subsequent Same-Unit Instruction Executable					
Instruction with long result	(shaded)	✓	✓	✓	Xw	✓



Legend:
- (shaded box) E1 phase of the single-cyle instruction
- R Sources read for the instruction
- W Destinations written for the instruction
- ✓ Next instruction can enter E1 during cycle
- Xw Next instruction cannot enter E1 during cycle–write hazard

6.3.5 Single-Cycle Instructions

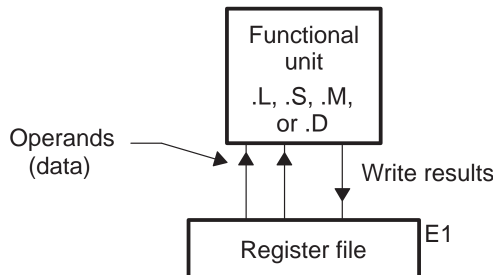
Single-cycle instructions complete execution during the E1 phase of the pipeline (see Table 6–20). Figure 6–8 shows the fetch, decode, and execute phases of the pipeline that single-cycle instructions use. Figure 6–9 is the single-cycle execution diagram. The operands are read, the operation is performed, and the results are written to a register, all during E1. Single-cycle instructions have no delay slots.

Table 6–20. Single-Cycle Execution

Pipeline Stage	E1
Read	*src1* *src2*
Written	*dst*
Unit in use	.L, .S., .M, or .D

Figure 6–8. Single-Cycle Instruction Phases

Figure 6–9. Single-Cycle Execution Block Diagram



6.3.6 16 × 16-Bit Multiply Instructions

The 16 × 16-bit multiply instructions use both the E1 and E2 phases of the pipeline to complete their operations (see Table 6–21). Figure 6–10 shows the pipeline phases the multiply instructions use. Figure 6–11 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.

Table 6–21. 16 × 16-Bit Multiply Execution

Pipeline Stage	**E1**	**E2**
Read	*src1* *src2*	
Written		*dst*
Unit in use	.M	

Figure 6–10. Multiply Instruction Phases

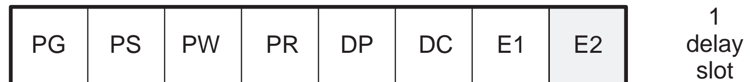
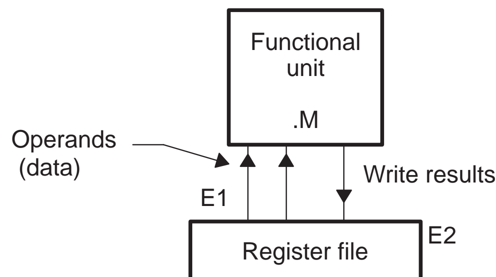


Figure 6–11. Multiply Execution Block Diagram



6.3.7 Store Instructions

Store instructions require phases E1 through E3 to complete their operations (see Table 6–22). Figure 6–12 shows the pipeline phases the store instructions use. Figure 6–13 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.

Table 6–22. Store Execution

Pipeline Stage	E1	E2	E3
Read	*baseR, offsetR src*		
Written	*baseR*		
Unit in use	.D2		

Figure 6–12. Store Instruction Phases

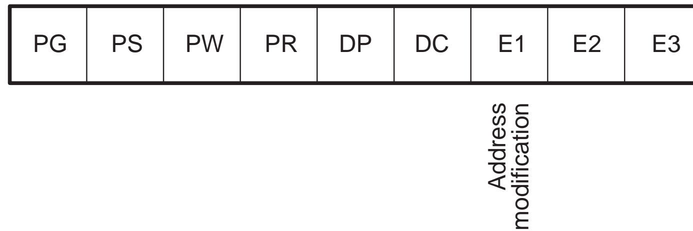
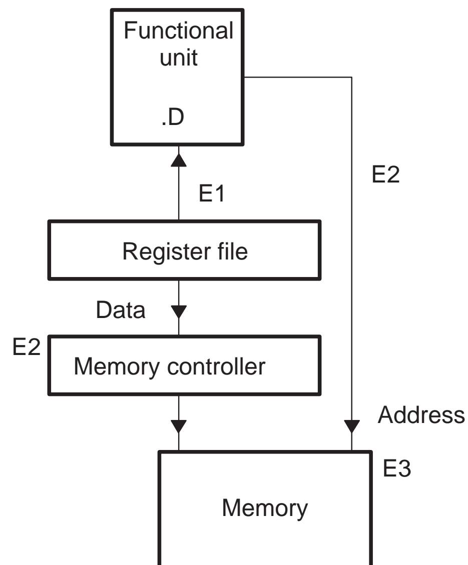


Figure 6–13. Store Execution Block Diagram



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 6.3.8.

6.3.8 Load Instructions

Data loads require five of the pipeline execute phases to complete their operations (see Table 6–23). Figure 6–14 shows the pipeline phases the load instructions use.

Table 6–23. Load Execution

Pipeline Stage	E1	E2	E3	E4	E5
Read	*baseR* *offsetR*				
Written	*baseR*				*dst*
Unit in use	.D				

Figure 6–14. Load Instruction Phases

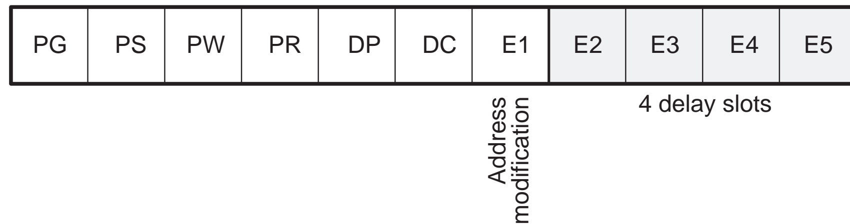
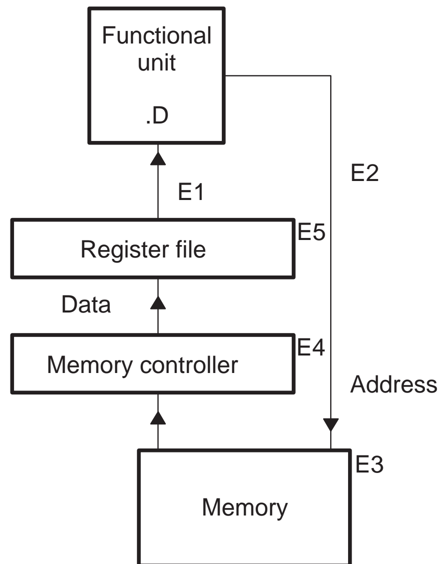


Figure 6–15 shows the operations occurring in the pipeline phases for a load. In the E1 phase, the data address pointer is modified in its register. In the E2 phase, the data address is sent to data memory. In the E3 phase, a memory read at that address is performed.

Figure 6–15. 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.

6.3.9 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 (see Table 6–24). Figure 6–16 shows the pipeline phases used by the branch instruction and branch target code. The delay slots are shaded.

Table 6–24. Branch Execution

Pipeline Stage	E1	PS	PW	PR	DP	DC	E1
Read	*src2*						
Written							
Branch Taken							✓
Unit in use	.S2						

Figure 6–16. Branch Instruction Phases

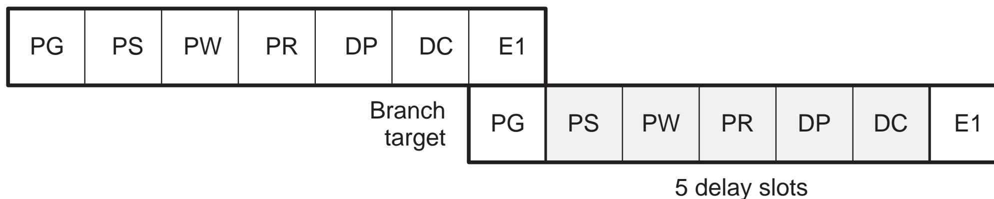
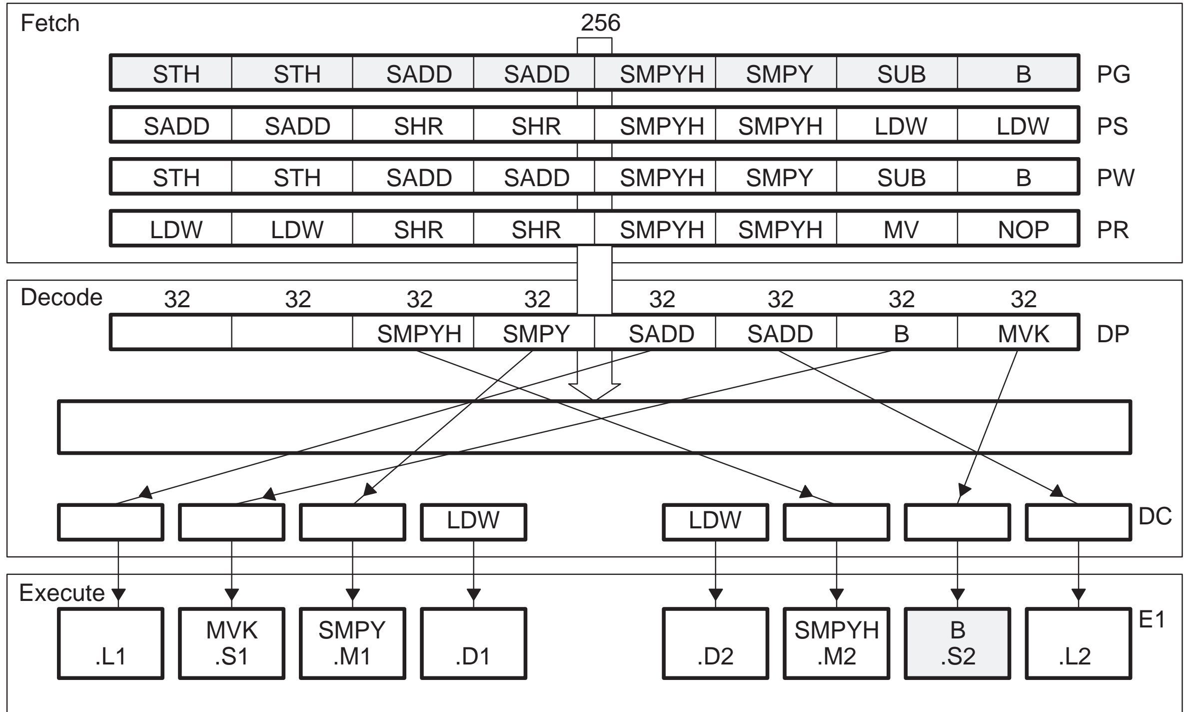


Figure 6–17 shows a branch execution block diagram. If a branch is in the E1 phase of the pipeline (in the .S2 unit in the figure), its branch target is in the fetch packet that is in PG during that same cycle (shaded in the figure). Because the branch target has to wait until it reaches the E1 phase to begin execution, the branch takes five delay slots before the branch target code executes.

Figure 6–17. Branch Execution Block Diagram



6.3.10 2-Cycle DP Instructions

Two-cycle DP instructions use the E1 and E2 phases of the pipeline to complete their operations (see Table 6–25). The following instructions are two-cycle DP instructions:

- ABSDP
- RCPDP
- RSQDP
- SPDP

The lower and upper 32 bits of the DP source are read on E1 using the src1 and src2 ports, respectively. The lower 32 bits of the DP source are written on E1 and the upper 32 bits of the DP source are written on E2. The 2-cycle DP instructions are executed on the .S units. The status is written to the FAUCR on E1. Figure 6–18 shows the pipeline phases the 2-cycle DP instructions use.

Table 6–25. 2-Cycle DP Execution

Pipeline Stage	E1	E2
Read	*src2_l* *src2_h*	
Written	*dst_l*	*dst_h*
Unit in use	.S	

Figure 6–18. 2-Cycle DP Instruction Phases

6.3.11 4-Cycle Instructions

Four-cycle instructions use the E1 through E4 phases of the pipeline to complete their operations (see Table 6–26). The following instructions are 4-cycle instructions:

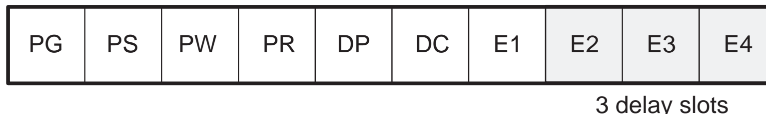
- ADDSP
- DPINT
- DPSP
- DPTRUNC
- INTSP
- MPYSP
- SPINT
- SPTRUNC
- SUBSP

The sources are read on E1 and the results are written on E4. The 4-cycle instructions are executed on the .M or .L units. The status is written to the FMCR or FADCR on E4. Figure 6–19 shows the pipeline phases the 4-cycle instructions use.

Table 6–26. 4-Cycle Execution

Pipeline Stage	E1	E2	E3	E4
Read	*src1* *src2*			
Written				*dst*
Unit in use	.L or .M			

Figure 6–19. 4-Cycle Instruction Phases



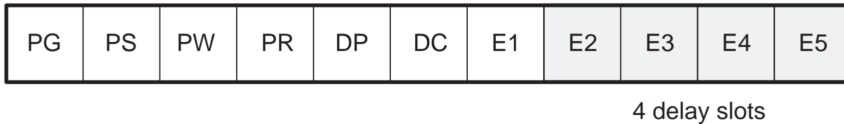
6.3.12 INTDP Instruction

The INTDP instruction uses the E1 through E5 phases of the pipeline to complete its operations (see Table 6–27). *src2* is read on E1, the lower 32 bits of the result are written on E4, and the upper 32 bits of the result are written on E5. The INTDP instruction is executed on the .L units. The status is written to the FADCR on E4. Figure 6–20 shows the pipeline phases the INTDP instructions use.

Table 6–27. INTDP Execution

Pipeline Stage	E1	E2	E3	E4	E5
Read	*src2*				
Written				*dst_l*	*dst_h*
Unit in use	.L				

Figure 6–20. INTDP Instruction Phases



6.3.13 DP Compare Instructions

The DP compare instructions use the E1 and E2 phases of the pipeline to complete their operations (see Table 6–28). The lower 32 bits of the sources are read on E1, the upper 32 bits of the sources are read on E2, and the results are written on E2. The following instructions are DP compare instructions:

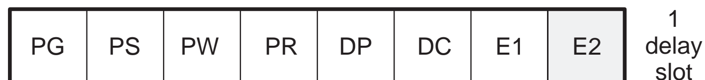
- CMPEQDP
- CMPLTDP
- CMPGTDP

The DP compare instructions are executed on the .S unit. The functional unit latency for DP compare instructions is 2. The status is written to the FAUCR on E2. Figure 6–21 shows the pipeline phases the DP compare instructions use.

Table 6–28. DP Compare Execution

Pipeline Stage	E1	E2
Read	*src1_l* *src2_l*	*src1_h* *src2_h*
Written		*dst*
Unit in use	.S	.S

Figure 6–21. DP Compare Instruction Phases



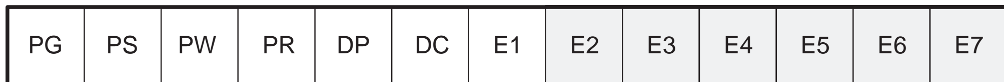
6.3.14 ADDDP/SUBDP Instructions

The ADDDP/SUBDP instructions use the E1 through E7 phases of the pipeline to complete their operations (see Table 6–29). The lower 32 bits of the result are written on E6, and the upper 32 bits of the result are written on E7. The ADDDP/SUBDP instructions are executed on the .L unit. The functional unit latency for ADDDP/SUBDP instructions is 2. The status is written to the FADCR on E6. Figure 6–22 shows the pipeline phases the ADDDP/SUBDP instructions use.

Table 6–29. ADDDP/SUBDP Execution

Pipeline Stage	E1	E2	E3	E4	E5	E6	E7
Read	*src1_l* *src2_l*	*src1_h* *src2_h*					
Written						*dst_l*	*dst_h*
Unit in use	.L	.L					

Figure 6–22. ADDDP/SUBDP Instruction Phases



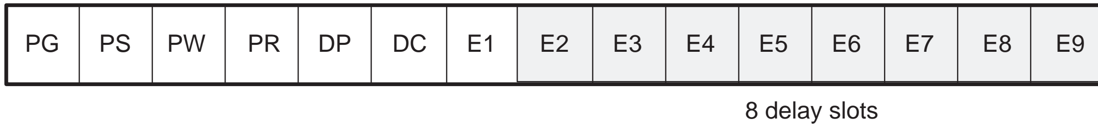
6.3.15 MPYI Instructions

The MPYI instruction uses the E1 through E9 phases of the pipeline to complete its operations (see Table 6–30). The sources are read on cycles E1 through E4 and the result is written on E9. The MPYI instruction is executed on the .M unit. The functional unit latency for the MPYI instruction is 4. Figure 6–23 shows the pipeline phases the MPYI instructions use.

Table 6–30. MPYI Execution

Pipeline Stage	E1	E2	E3	E4	E5	E6	E7	E8	E9
Read	*src1* *src2*	*src1* *src2*	*src1* *src2*	*src1* *src2*					
Written									*dst*
Unit in use	.M	.M	.M	.M					

Figure 6–23. MPYI Instruction Phases



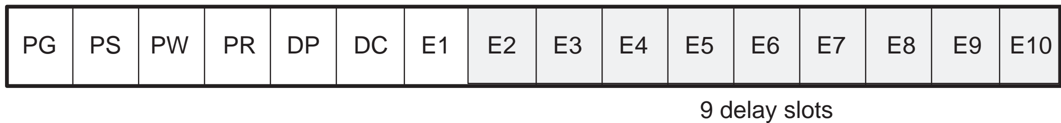
6.3.16 MPYID Instructions

The MPYID instruction uses the E1 through E10 phases of the pipeline to complete its operations (see Table 6–31). The sources are read on cycles E1 through E4, the lower 32 bits of the result are written on E9, and the upper 32 bits of the result are written on E10. The MPYID instruction is executed on the .M unit. The functional unit latency for the MPYID instruction is 4. Figure 6–24 shows the pipeline phases the MPYID instructions use.

Table 6–31. MPYID Execution

Pipeline Stage	E1	E2	E3	E4	E5	E6	E7	E8	E9	E10
Read	*src1* *src2*	*src1* *src2*	*src1* *src2*	*src1* *src2*						
Written									*dst_l*	*dst_h*
Unit in use	.M	.M	.M	.M						

Figure 6–24. MPYID Instruction Phases



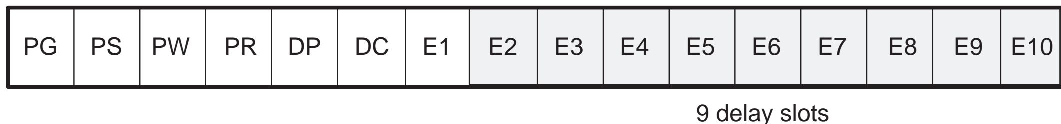
6.3.17 MPYDP Instructions

The MPYDP instruction uses the E1 through E10 phases of the pipeline to complete its operations (see Table 6–32). The lower 32 bits of *src1* are read on E1 and E2, and the upper 32 bits of *src1* are read on E3 and E4. The lower 32 bits of *src2* are read on E1 and E3, and the upper 32 bits of *src2* are read on E2 and E4. The lower 32 bits of the result are written on E9, and the upper 32 bits of the result are written on E10. The MPYDP instruction is executed on the .M unit. The functional unit latency for the MPYDP instruction is 4. The status is written to the FMCR on E9. Figure 6–25 shows the pipeline phases the MPYDP instructions use.

Table 6–32. MPYDP Execution

Pipeline Stage	E1	E2	E3	E4	E5	E6	E7	E8	E9	E10
Read	*src1_l* *src2_l*	*src1_l* *src2_h*	*src1_h* *src2_l*	*src1_h* *src2_h*						
Written									*dst_l*	*dst_h*
Unit in use	.M	.M	.M	.M						

Figure 6–25. MPYDP Instruction Phases



6.4 Performance Considerations

The ’C67x 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 6.4.2 covers the effect of including a multicycle **NOP** in an individual EP.

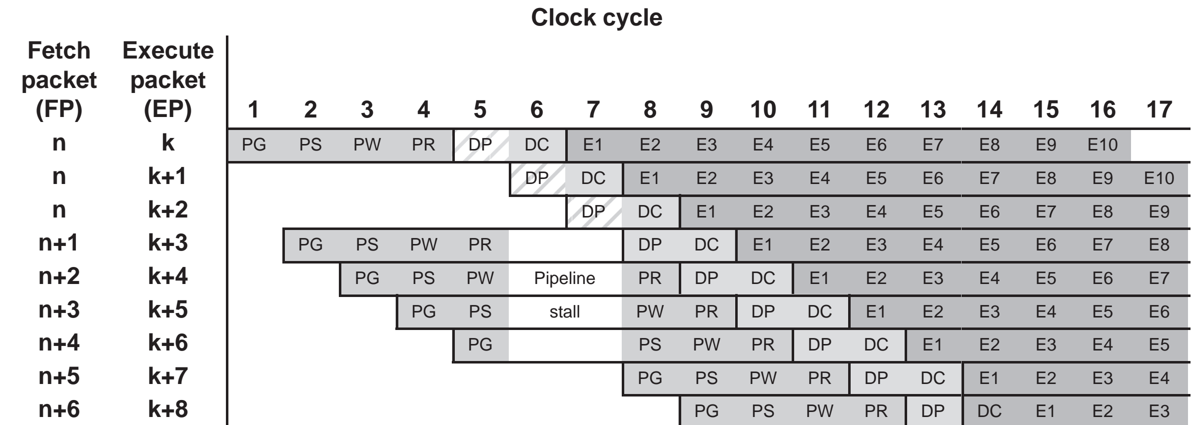
Finally, the effect of the memory system on the operation of the pipeline is considered. The access of program and data memory is discussed, along with memory stalls.

6.4.1 Pipeline Operation With Multiple Execute Packets in a Fetch Packet

Again referring to Figure 6–6 on page 6-6, pipeline operation is shown with eight instructions in every fetch packet. Figure 6–26, however, shows the pipeline operation with a fetch packet that contains multiple execute packets. Code for Figure 6–26 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 6–26. Pipeline Operation: Fetch Packets With Different Numbers of Execute Packets



Fetch packet (FP)	Execute packet (EP)	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
n	k	PG	PS	PW	PR	DP	DC	E1	E2	E3	E4	E5	E6	E7	E8	E9	E10	
n	k+1						DP	DC	E1	E2	E3	E4	E5	E6	E7	E8	E9	E10
n	k+2							DP	DC	E1	E2	E3	E4	E5	E6	E7	E8	E9
n+1	k+3		PG	PS	PW	PR			DP	DC	E1	E2	E3	E4	E5	E6	E7	E8
n+2	k+4			PG	PS	PW	Pipeline		PR	DP	DC	E1	E2	E3	E4	E5	E6	E7
n+3	k+5				PG	PS	stall		PW	PR	DP	DC	E1	E2	E3	E4	E5	E6
n+4	k+6					PG			PS	PW	PR	DP	DC	E1	E2	E3	E4	E5
n+5	k+7								PG	PS	PW	PR	DP	DC	E1	E2	E3	E4
n+6	k+8									PG	PS	PW	PR	DP	DC	E1	E2	E3

(Clock cycle 1–17)

In Figure 6–26, 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.

6.4.2 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 6–27 shows how a multicycle **NOP** can drive the execution of other instructions in the same execute packet. Figure 6–27(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 6–27(b) shows the replacement of a single-cycle **NOP with** a multicycle **NOP** (**NOP** 5) in the same execute packet. 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 6–27. Multicycle NOP in an Execute Packet

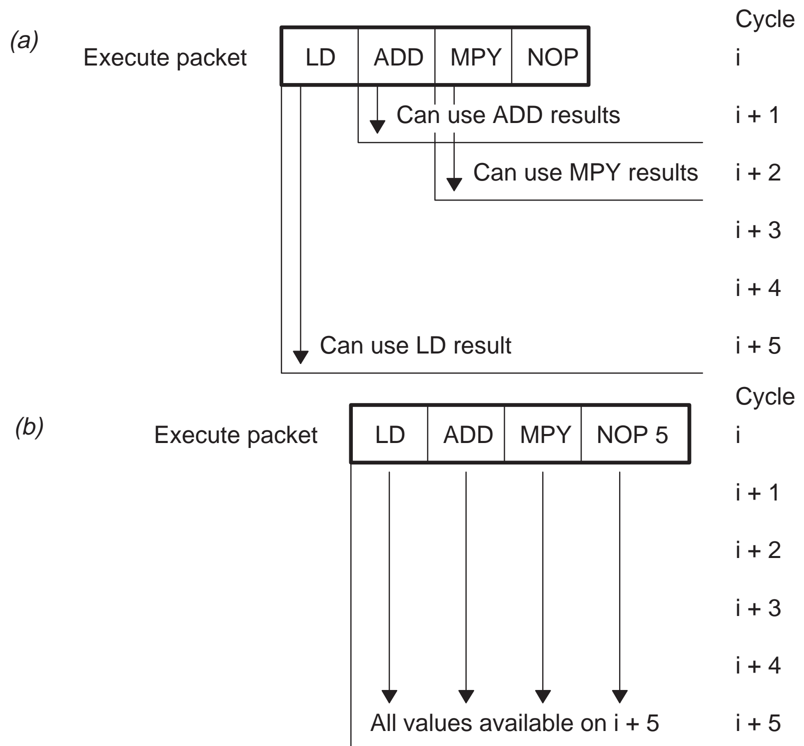
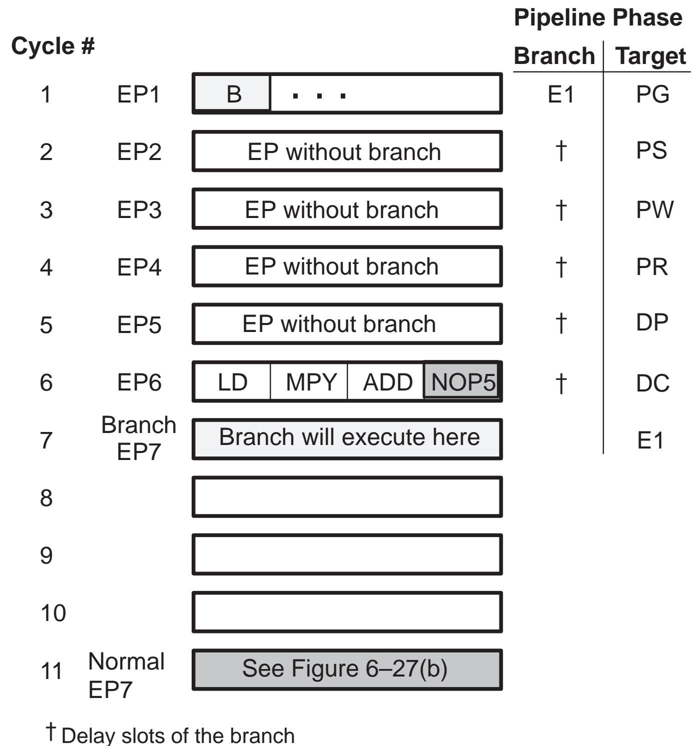


Figure 6–28 shows how a multicycle **NOP** can be affected by a branch. If the delay slots of a branch finish while a multicycle **NOP** is still dispatching **NOP**s into the pipeline, the branch overrides the multicycle **NOP** and the branch target begins execution five delay slots after the branch was issued.

*Figure 6–28. Branching and Multicycle **NOP**s*



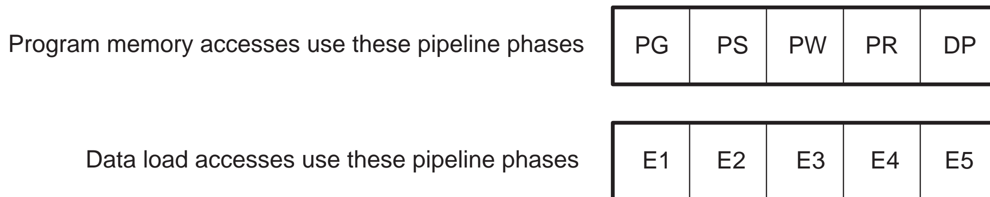
In one case, execute packet 1 (EP1) does not have a branch. The **NOP** 5 in EP6 will force the CPU to wait until cycle 11 to execute EP7.

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.

6.4.3 Memory Considerations

The ’C67x 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 ’C67x to access memory at a high speed. These phases are shown in Figure 6–29.

Figure 6–29. 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 ’C67x and perform the same types of operations (listed in Table 6–33) to accommodate those memories. Table 6–33 shows the operation of program fetches pipeline versus the operation of a data load.

Table 6–33. 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

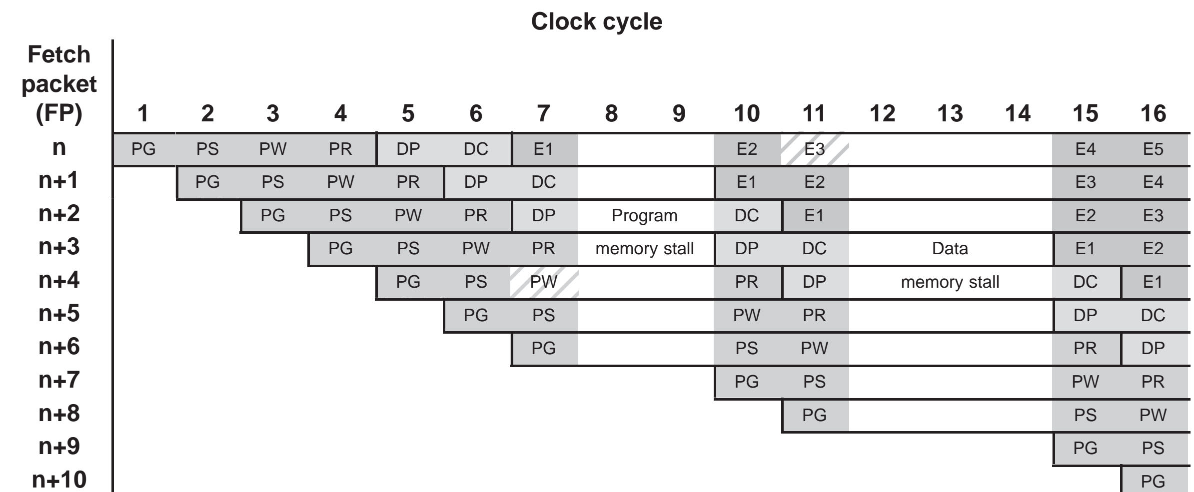
Depending on the type of memory and the time required to complete an access, the pipeline may stall to ensure proper coordination of data and instructions. This is discussed in section 6.4.3.1, *Memory Stalls.*

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 6.4.3.2, *Memory Bank Hits*.

6.4.3.1 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 6–30 illustrates this point.

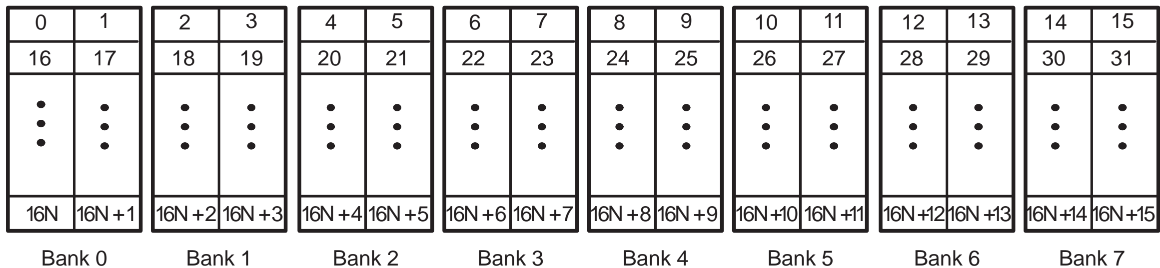
Figure 6–30. Program and Data Memory Stalls



6.4.3.2 Memory Bank Hits

Most ’C67x devices use an interleaved memory bank scheme, as shown in Figure 6–31. 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**) instruction from address 0 loads the halfword value in bytes 0 and 1, which are also in bank 0. A load word (**LDW**) instruction from address 0 loads bytes 0 through 3 in banks 0 and 1. A load double-word (**LDDW**) instruction from address 0 loads bytes 0 through 7 in banks 0 through 3.

Figure 6–31. 8-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 6–2. 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 6–34 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 *TMS320C62x/C67x Programmer’s Guide.*

Example 6–2. 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 6–34. Loads in Pipeline From Example 6–2

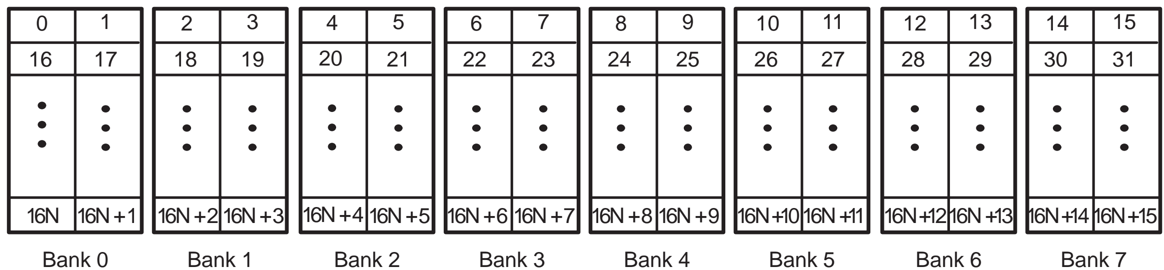
	i	*i* + 1	*i* + 2	*i* + 3	*i* + 4	*i* + 5
LDW .D1 Bank 0	E1	E2	E3	–	E4	E5
LDW .D2 Bank 0	E1	E2	–	E3	E4	E5

For devices that have more than one memory space (see Figure 6–32), 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.

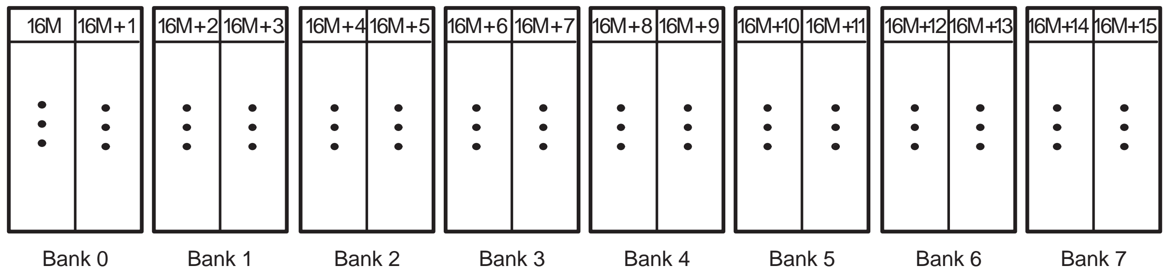
The internal memory of the ’C67x family varies from device to device. See the *TMS320C62x/C67x Peripherals Reference Guide* to determine the memory spaces in your particular device.

Figure 6–32. 8-Bank Interleaved Memory With Two Memory Spaces

Memory space 0



Memory space 1



Chapter 7

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**

7.1 Overview of Interrupts 7-2

7.2 Globally Enabling and Disabling Interrupts (Control Status Register–CSR) 7-11

7.3 Individual Interrupt Control 7-13

7.4 Interrupt Detection and Processing 7-18

7.5 Performance Considerations 7-24

7.6 Programming Considerations 7-25

7.1 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.

7.1.1 Types of Interrupts and Signals Used

There are three types of interrupts on the ’C62x/C67x CPUs. These three types are differentiated by their priorities, as shown in Table 7–1. 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 ’C62x/C67x 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 7–1. Interrupt Priorities

Priority	Interrupt Name
Highest	Reset
	NMI
	INT4
	INT5
	INT6
	INT7
	INT8
	INT9
	INT10
	INT11
	INT12
	INT13
	INT14
Lowest	INT15

7.1.1.1 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.

7.1.1.2 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.

7.1.1.3 Maskable Interrupts (INT4–INT15)

The ’C62x/C67x CPUs have 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.

7.1.1.4 Interrupt Acknowledgment (IACK and INUMx)

The IACK and INUMx signals alert hardware external to the ’C62x and C67x 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.

7.1.2 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 (ISFP) 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 7–1. 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 7–1. Interrupt Service Table

Interrupt service table (IST)

Address	Contents
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

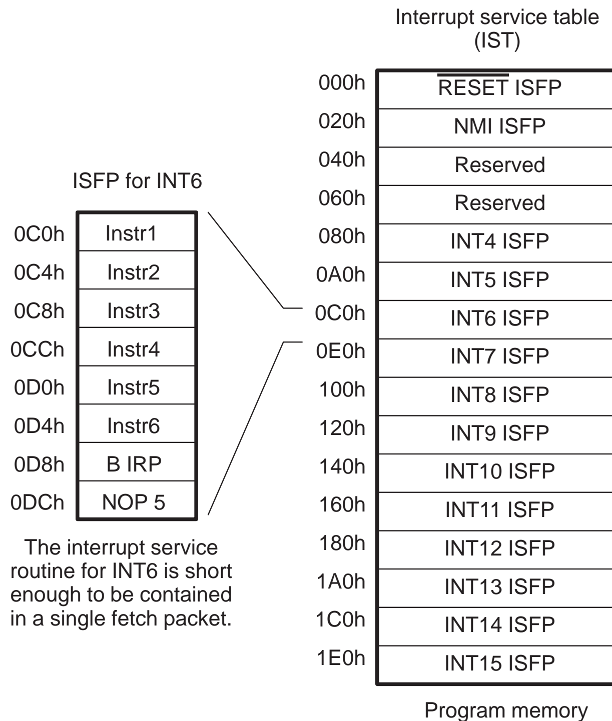
7.1.2.1 Interrupt Service Fetch Packet (ISFP)

An ISFP is a fetch packet used to service an interrupt. Figure 7–2 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:

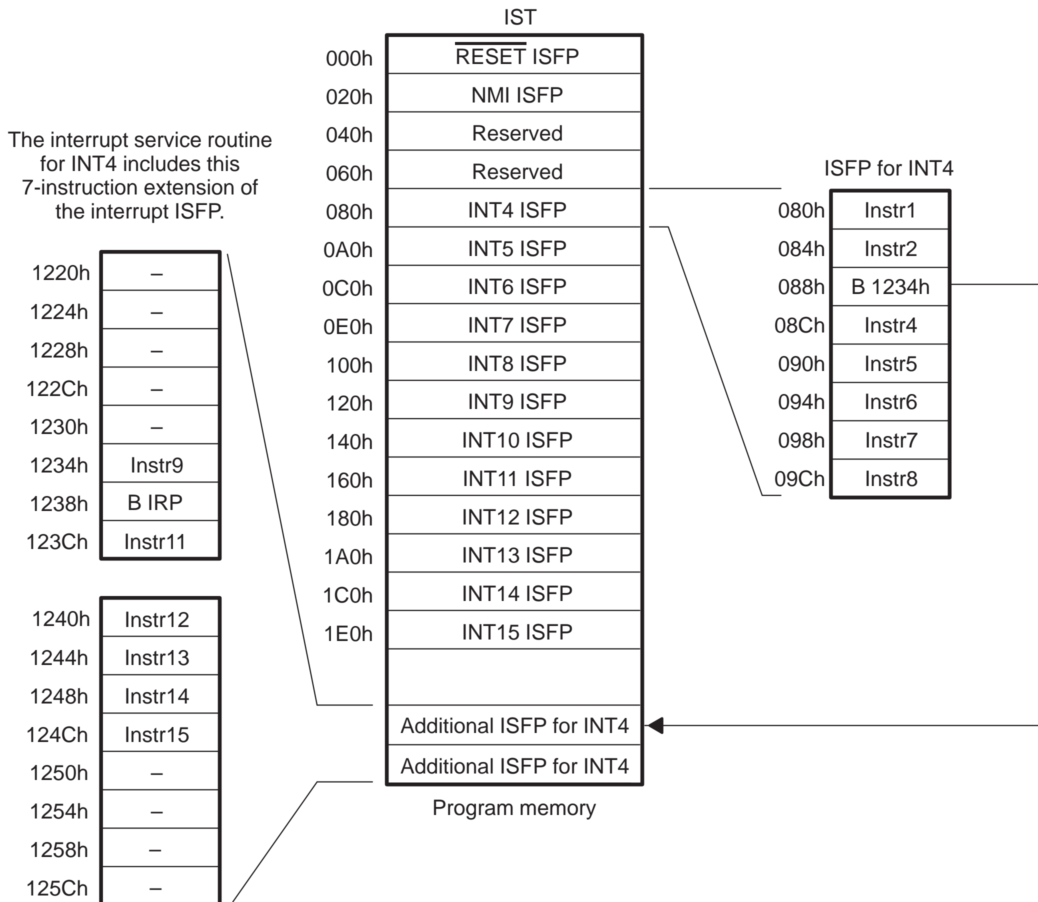
If the **NOP** 5 was not in the routine, the CPU would execute the next five execute packets that are associated with the next ISFP.

Figure 7–2. 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 7–3 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 7–3. IST With Branch to Additional Interrupt Service Code Located Outside the IST



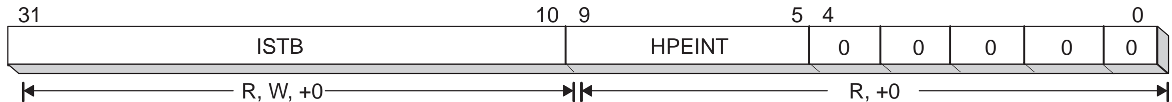
Note:

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.

7.1.2.2 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 7–4 shows the fields of the ISTP. Table 7–2 describes the fields and how they are used.

Figure 7–4. 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 7–2. Interrupt Service Table Pointer (ISTP) Field Descriptions

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 7–1) 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 7–1.

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) field of the ISTP. Example 7–1 shows the relationship of the ISTB to the table location.

Example 7–1. 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

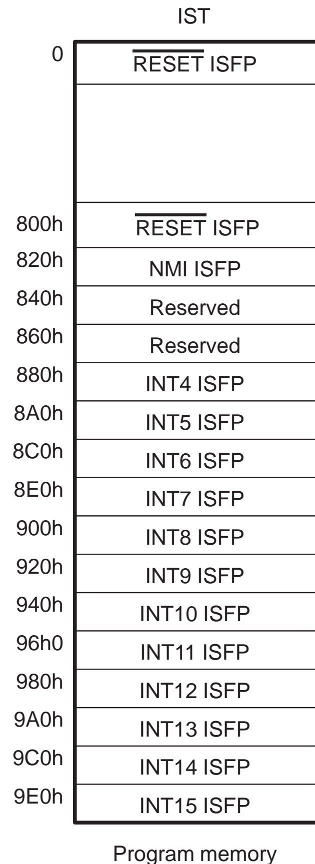
(b) How the ISTP directs the CPU to the appropriate ISFP in the relocated IST

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



7.1.3 Summary of Interrupt Control Registers

Table 7–3 lists the eight interrupt control registers on the ’C62x and ’C67x 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 register (ISR) and interrupt clear register (ICR) 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.

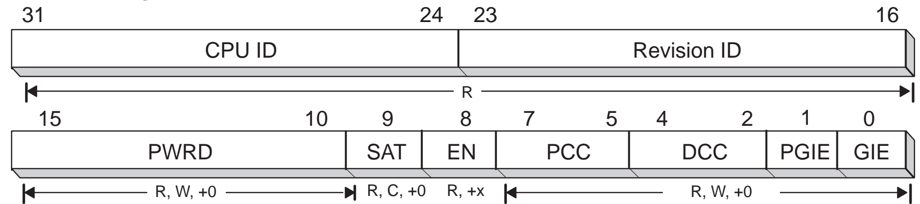
Table 7–3. Interrupt Control Registers

Abbreviation	Name	Description	Page Number
CSR	Control status register	Allows you to globally set or disable interrupts	7-11
IER	Interrupt enable register	Allows you to enable interrupts	7-13
IFR	Interrupt flag register	Shows the status of interrupts	7-14
ISR	Interrupt set register	Allows you to set flags in the IFR manually	7-14
ICR	Interrupt clear register	Allows you to clear flags in the IFR manually	7-14
ISTP	Interrupt service table pointer	Pointer to the beginning of the interrupt service table	7-8
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.	7-16
IRP	Interrupt return pointer	Contains the return address used on return from a maskable interrupt. This return is accomplished via the B IRP instruction.	7-17

7.2 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 7–5 and Table 7–4. The other fields of the registers serve other purposes and are discussed in section 2.6.2 on page 2-11.

Figure 7–5. 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 7–4. Control Status Register (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 7–2 shows how to disable maskable interrupts globally and Example 7–3 shows how to enable maskable interrupts globally.

Example 7–2. Code Sequence to Disable Maskable Interrupts Globally

```
MVC       CSR,B0    ; get CSR
AND       -2,B0,B0  ; get ready to clear GIE
MVC       B0,CSR    ; clear GIE
```

Example 7–3. Code Sequence to Enable Maskable Interrupts Globally

```
MVC       CSR,B0    ; get CSR
OR        1,B0,B0   ; get ready to set GIE
MVC       B0,CSR    ; set GIE
```

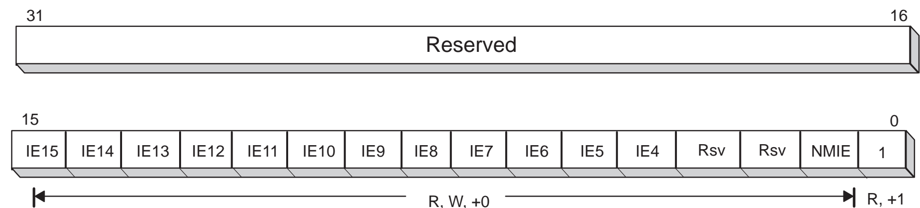
7.3 Individual Interrupt Control

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.

7.3.1 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 7–6.

Figure 7–6. 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 7–4 and Example 7–5 show code for enabling and disabling individual interrupts, respectively.

Example 7–4. Code Sequence to Enable an Individual Interrupt (INT9)

```
MVK      200h,B1 ; set bit 9
MVC      IER,B0 ; get IER
OR       B1,B0,B0 ; get ready to set IE9
MVC      B0,IER ; set bit 9 in IER
```

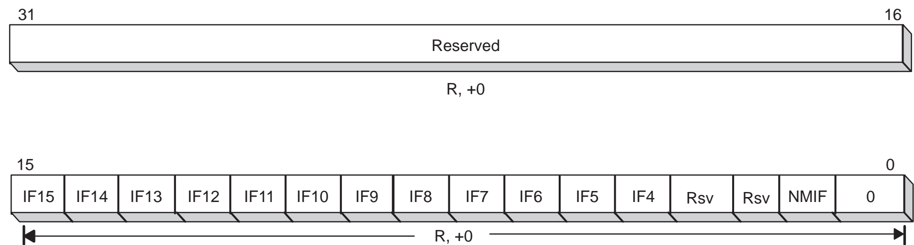
Example 7–5. Code Sequence to Disable an Individual Interrupt (INT9)

```
MVK      FDFFh,B1 ; clear bit 9
MVC      IER,B0
AND      B1,B0,B0 ; get ready to clear IE9
MVC      B0,IER ; clear bit 9 in IER
```

7.3.2 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 7–7 shows the IFR.

Figure 7–7. Interrupt Flag Register (IFR)



Legend: R = Readable by the **MVC** instruction
+0 = Cleared at reset
rsv = Reserved

The interrupt set register (ISR), shown in Figure 7–8, and the interrupt clear register (ICR), shown in Figure 7–9, 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.

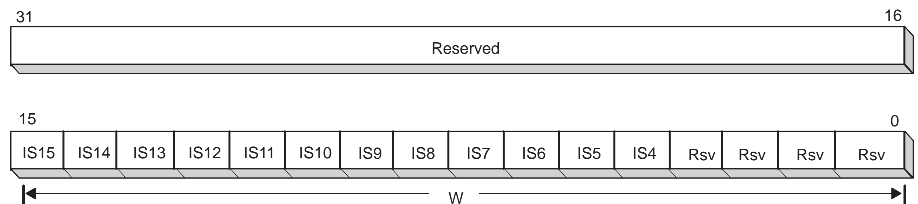
Note:

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.

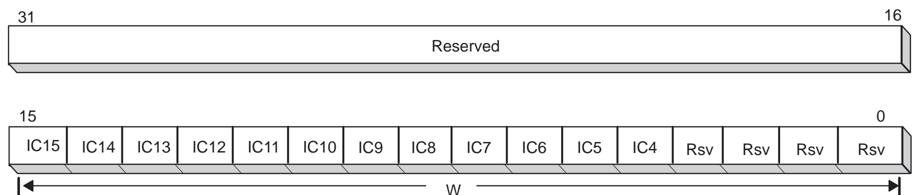
Example 7–6 and Example 7–7 show code examples to set and clear individual interrupts.

Figure 7–8. Interrupt Set Register (ISR)



Legend: W = Writeable by the **MVC** instruction
Rsv = Reserved

Figure 7–9. Interrupt Clear Register (ICR)



Legend: W = Writeable by the **MVC** instruction
Rsv = Reserved

Example 7–6. Code to Set an Individual Interrupt (INT6) and Read the Flag Register

```
MVK     40h,B3
MVC     B3,ISR
NOP
MVC     IFR,B4
```

Example 7–7. Code to Clear an Individual Interrupt (INT6) and Read the Flag Register

```
MVK     40h,B3
MVC     B3,ICR
NOP
MVC     IFR,B4
```

7.3.3 Returning From Interrupt Servicing

After 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.

7.3.3.1 CPU State After RESET

After 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

7.3.3.2 Returning From Nonmaskable Interrupts (NMI Return Pointer Register–NRP)

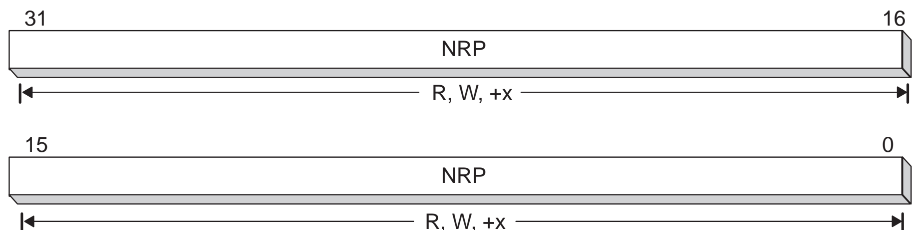
The NMI return pointer register (NRP) contains the return pointer that directs the CPU to the proper location to continue program execution after NMI processing. A branch using the address in the NRP (**B NRP**) in your interrupt service routine returns to the program flow when NMI servicing is complete. Example 7–8 shows how to return from an NMI.

Example 7–8. Code to Return From NMI

```
B          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 7–10 shows the NRP register.

Figure 7–10. NMI Return Pointer (NRP)



Legend: R = Readable by the **MVC** instruction
W = Writeable by the **MVC** instruction
+x = value undefined after reset

7.3.3.3 Returning From Maskable Interrupts (Interrupt Return Pointer Register–IRP)

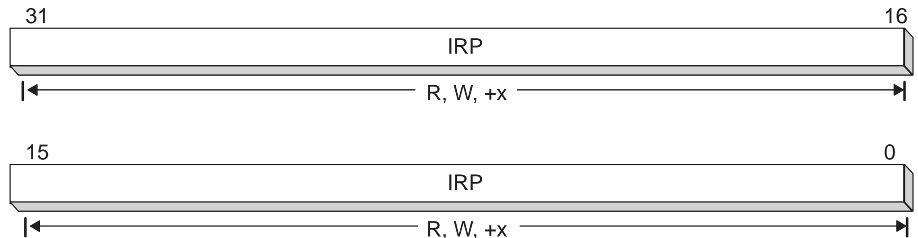
The interrupt return pointer register (IRP) contains the return pointer that directs the CPU to the proper location to continue program execution after processing a maskable interrupt. A branch using the address in the IRP (**B IRP**) in your interrupt service routine returns to the program flow when interrupt servicing is complete. Example 7–9 shows how to return from a maskable interrupt.

Example 7–9. Code to Return from a Maskable Interrupt

```
B         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 7–11 shows the IRP register.

Figure 7–11. Interrupt Return Pointer (IRP)



Legend: R = Readable by the **MVC** instruction
W = Writeable by the **MVC** instruction
+x = Value undefined after reset

7.4 Interrupt Detection and Processing

When an interrupt occurs, it sets a flag in the IFR. 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.

7.4.1 Setting the Nonreset Interrupt Flag

Figure 7–12 and Figure 7–13 show the processing of a nonreset interrupt (INTm) for the ’C62x and ’C67x, respectively. 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 7–12 and Figure 7–13, 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 7–12 and Figure 7–13 assume INTm is the highest priority pending interrupt and is enabled by GIE and NMIE as necessary. If it is not the highest priority pending interrupt, IFm remains set until either you clear it by writing a1 to bit m of the ICR, or the processing of INTm occurs.

7.4.2 Conditions for Processing a Nonreset Interrupt

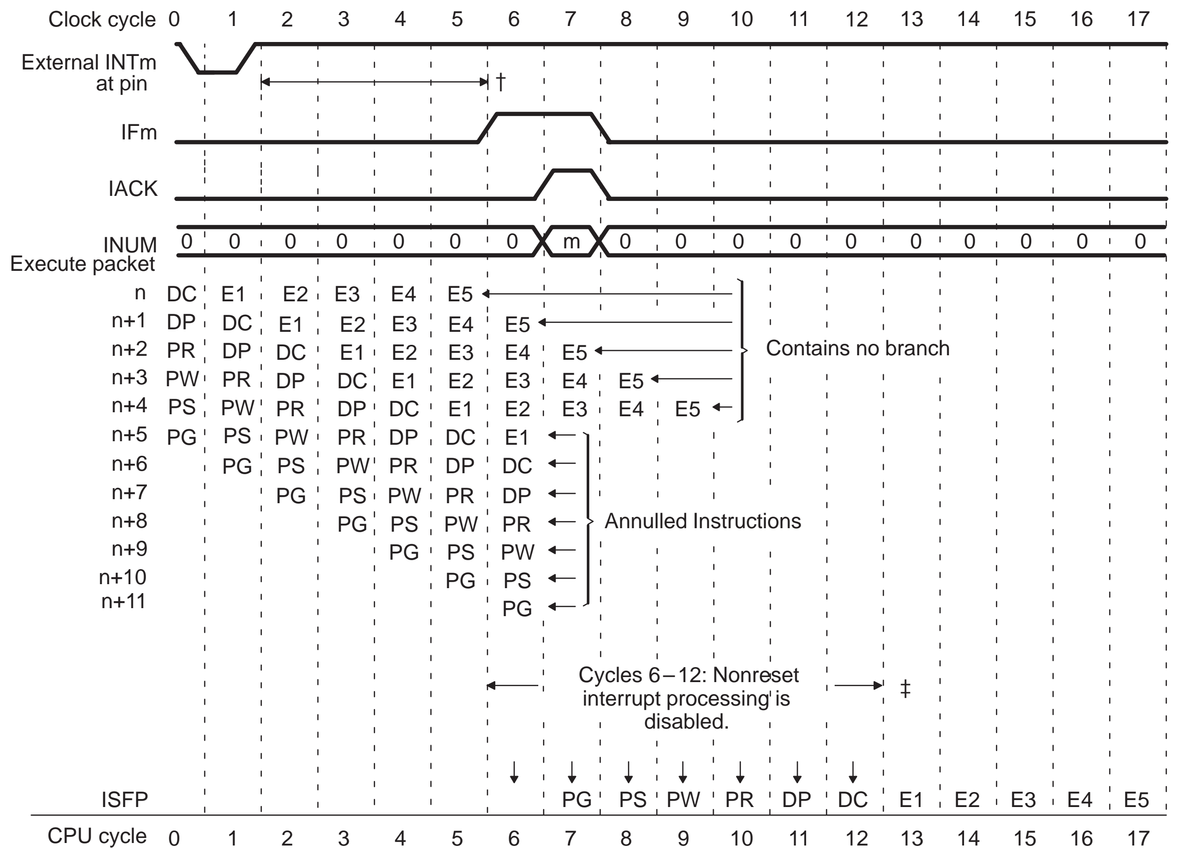
In clock cycle 4 of Figure 7–12 and Figure 7–13, a nonreset interrupt in need of processing is detected. For this interrupt to be processed, the following conditions must be valid on the same clock cycle and are evaluated every clock cycle:

- ❑ IFm is set during CPU cycle 6. (This determination is made in CPU cycle 4 by the interrupt logic.)
- ❑ There is not a 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 7–12. TMS320C62x Nonreset Interrupt Detection and Processing: Pipeline Operation



† 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.

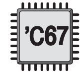
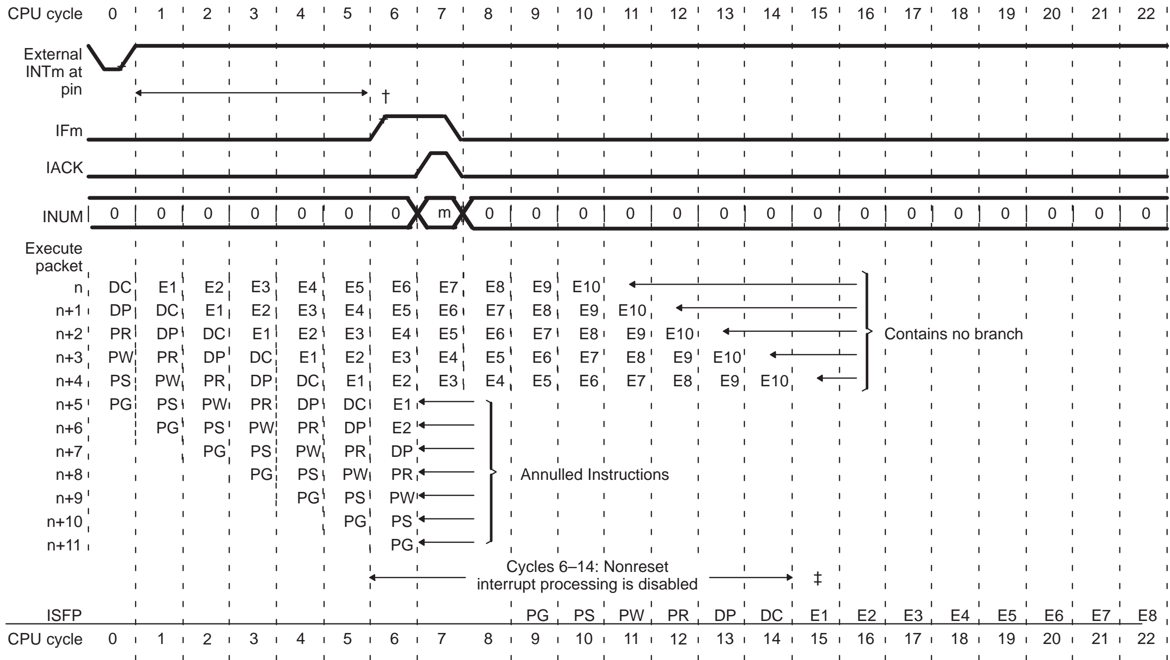


Figure 7–13. TMS320C67x Nonreset Interrupt Detection and Processing: Pipeline Operation



† 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.

7.4.3 Actions Taken During Nonreset Interrupt Processing

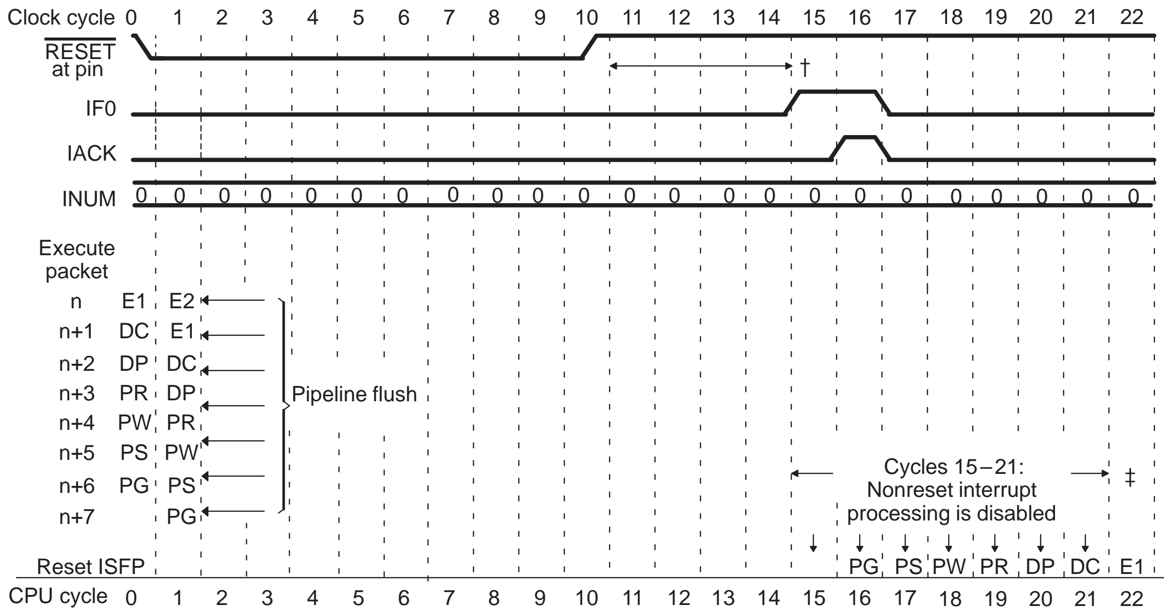
During CPU cycles 6–12 of Figure 7–12 and cycles 6–14 of Figure 7–13, the following interrupt processing actions occur:

- 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 for the ’C62x and cycle 9 for the ’C67x.
- 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 7–12 and Figure 7–13 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.

7.4.4 Setting the $\overline{\text{RESET}}$ Interrupt Flag for the TMS320C62x/C67x

$\overline{\text{RESET}}$ must be held low for a minimum of ten clock cycles. Four clock cycles after $\overline{\text{RESET}}$ 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 7–14, 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 7–14. $\overline{\text{RESET}}$ Interrupt Detection and Processing: Pipeline Operation



† 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.

7.4.5 Actions Taken During $\overline{\text{RESET}}$ Interrupt Processing

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 reset, see section 7.3.3.1 on page 7-16.

During CPU cycles 15–21 of Figure 7–14, 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 a reset is being processed.
- IF0 is cleared during cycle 17.

Note:

Code that starts running after reset must explicitly enable GIE, NMIE, and IER to allow interrupts to be processed.

7.5 Performance Considerations

The interaction of the 'C62x/C67x CPU and sources of interrupts present performance issues for you to consider when you are developing your code.

7.5.1 General Performance

- **Overhead**. Overhead for all CPU interrupts is seven cycles for the 'C62x and nine cycles for the 'C67x. You can see this in Figure 7–12 and Figure 7–13, where no new instructions are entering the E1 pipeline phase during CPU cycles 6 through 12 for the 'C62x and CPU cycles 6 through 14 for the 'C67x.
- **Latency**. Interrupt latency is 11 cycles for the 'C62x and 13 cycles for the 'C67x (21 cycles for $\overline{\text{RESET}}$). In Figure 7–13, although the interrupt is active in cycle 2, execution of interrupt service code does not begin until cycle 13 for the 'C62x and cycle 15 for the 'C67x.
- **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.

7.5.2 Pipeline Interaction

Because the serial or parallel encoding of fetch packets does not affect the DC and subsequent 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.** Nonreset interrupts are delayed if any execute packets n through n + 4 in Figure 7–12 or Figure 7–13 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.

7.6 Programming Considerations

The interaction of the ’C62x/’C67x CPUs and sources of interrupts present programming issues for you to consider when you are developing your code.

7.6.1 Single Assignment Programming

Example 7–10 shows code without single assignment and Example 7–11 shows code using the single assignment programming method.

To avoid unpredictable operation, you must employ the single assignment method in 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 contains 0 and register A0 points to a memory location containing a value of 10 before reaching the code in Example 7–10. The **ADD** instruction, which is in a delay slot of the **LDW**, sums A2 with the value in A1 (0) 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 (10), the interrupt would be processed, and the **ADD** would sum A1 (10) with A2 and place the result in A3 (equal to A2 + 10). Obviously, this situation produces incorrect results.

In Example 7–11, 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 7–10. 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 7–11. Code Using Single Assignment

```
LDW    .D1    *A0,A6
ADD    .L1    A1,A2,A3
NOP           3
MPY    .M1    A6,A4,A5 ; uses A6
```

7.6.2 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 registers saved if they are 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.

7.6.3 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 7–12.

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 7–12. Manual Interrupt Processing

```
             MVC     ISTP,B2        ; get related ISFP address
             EXTU    B2,23,27,B1    ; extract HPEINT
      [B1]   B       B2             ; branch to interrupt
   || [B1]   MVK     1,A0           ; setup ICR word
      [B1]   MVK     RET_ADR,B2     ; create return address
      [B1]   MVKH    RET_ADR,B2     ;
      [B1]   MVC     B2,IRP         ; save return address
      [B1]   SHL     A0,B1,B1       ; create ICR word
      [B1]   MVC     B1,ICR         ; clear interrupt flag
      RET_ADR:    (Post interrupt service routine Code)
```

7.6.4 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 7–13 and Example 7–14 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 7–14 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 7–13. Code Sequence to Invoke a Trap

```
[A1]    MVK          TRAP_HANDLER,B0    ; load 32-bit trap address
[A1]    MVKH         TRAP_HANDLER,B0
[A1]    B            B0                 ; branch to trap handler
[A1]    MVC          CSR,B0             ; read CSR
[A1]    AND          -2,B0,B1           ; disable interrupts: GIE = 0
[A1]    MVC          B1,CSR             ; write to CSR
[A1]    MVK          TRAP_RETURN,B1     ; load 32-bit return address
[A1]    MVKH         TRAP_RETURN,B1
TRAP_RETURN:       (post-trap code)
```

Note: A1 contains the trap condition.

Example 7–14. Code Sequence for Trap Return

```
        B            B1          ; return
        MVC          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: To cause an instruction to not complete its execution.

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 (EP): A block of instructions that execute in parallel.

external interrupt: A hardware interrupt triggered by a specific value on a pin.

F

fetch packet (FP): 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): See also *fetch packet (FP)*. 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): Sixteen 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 × 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 waiting for a memory load or store to finish.

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 manually 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

[ ] in code 3-16
|| in code 3-15
1X and 2X cross paths. *See* cross paths
1X and 2X paths. *See* crosspaths
40-bit data, conflicts 3-18
40-bit data 2-4 to 2-6
8-bank interleaved memory 6-58
8-bank interleaved memory with two memory spaces 6-59

A

ABS instruction 3-28
ABSDP instruction 4-16 to 4-17
ABSSP instruction 4-18 to 4-19
ADD instruction 3-30 to 3-33, 7-25
add instructions
- using circular addressing 3-22
- using linear addressing 3-21

ADD2 instruction 3-37
ADDAB instruction 3-22, 3-34 to 3-35
ADDAD instruction 4-20 to 4-21
ADDAH instruction 3-22, 3-34 to 3-35
ADDAW instruction 3-22, 3-34 to 3-35
ADDDP instruction 4-22 to 4-24
ADDDP instruction
- .L-unit instruction hazards 6-33
- execution 6-49
- figure of phases 6-49
- pipeline operation 6-49

ADDK instruction 3-36
address A-1
address generation for load/store 3-23
address paths 2-7
addressing mode
- circular mode 3-21
- definition A-1
- linear mode 3-21

addressing mode register (AMR) 2-8, 2-9
- field encoding, table 2-9
- figure 2-9

ADDSP instruction 4-25 to 4-27
ADDU instruction 3-30 to 3-33
AMR. *See* addressing mode register (AMR)
AND instruction 3-38 to 3-39
architecture 1-7
assembler conflict detectability for writes 3-20

B

B instruction
- using a displacement 3-40 to 3-41
- using a register 3-42 to 3-43

B IRP instruction 3-44 to 3-45, 7-6, 7-11, 7-17
B NRP instruction 3-46 to 3-47, 7-13, 7-16
base + index addressing syntax 3-23
BK field (BK1,BK2) 2-9
block size field 2-9
block size calculations 2-10
branch instruction
- .S-unit instruction hazards 6-24
- execution block diagram 5-17, 6-45
- figure of phases 5-16, 6-44
- pipeline operation 5-16, 6-44
- using a displacement 3-40 to 3-41
- using a register 3-42 to 3-43

branching
- and multicycle NOPs 5-21, 6-55
- performance considerations 7-24
- to additional interrupt service routine 7-6
- to the middle of an execute packet 3-15

C

circular addressing
  block size calculations 2-10
  block size specification 3-21
  registers that perform 2-9
clearing
  an individual interrupt 7-14
  interrupts 7-14
clock cycle 5-9, 6-11
CLR instruction 3-48 to 3-50
CMPEQ instruction 3-51 to 3-52
CMPEQDP instruction 4-28 to 4-29
CMPEQSP instruction 4-30 to 4-31
CMPGT instruction 3-53 to 3-55
CMPGTDP instruction 4-32 to 4-33
CMPGTSP instruction 4-34 to 4-35
CMPGTU instruction 3-53 to 3-55
CMPLT instruction 3-56 to 3-58
CMPLTDP instruction 4-36 to 4-37
CMPLTSP instruction 4-38 to 4-39
CMPLTU instruction 3-56 to 3-58
code, definition A-1
conditional operations 3-16
conditional registers 3-16
conflict detectability 3-20
contraints
  on crosspaths 3-17
  on floatin-point instructions 6-16 to 6-19
  on floating-point instructions 4-12 to 4-15
  on general-purpose registers 3-19 to 3-21
  on instructions using the same functional unit 3-17
  on LDDW instruction 4-14
  on loads and stores 3-18
  on long data 3-18
  on register reads 3-19
  on resources 3-17
control
  individual interrupts 7-13
  of interrupts 7-11
control register
  file extension ('C67x) 2-13
  interrupt 7-10
  list of 2-8
  register addresses for accessing 3-87
control status register (CSR) 7-10
  description 2-8, 2-11
  figure 2-11, 7-11
  interrupt control fields 7-11
CPU
  control register file 2-8
  cycle 5-9, 5-11, 6-11, 6-16
  data paths
    TMS320C62x 2-2
    TMS320C67x 2-3
  functional units 2-6
  general-purpose register files 2-4
  introduction 1-8
  load and store paths 2-7
  TMS320C62x block diagram 5-5
  TMS320C67x block diagram 6-5
CPU data paths 2-1
  relationship to register files 2-7
  TMS320C62x 2-2
  TMS320C67x 2-3
CPU ID field (CSR) 2-11
creg opcode field defined 3-16
cross paths 2-7, 3-17
CSR. *See* control status register (CSR)

D

.D functional unit
  load hazard 6-34
  store hazard 6-35
  LDDW instruction with long write hazard 6-37
  single-cycle 6-36
.D functional units 2-6
.D unit hazards
  LDDW instruction with long write instruction 6-37
  load instruction 6-34
  single-cycle instruction 6-36
  store instruction 6-35
DA1 and DA2. *See* data address paths
data address paths 2-7
data address pointer 5-15, 6-42
data format (IEEE standard) 4-6
data load accesses, versus program memory accesses 5-22, 6-56
data paths. *See* CPU data paths
data storage format, 40-bit 2-5
DC pipeline phase 5-4, 6-4

DCC field (CSR) 2-11
decode pipeline stage 5-4, 6-4
decoding instructions 5-4, 6-4
delay slots
  description 5-11, 6-16
  fixed-point instructions 3-12
  floating-point instructions 4-11
  stores 5-16, 6-43
DEN1,DEN2 fields
  FADCR 2-14 to 2-16
  FAUCR 2-16 to 2-18
  FMCR 2-18 to 2-20
detection of interrupts 7-18
digital signal processors (DSPs) 1-1
direct memory access (DMA) controller 1-9
disabling an individual interrupt 7-14
disabling maskable interrupts globally 7-12
DIV0 fields (FAUCR) 2-16 to 2-18
double-precision data format 4-6
DP compare instructions
  .S-unit instruction hazards 6-22
  execution 6-48
  figure of phases 6-48
  pipeline operation 6-48
DP pipeline phase 5-4, 5-19, 6-4, 6-53
DPINT instruction 4-40 to 4-41
DPSP instruction 4-42 to 4-44
DPTRUNC instruction 4-45 to 4-46

E

E1 phase program counter (PCE1) 2-12
E1–E5 (or E10) pipeline phases 6-5
E1–E5 pipeline phases 5-5
EMIF. *See* external memory interface (EMIF)
EN field (CSR) 2-11
enabling an individual interrupt 7-14
enabling maskable interrupts globally 7-12
execute packet
  multicycle NOPs in 5-20, 6-54
  parallel operations 3-13
  performance considerations ('C67x) 6-52
  pipeline operation 5-18
execute phases of the pipeline 5-22, 6-56
  figure 5-5, 6-5
execution notations
  fixed-point instructions 3-2
  floating-point instructions 4-2
execution table
  ADDDP/SUBDP 6-49
  INTDP 6-48
  MPYDP 6-51
  MPYI 6-50
  MPYID 6-50
EXT instruction 3-59 to 3-61
external memory interface (EMIF) 1-9
EXTU instruction 3-62 to 3-64

F

FADCR. *See* floating-point adder configuration register (FADCR)
FAUCR. *See* floating-point auxiliary configuration register (FAUCR)
fetch packet (FP) 3-13, 5-18, 6-52, 7-6
fetch phases of the pipeline 5-22
fetch pipeline phase 5-2, 6-56
fetch pipline phase
  TMS320C62x 5-3
  TMS320C67x 6-2, 6-3
fixed-point instruction set 3-1 to 3-139
flag, interrupt 7-18, 7-22
floating-point instruction constraints 4-12
floating-point instruction set 4-1 to 4-83
floating-point adder configuration register (FADCR) 2-13, 2-14 to 2-16
floating-point auxiliary configuration register (FAUCR) 2-13, 2-16 to 2-18
floating-point multiplier configuration register (FMCR) 2-13, 2-18 to 2-20
floating-point field definitions
  double-precision 4-9
  single-precision 4-8
floating-point operands
  double precision 4-6
  single precision 4-6
FMCR. *See* floating-point multiplier configuration register (FMCR)
4-cycle instructions
  .L-unit instruction hazards 6-31
  .M-unit instruction hazards 6-26
  execution 6-47

- figure of phases 6-47
- pipeline operation 6-47

Functional Unit Hazards 6-20

functional unit to instruction mapping 3-5, 4-4

functional units 2-6
- constraints on instructions 3-17
- fixed-point operations 2-6
- floating-point operations 2-6
- list of 2-6
- operations performed on 2-6

G

general-purpose register files
- cross paths 2-7
- data address paths 2-7
- description 2-4
- memory, load, and store paths 2-7

GIE bit 2-11, 7-4, 7-11, 7-19, 7-21

H

host port interface 1-9

HPEINT bit 7-8

I

IACK signal 7-4, 7-21, 7-23

ICR. *See* interrupt clear register (ICR)

IDLE instruction 3-65

IEEE standard formats 4-6

IEm bit 7-18

IER. *See* interrupt enable register (IER)

IFm bits 7-21, 7-23

IFR. *See* interrupt flag register (IFR)

individual interrupt control 7-13

INEX fields
- FADCR 2-14 to 2-16
- FAUCR 2-16 to 2-18
- FMCR 2-18 to 2-20

INFO fields
- FADCR 2-14
- FAUCR 2-16
- FMCR 2-18

instruction constraints 4-12

instruction descriptions
- fixed-point instruction set 3-24
- floating-point instruction set 4-15
  - *constraints 4-12*

instruction operation
- fixed-point, notations for 3-2
- floating-point, notations for 4-2

instruction to functional unit mapping 3-4, 4-4

instruction types
- 2-cycle DP instructions 6-46
- 4-cycle instructions 6-47
- ADDDP instructions 6-49
- ADDDP/SUBDP instructions 4-22 to 4-25, 4-77
- branch instructions 5-16, 6-44
- DP compare instructions 6-48
- execution phases 5-11, 6-13
- INTDP 4-47 to 4-49
- INTDP instructions 6-47
- load instructions 5-15, 6-42
- MPYDP 4-54 to 4-56
- MPYDP instructions 6-51
- MPYI 4-56
- MPYI instructions 6-50
- MPYID 4-57 to 4-59
- MPYID instructions 6-50
- multiply instructions 5-12, 6-39
- operation phases 6-7
- pipeline execution 5-11, 6-13
- single-cycle 5-12, 6-38
- store instructions 5-13, 6-40
- SUBDP instructions 6-49

INT4–INT15 interrupt signals 7-4

INTDP instruction 4-47 to 4-48
- execution 6-48
- figure of phases 6-48
- .L-unit instruction hazards 6-32
- pipeline operation 6-47

INTDPU instruction 4-47 to 4-48

interleaved memory bank scheme 5-24, 6-58
- 4-bank memory 5-24, 5-25
- 8-bank memory
  - *single memory space 6-58*
  - *with two memory spaces 6-59*

interrupt clear register (ICR) 2-8, 7-10, 7-14
- figure 7-15
- writing to 7-14

interrupt control 7-11
- individual 7-13

interrupt control registers 7-10

interrupt detection and processing 7-18 to 7-23
- actions taken during nonreset 7-21
- actions taken during RESET 7-23
- figure 7-22

interrupt enable register (IER) 2-8, 7-4, 7-10, 7-13
- polling 7-26

interrupt flag, setting 7-18, 7-22

interrupt flag register (IFR)
- description 2-8, 7-10
- figure 7-14
- maskable interrupts 7-4
- overview 7-2
- polling 7-26
- reading from 7-15
- writing to 7-14

interrupt performance
- frequency 7-24
- latency 7-24
- overhead 7-24

interrupt pipeline interaction
- branching 7-24
- code parallelism 7-24
- memory stalls 7-24
- multicycle NOPs 7-24

interrupt return pointer (IRP) 2-8, 7-10, 7-17, 7-26

interrupt service fetch packet (ISFP) 7-6

interrupt service table (IST)
- figure 7-5
- relocation of 7-9

interrupt service table pointer (ISTP) 2-8, 7-21, 7-23, 7-26
- description 7-10
- description of fields 7-8
- figure 7-8
- overview 7-8

interrupt set register (ISR) 2-8, 7-10, 7-14
- figure 7-15

interrupts
- branching 7-21, 7-23
- clearing 7-14
- control 7-13 to 7-17
- detection 7-18 to 7-23
- globally disabling 7-11 to 7-12
- globally enabling 7-11 to 7-12
- list of control registers 7-10
- nesting 7-26
- overview 7-2 to 7-10
- performance considerations 7-24
- priorities 7-3
- processing 7-18 to 7-23
- programming considerations 7-25 to 7-28
- setting 7-14
- signals used 7-2
- traps 7-27
- types of 7-2

INTSP instruction 4-49 to 4-50

INTSPU instruction 4-49 to 4-50

INUM3–INUM0 signals 7-4, 7-21, 7-23

INVAL fields
- FADCR 2-14 to 2-16
- FAUCR 2-16 to 2-18
- FMCR 2-18 to 2-20

invoking a trap 7-27

IRP. *See* interrupt return pointer (IRP)

ISFP. *See* interrupt service fetch packet (ISFP)

ISR. *See* interrupt set register (ISR)

IST. *See* interrupt service table (IST)

ISTB field 7-8, 7-9

ISTP. *See* interrupt service table pointer (ISTP)

L

.L functional units 2-6, 2-13

.L unit hazards
- ADDDP instruction 6-33
- 4-cycle .L-unit instruction hazards 6-31
- INTDP instruction 6-32
- single-cycle instruction 6-30
- SUBDP instruction 6-33

latency
- fixed-point instructions 3-12
- floating-point instructions 4-11

LDB instruction
- 15-bit constant offset 3-71 to 3-73
- 5-bit unsigned constant offset or register offset 3-66 to 3-70
- using circular addressing 3-21

LDBU instruction
- 15-bit constant offset 3-71 to 3-73
- 5-bit unsigned constant offset or register offset 3-66 to 3-70

LDDW instruction 4-51 to 4-53
- instruction with long write instruction hazards 6-37

LDH instruction
- 15-bit constant offset 3-71 to 3-73
- 5-bit unsigned constant offset or register offset 3-66 to 3-70

using circular addressing 3-21

LDHU instruction
- 15-bit constant offset 3-71 to 3-73
- 5-bit unsigned constant offset or register offset 3-66 to 3-70

LDW instruction 7-25
- 15-bit constant offset 3-71 to 3-73
- 5-bit unsigned constant offset or register offset 3-66 to 3-70
- using circular addressing 3-21

linear addressing mode 3-21

LMBD instruction 3-74 to 3-75

load, paths 2-7

load address generation, syntax 3-23

load and store paths, CPU 2-7

load from memory banks, example 5-24, 6-58

load instructions
- conflicts 3-18
- .D-unit instruction hazards 6-34
- execution block diagram 5-15, 6-43
- figure of phases 6-42
- phases 5-15
- pipeline operation 5-15, 6-42
- syntax for indirect addressing 3-23
- types 5-15
- using circular addressing 3-21
- using linear addressing 3-21

load or store to the same memory location, rules 5-14, 6-41

load paths 2-7

loads, and memory banks 5-24, 6-58

long (40-bit) data 3-18

long (40-bit) data, register pairs 2-4 to 2-6

M

.M functional units 2-6, 2-13

.M unit hazards
- 4-cycle instruction 6-26
- MPYDP instruction 6-29
- MPYI instruction 6-27
- MPYID instruction hazards 6-28
- multiply instruction 6-25

mapping
- functional unit to instruction 3-5, 4-4
- instruction to functional unit 3-4, 4-4

maskable interrupt
- description 7-4
- return from 7-17

memory
- considerations 5-22
- internal 1-8
- paths 2-7
- pipeline phases used during access 5-22, 6-56
- stalls 5-23, 6-57

memory bank hits 5-24, 6-58

memory paths 2-7

memory stalls 5-23, 6-57

million instructions per second (MIPS) 1-4

MPY instruction 3-76 to 3-78

MPYDP instruction 4-54 to 4-55
- .M-unit instruction hazards 6-29
- execution 6-51
- figure of phases 6-51
- pipeline operation 6-51

MPYH instruction 3-79 to 3-80

MPYHL instruction 3-81 to 3-82

MPYHLU instruction 3-81 to 3-82

MPYHSLU instruction 3-81 to 3-82

MPYHSU instruction 3-79 to 3-80

MPYHU instruction 3-79 to 3-80

MPYHULS instruction 3-81 to 3-82

MPYHUS instruction 3-79 to 3-80

MPYI instruction 4-56
- .M-unit instruction hazards 6-27
- execution 6-50
- figure of phases 6-50
- pipeline operation 6-50

MPYID instruction 4-57 to 4-58
- .M-unit instruction hazards 6-28
- execution 6-50
- figure of phases 6-51
- pipeline operation 6-50

MPYLH instruction 3-83 to 3-84

MPYLHU instruction 3-83 to 3-84

MPYLSHU instruction 3-83 to 3-84

MPYLUHS instruction 3-83 to 3-84

MPYSP instruction 4-59 to 4-60

MPYSU instruction 3-76 to 3-78

MPYU instruction 3-76 to 3-78

MPYUS instruction 3-76 to 3-78

multicycle NOPs 5-20, 6-54
- in execute packets 5-20, 6-54

multiply execution, execution block diagram 5-13
multiply instructions
- .M-unit instruction hazards 6-25
- execution 6-39
- execution block diagram 6-39
- figure of phases 5-12, 6-39
- pipeline operation 5-12, 6-39

MV instruction 3-85
MVC instruction 2-8, 3-86 to 3-88, 7-18
- writing to IFR or ICR 7-14

MVK instruction 3-89 to 3-90
MVKH instruction 3-91 to 3-92
MVKLH instruction 3-91 to 3-92

N

NEG instruction 3-93
nesting interrupts 7-26
NMI. *See* nonmaskable interrupt (NMI)
NMIE bit 7-4, 7-13, 7-19
nonmaskable interrupt (NMI) 7-3, 7-21, 7-26
- return from 7-16

nonmaskable interrupt return pointer (NRP) 2-8, 7-10, 7-16
- figure 7-16

NOP instruction 3-94 to 3-95, 5-4, 6-4, 7-6
NORM instruction 3-96 to 3-97
NOT instruction 3-98
notations
- for fixed-point instructions 3-2 to 3-3
- for floating-point instructions 4-2

NRP. *See* nonmaskable interrupt return pointer (NRP)

O

opcode map 3-9
- figure 3-10 to 3-11
- symbols and meanings 3-9

operands, examples 3-25
OR instruction 3-99 to 3-100
overview, TMS320 family 1-2

P

p-bit 3-13
parallel code, example 3-15
parallel fetch packets 3-14
parallel operations 3-13
partially serial fetch packets 3-15
PCC field (CSR) 2-11
PCE1. *See* program counter (PCE1)
performance considerations, pipeline 5-18, 6-52
peripherals 1-9
PG pipeline phase 5-2, 6-2
PGIE bit 2-11, 7-11, 7-21
pipeline
- decode stage 5-2, 5-4, 6-2, 6-4
- execute stage 5-2, 5-5, 6-2, 6-5
- execution 5-11, 6-13
- factors that provide programming flexibility 5-1, 6-1
- fetch stage 5-2, 6-2
- operation overview 5-2, 6-2
- performance considerations 5-18, 6-52
- phases 5-2, 5-6, 6-2, 6-6
- stages 5-2, 6-2

pipeline execution 5-11, 6-13
pipeline operation
- 2-cycle DP instructions 6-46
- 4-cycle instructions 6-47
- ADDDP instructions 6-49
- branch instructions 5-16, 6-44
- description 5-6 to 5-10, 6-6 to 6-12
- DP compare instructions 6-48
- fetch packets with different numbers of execute packets 5-19, 6-53
- INTDP instructions 6-47
- load instructions 5-15, 6-42
- MPYDP instructions 6-51
- MPYI instructions 6-50
- MPYID instructions 6-50
- multiple execute packets in a fetch packet 5-18, 6-52
- multiply instructions 5-12, 6-39
- one execute packet per fetch packet 5-6, 6-6
- single-cycle instructions 5-12, 6-38
- store instructions 5-13, 6-40
- SUBDP instructions 6-49

pipeline phases
- functional block diagram 5-8, 6-10

operations occurring during 5-7
used during memory accesses 5-22, 6-56
PR pipeline phase 5-2, 6-2
program access ready wait. *See* PW pipeline phase
program address generate. *See* PG pipeline phase
program address send. *See* PS pipeline phase
program counter (PCE1) 2-8, 2-12, 3-40
figure 2-12
program fetch counter (PFC) 3-40
program fetch packet receive. *See* PR pipeline phase
program memory accesses, versus data load accesses 5-22, 6-56
PS pipeline phase 5-2, 6-2
push, definition A-3
PW pipeline phase 5-2, 6-2
PWRD field (CSR) 2-11

R

RCPDP instruction 4-61 to 4-62
RCPSP instruction 4-63 to 4-64
register files
cross paths 2-7
data address paths 2-7
general-purpose 2-4
memory, load, and store paths 2-7
relationship to data paths 2-7
register storage scheme, 40-bit data, figure 2-5
registers
AMR. *See* addressing mode register (AMR)
CSR. *See* control status register (CSR)
FADCR. *See* floating-point adder configuration register (FADCR)
FAUCR. *See* floating-point auxiliary configuration register (FAUCR)
FMCR. *See* floating-point multiplier configuration register (FMCR)
ICR. *See* interrupt clear register (ICR)
IER. *See* interrupt enable register (IER)
IFR. *See* interrupt flag register (IFR)
IRP. *See* interrupt return pointer (IRP)
ISR. *See* interrupt set register (ISR)
ISTP. *See* interrupt service table pointer (ISTP)
NRP. *See* nonmaskable interrupt return pointer (NRP)
PCE1. *See* program counter (PCE1)
read constraints 3-19
write constraints 3-19
relocation of the interrupt service table (IST) 7-9
reset interrupt 7-3
$\overline{\text{RESET}}$ signal
as an interrupt 7-3
CPU state after 7-16
resource constraints 3-17
using the same functional unit 3-17
returning from a trap 7-27
returning from interrupt servicing 7-16
returning from maskable interrupt 7-17
returning from NMI 7-16
RSQRDP instruction 4-65 to 4-67
RSQRSP instruction 4-68 to 4-70

S

.S functional units 2-6
.S unit hazards
2-cycle DP instruction 6-23
branch instruction 6-24
DP compare instruction 6-22
single-cycle instruction 6-21
SADD instruction 3-101 to 3-103
SAT field (CSR) 2-11
SAT instruction 3-104 to 3-105
serial fetch packets 3-14
serial ports 1-9
SET instruction 3-106 to 3-108
setting an individual interrupt, example 7-15
setting interrupts 7-14
setting the interrupt flag 7-18, 7-22
SHL instruction 3-109 to 3-110
SHR instruction 3-111 to 3-112
SHRU instruction 3-113 to 3-114
single-cycle instructions
.L-unit instruction hazards 6-30
.S-unit instruction hazards 6-21
.D-unit instruction hazards 6-36
execution 6-38
execution block diagram 6-38
figure of phases 6-38
pipeline operation 6-38
single-cycle instructions
execution block diagram 5-12
figure of phases 5-12

pipeline operation 5-12
SMPY instruction 3-115 to 3-117
SMPYH instruction 3-115 to 3-117
SMPYHL instruction 3-115 to 3-117
SMPYLH instruction 3-115 to 3-117
SPDP instruction 4-71 to 4-72
SPINT instruction 4-73 to 4-74
SPTRUNC instruction 4-75 to 4-76
SSHL instruction 3-118 to 3-119
SSUB instruction 3-120
STB instruction
15-bit offset 3-126 to 3-127
register offset or 5-bit unsigned constant offset 3-122 to 3-125
using circular addressing 3-21
STH instruction
15-bit offset 3-126 to 3-127
register offset or 5-bit unsigned constant offset 3-122 to 3-125
using circular addressing 3-21
store address generation, syntax 3-23
store instructions
conflicts 3-18
.D-unit instruction hazards 6-35
execution block diagram 5-14, 6-41
figure of phases 5-13, 6-40
pipeline operation 5-13, 6-40
syntax for indirect addressing 3-23
using circular addressing 3-21
using linear addressing 3-21
store or load to the same memory location, rules 5-14, 6-41
store paths 2-7
STW instruction
15-bit offset 3-126 to 3-127
register offset or 5-bit unsigned constant offset 3-122 to 3-125
using circular addressing 3-21
SUB instruction 3-128 to 3-130
SUB2 instruction 3-135
SUBAB instruction 3-22, 3-131 to 3-132
SUBAH instruction 3-22, 3-131 to 3-132
SUBAW instruction 3-22, 3-131 to 3-132
SUBC instruction 3-133 to 3-134
SUBDP instruction 4-77 to 4-79
.L-unit instruction hazards 6-33
execution 6-49
figure of phases 6-49
pipeline operation 6-49
SUBSP instruction 4-80 to 4-82
subtract instructions
using circular addressing 3-22
using linear addressing 3-21
SUBU instruction 3-128 to 3-130

T

timers 1-9
TMS320 family
advantages 1-2
applications 1-2 to 1-3
history 1-2
overview 1-2
TMS320C62x devices
architecture 1-7 to 1-10
block diagram 1-7
features 1-5
options 1-5 to 1-6
performance 1-4
TMS320C67x devices
architecture 1-7 to 1-10
block diagram 1-7
features 1-5
options 1-5 to 1-6
performance 1-4
traps 7-27
2-cycle DP instructions
.S-unit instruction hazards 6-23
execution 6-46
figure of phases 6-46
pipeline operation 6-46

V

VelociTI architecture 1-1
VLIW (very long instruction word) architecture 1-1

X

XOR instruction 3-136 to 3-137

Z

ZERO instruction 3-138