TMS320C6000
CPU and Instruction Set
Reference Guide

Literature Number: SPRU189F
October 2000

PRINTED WITH SOY INK ™

TEXAS INSTRUMENTS

Printed on Recycled Paper

IMPORTANT NOTICE

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

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

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

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

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

Copyright © 2000, 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, C64x™ refers to the TMS320C64x™ 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, 6, 7, and 8 to grasp the concepts of the architecture. Chapter 3, Chapter 4, and Chapter 5 contain detailed information about each instruction and are best used as reference material. However, you may want to read sections 3.1 through 3.9, sections 4.1 through 4.6, and sections 5.1 through 5.8 for general information about the instruction set and to understand the instruction descriptions. Then browse through Chapter 3, Chapter 4, and Chapter 5 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/C64x/C67x Fixed-Point Instruction Set* Chapter 4, *TMS320C67x Floating-Point Instruction Set* Chapter 5, *TMS320C64x Fixed-Point Instruction Set*
Conditional operations	Chapter 3, *TMS320C62x/C64x/C67x Fixed-Point Instruction Set* Chapter 4, *TMS320C67x Floating-Point Instruction Set* Chapter 5, *TMS320C64x Fixed-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/C64x/C67x Fixed-Point Instruction Set* Chapter 4, *TMS320C67x Floating-Point Instruction Set* Chapter 5, *TMS320C64x Fixed-Point Instruction Set* Chapter 6, *TMS320C62x/C64x Pipeline* Chapter 7, *TMS320C67x Pipeline*
General-purpose register files	Chapter 2, *CPU Data Paths and Control*
Instruction set	Chapter 3, *TMS320C62x/C64x/C67x Fixed-Point Instruction Set* Chapter 4, *TMS320C67x Floating-Point Instruction Set* Chapter 5, *TMS320C64x Fixed-Point Instruction Set*
Interrupts and control registers	Chapter 8, *Interrupts*

Parallel operations	Chapter 3, *TMS320C62x/C64x/C67x Fixed-Point Instruction Set* Chapter 4, *TMS320C67x Floating-Point Instruction Set* Chapter 5, *TMS320C64x Fixed-Point Instruction Set*
Pipeline phases and operation	Chapter 6, *TMS320C62x/C64x Pipeline* Chapter 7, *TMS320C67x Pipeline*
Reset	Chapter 8, *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/C64x/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*.

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.

TMS320C64x Technical Overview (literature number SPRU395) gives an introduction to the C64x digital signal processor, and discusses the application areas that are enhanced by the C64x VelociTI.2 extensions to the C62x/C67x architecture.

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.

TMS320C6203 Digital Signal Processor Data Sheet (literature number SPRS086) describes the features of the TMS320C6203 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.

TMS320C6711 Digital Signal Processor Data Sheet (literature number SPRS088) describes the features of the TMS320C6711 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), enhanced direct memory access (EDMA), expansion bus (XBUS), clocking and phase-locked loop (PLL), and the power-down modes.

TMS320C6000 Programmer's Guide (literature number SPRU198) describes ways to optimize C and assembly code for the TMS320C6000 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.

TMS320C6x Peripheral Support Library Programmer's Reference (literature number SPRU273) describes the contents of the 'C6x peripheral support library of functions and macros. It lists functions and macros both by header file and alphabetically, provides a complete description of each, and gives code examples to show how they are used.

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

TMS320C6x Evaluation Module Reference Guide (literature number SPRU269) provides instructions for installing and operating the 'C6x evaluation module. It also includes support software documentation, application programming interfaces, and technical reference material.

TMS320C62x Multichannel Evaluation Module User's Guide (literature number SPRU285) provides instructions for installing and operating the 'C62x multichannel evaluation module. It also includes support software documentation, application programming interfaces, and technical reference material.

TMS320C62x Multichannel Evaluation Module Technical Reference (SPRU308) provides provides technical reference information for the 'C62x multichannel evaluation module (McEVM). It includes support software documentation, application programming interface references, and hardware descriptions for the 'C62x McEVM.

TMS320C6201/6701 Evaluation Module Technical Reference (SPRU305) provides provides technical information that describes the 'C6x evaluation module functionality. It includes a description of host software utilities and a complete application programming interface reference.

TMS320C6000 DSP/BIOS User's Guide (literature number SPRU303) describes how to use DSP/BIOS tools and APIs to analyze embedded real-time DSP applications.

TMS320C6000 Code Composer Studio Tutorial (literature number SPRU301) introduces the Code Composer Studio integrated development environment and software tools for the TMS320C6000.

Trademarks

XDS510, VelociTI, and 320 Hotline On-line are trademarks of Texas Instruments. All of the digital signal processors within the TMS320 family are trademarks of Texas Instruments.

Windows and Windows NT are registered trademarks of Microsoft Corporation.

Contents

Figures

Tables

Examples

Chapter 1

Introduction

The TMS320C6000™ digital signal processor (DSP) platform is part of the TMS320™ DSP family. The TMS320C62x™ DSP generation and the TMS320C64x™ DSP generation comprise fixed-point devices in the C6000™ DSP platform, and the TMS320C67x™ DSP generation comprises floating-point devices in the C6000 DSP platform. The TMS320C62x and TMS320C64x™ DSPs are code-compatible. The TMS320C62x and TMS320C67x DSPs are code-compatible. All three 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 C6000 platform of devices 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 TMSC6000 Optimizing C 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

The TMS320™ DSP 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 (TI) 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 6000 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
- Remote access servers (RAS)
- Digital subscriber loop (DSL) systems
- Cable modems
- Multichannel telephony systems

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

- Personalized home security with face and hand/fingerprint recognition
- Advanced cruise control with global positioning systems (GPS) navigation and accident avoidance
- Remote medical diagnostics.
- Beam-forming base stations
- Virtual reality 3-D graphics
- Speech recognition
- Audio
- Radar
- Atmospheric modeling
- Finite element analysis
- Imaging (examples: fingerprint recognition, ultrasound, and MRI).

1.3 Features and Options of the TMS320C62x/C64x/C67x

The C6000 devices execute up to eight 32-bit instructions per cycle. The C62x/C67x device's core CPU consists of 32 general-purpose registers of 32-bit word length and eight functional units. The C64x core CPU consists of 64 general-purpose 32-bit registers and eight functional units. These eight functional units contain:

- Two multipliers
- Six ALUs

The C6000 generation has a complete set of optimized development tools, including an efficient C compiler, an assembly optimizer for simplified assembly-language programming and scheduling, and a Windows™ based debugger interface for visibility into source code execution characteristics. A hardware emulation board, compatible with the TI XDS510™ emulator interface, is also available. This tool complies with IEEE Standard 1149.1–1990, IEEE Standard Test Access Port and Boundary-Scan Architecture.

Features of the C6000 devices 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
- Conditional execution of all instructions
  - Reduces costly branching
  - Increases parallelism for higher sustained performance
- Efficient code execution 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:

- ❑ Hardware support for single-precision (32-bit) and double-precision (64-bit) IEEE floating-point operations
- ❑ 32 × 32-bit integer multiply with 32- or 64-bit result.

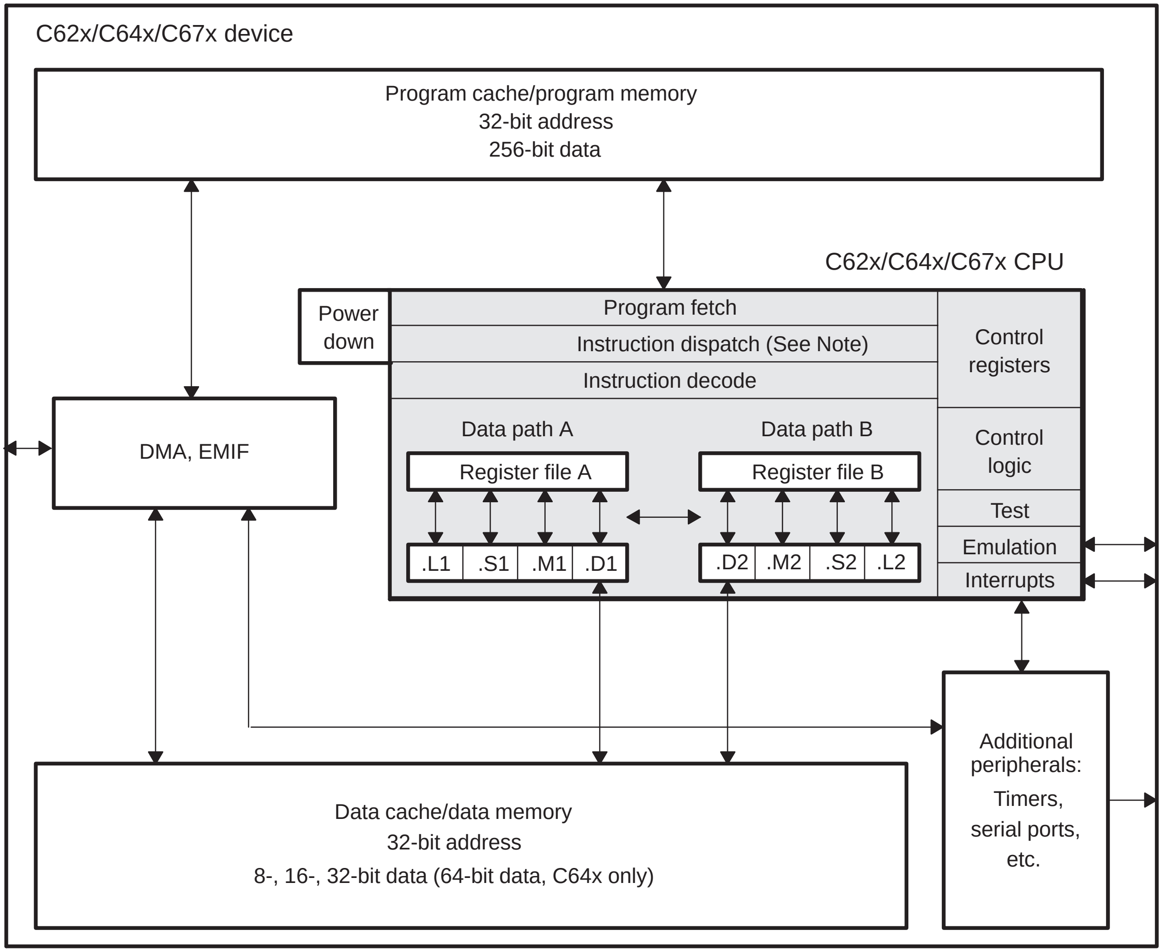
The C64x additional features include:

- ❑ Each multiplier can perform two 16 x 16-bit or four 8 x 8 bit multiplies every clock cycle.
- ❑ Quad 8-bit and dual 16-bit instruction set extensions with data flow support
- ❑ Support for non-aligned 32-bit (word) and 64-bit (double word) memory accesses
- ❑ Special communication-specific instructions have been added to address common operations in error-correcting codes.
- ❑ Bit count and rotate hardware extends support for bit-level algorithms.

1.4 TMS320C62x/C64x/C67x Architecture

Figure 1–1 is the block diagram for the TMS320C62x/C64x/C67x DSPs. The C6000 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/C64x/C67x Block Diagram



Note: The instruction dispatch unit, on the C64x only, has advanced instruction packing.

1.4.1 Central Processing Unit (CPU)

The C62x/C64x/C67x CPU, shaded in Figure 1–1, is common to all the C62x/C64x/C67x devices. The CPU contains:

❑ Program fetch unit
❑ Instruction dispatch unit, advanced instruction packing (C64 only)
❑ Instruction decode unit
❑ Two data paths, each with four functional units
❑ 32 32-bit registers, 64 32-bit registers (C64 only)
❑ 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 for the C62x/C67x and 32 32-bit general-purpose registers for the C64x. 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 6, *TMS320C62x/C64x Pipeline*, and Chapter 7, *TMS320C67x Pipeline*.

1.4.2 Internal Memory

The C62x/C64x/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 C64x has two 64-bit internal ports to access internal data memory. The C62x/C64x/C67x have a single internal port to access internal program memory, with an instruction-fetch width of 256 bits.

1.4.3 Memory and Peripheral Options

A variety of memory and peripheral options are available for the C6000 platform:

- Large on-chip RAM, up to 7M bits
- Program cache
- 2-level caches
- 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.
- DMA Controller transfers data between address ranges in the memory map without intervention by the CPU. The DMA controller has four programmable channels and a fifth auxiliary channel.
- EDMA Controller performs the same functions as the DMA controller. The EDMA has 16 programmable channels, as well as a RAM space to hold multiple configurations for future transfers.
- HPI is a parallel port through which a host processor can directly access the CPU's memory space. The host device has ease of access because it is the master of the interface. The host and the CPU can exchange information via internal or external memory. In addition, the host has direct access to memory-mapped peripherals.
- Expansion bus is a replacement for the HPI, as well as an expansion of the EMIF. The expansion provides two distinct areas of functionality (host port and I/O port) which can co-exist in a system. The host port of the expansion bus can operate in either asynchronous slave mode, similar to the HPI, or in synchronous master/slave mode. This allows the device to interface to a variety of host bus protocols. Synchronous FIFOs and asynchronous peripheral I/O devices may interface to the expansion bus.
- McBSP (multichannel buffered serial port) is based on the standard serial port interface found on the TMS320C2000 and C5000 platform devices. In addition, the port can buffer serial samples in memory automatically with the aid of the DMA/EDNA controller. It also has multichannel capability compatible with the T1, E1, SCSA, and MVIP networking standards.

- ❑ Timers in the C6000 devices are two 32-bit general-purpose timers used for these functions:
  - ■ Time events
  - ■ Count events
  - ■ Generate pulses
  - ■ Interrupt the CPU
  - ■ Send synchronization events to the DMA/EDMA controller.
- ❑ Power-down logic allows reduced clocking to reduce power consumption. Most of the operating power of CMOS logic dissipates during circuit switching from one logic state to another. By preventing some or all of the chip's logic from switching, you can realize significant power savings without losing any data or operational context.

For more information on features and options of the peripherals for the TMS320C6000, refer to the TM320C6000 *Peripherals Reference Guide* (SPRU190).

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 cross paths are described.

The components of the data path for TMS320C62x™, TMS320C67x™, and TMS320C64x™ are shown in Figure 2–1, Figure 2–2, and Figure 2–3, respectively.

These components consist of the following:

- ❑ Two general-purpose register files (A and B)
- ❑ Eight functional units (.L1, .L2, .S1, .S2, .M1, .M2, .D1, and .D2)
- ❑ Two load-from-memory data paths (LD1 and LD2)
- ❑ Two store-to-memory data paths (ST1 and ST2)
- ❑ Two data address paths (DA1 and DA2)
- ❑ Two register file data cross paths (1X and 2X).

Topic **Page**

Figure 2–1. TMS320C62x CPU Data Paths

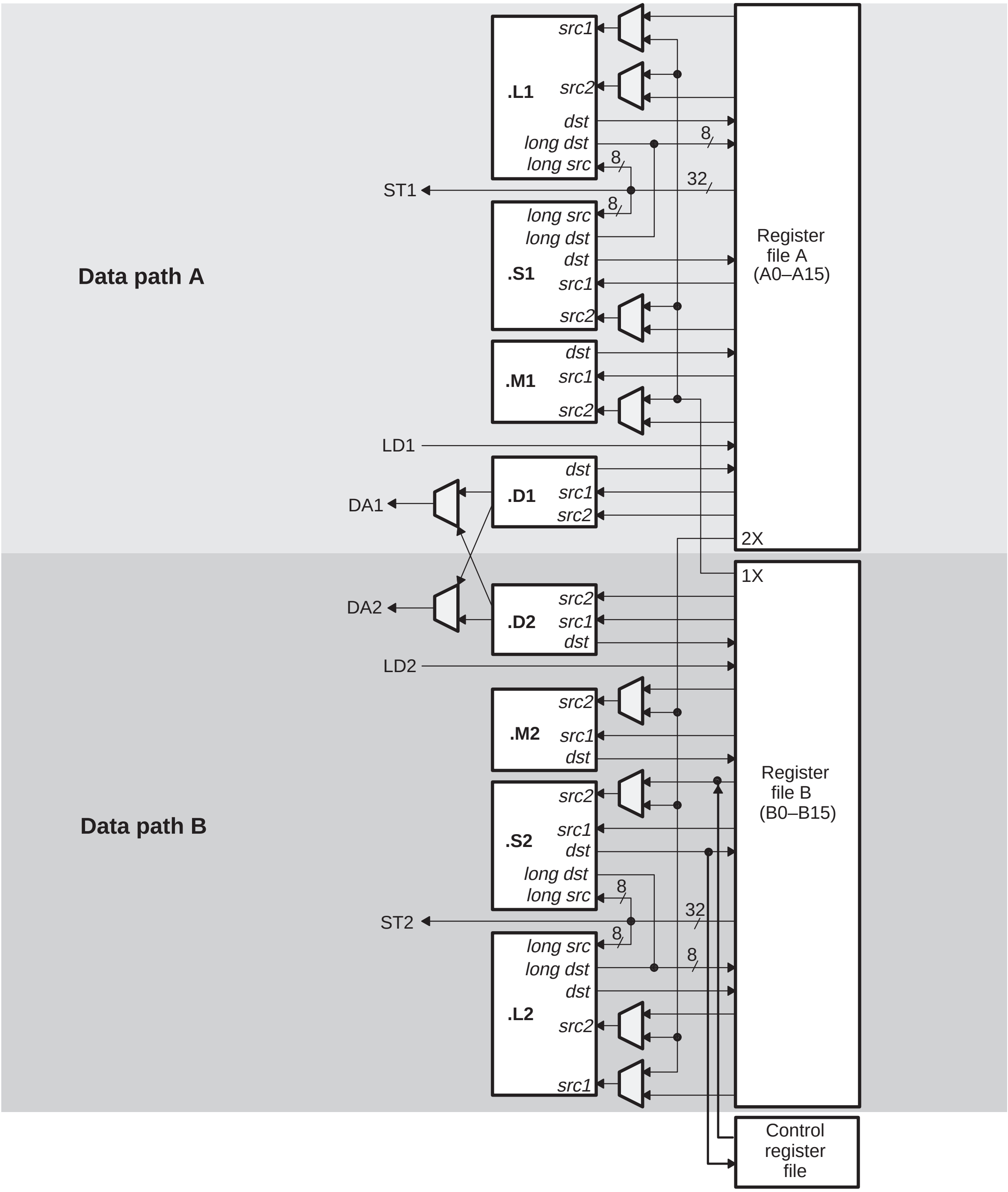


Figure 2–2. TMS320C67x CPU Data Paths

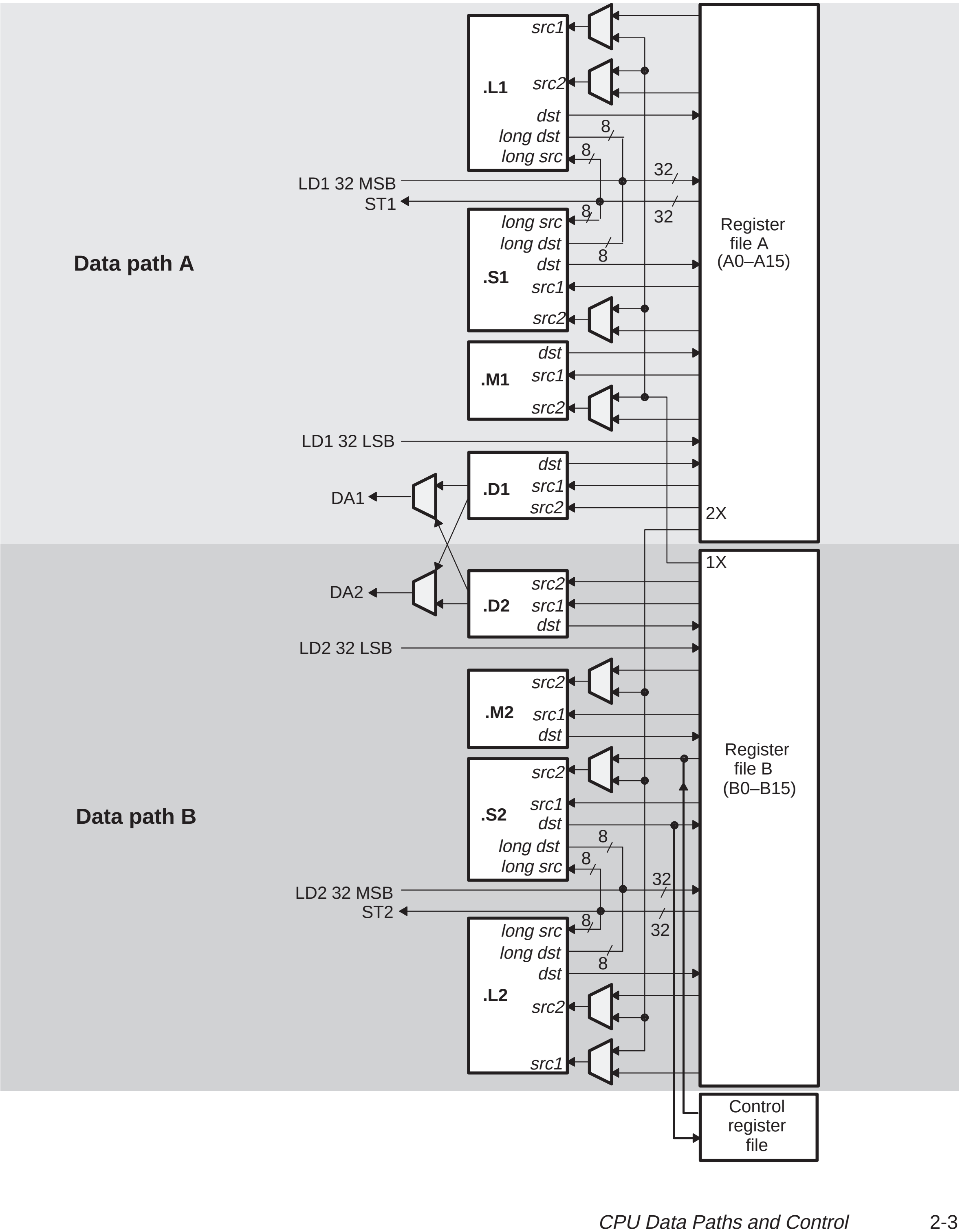
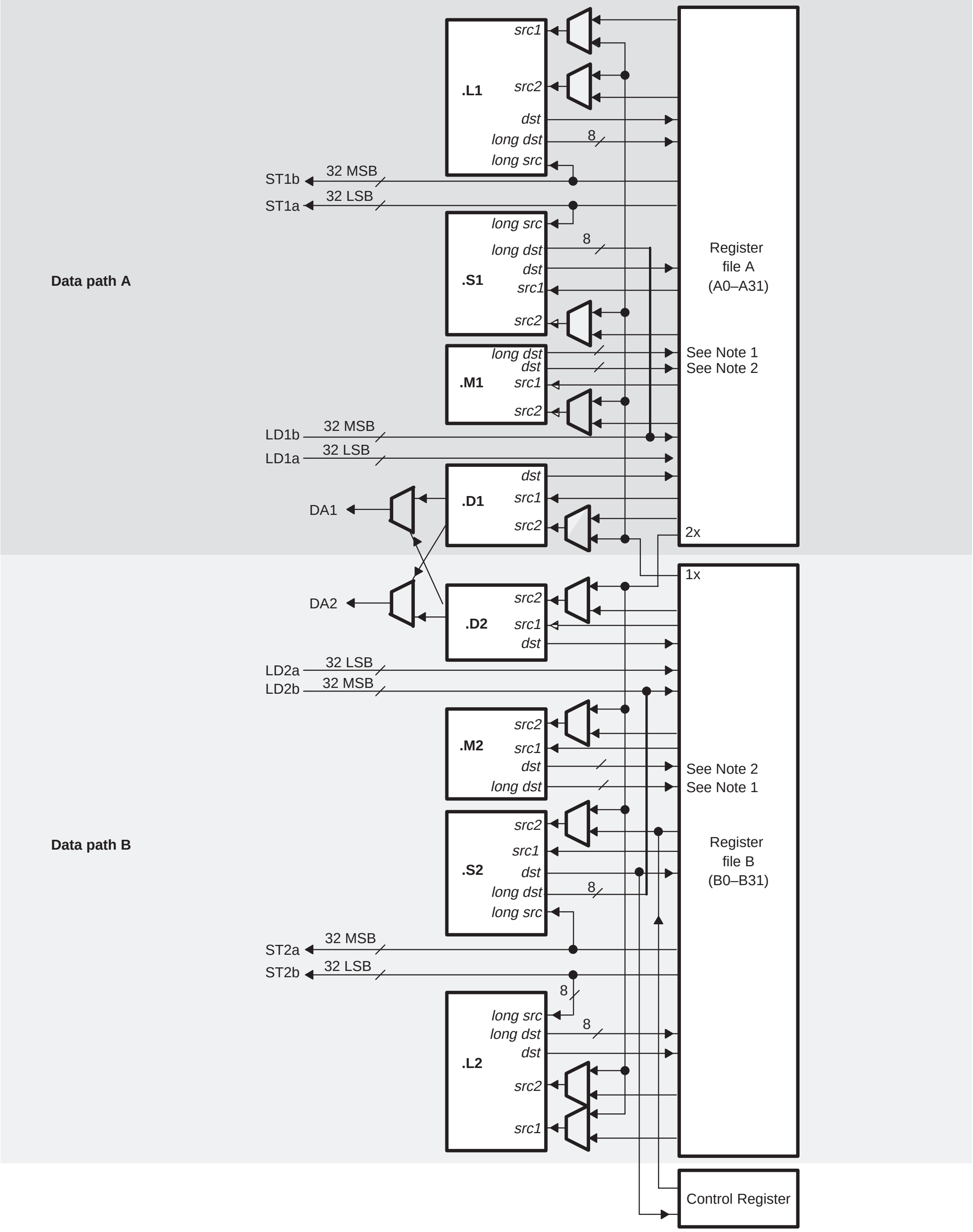


Figure 2–3. TMS320C64x CPU Data Path



Notes for .M unit:

1. *long dst* is 32 MSB
2. *dst* is 32 LSB

2.1 General-Purpose Register Files

There are two general-purpose register files (A and B) in the C6000™ data paths. For the C62x™/C67x™ DSPs, 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 C64x™ DSP register file doubles the number of general-purpose registers that are in the C62x/C67x cores, with 32 32-bit registers (A0–A31 for file A and B0–B31 for file B).

The C62x/C67x general-purpose register files support data ranging in size from packed 16-bit data through 40-bit fixed-point and 64-bit floating point data. Values larger than 32 bits, such as 40-bit long and 64-bit float quantities, are stored in register pairs. In these the 32 LSBs of data are placed in an even-numbered register and the remaining 8 or 32 MSBs in the next upper register (which is always an odd-numbered register). The C64x register file extends this by additionally supporting packed 8-bit types and 64-bit fixed-point data types. (The C64x does not directly support floating-point data types.) Packed data types store either four 8-bit values or two 16-bit values in a single 32-bit register, or four 16-bit values in a 64-bit register pair.

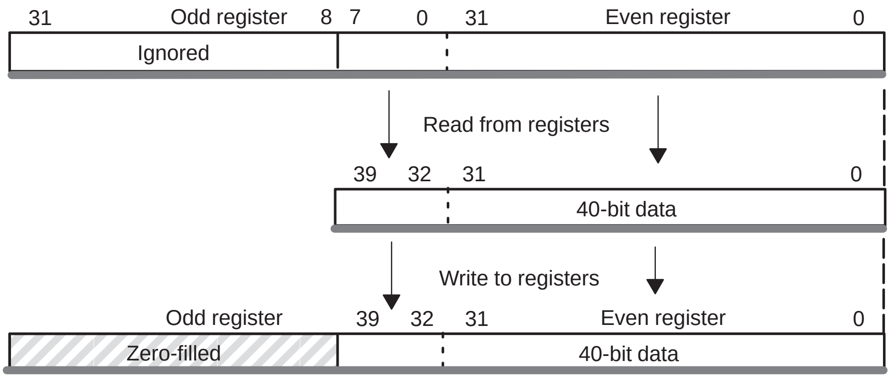
There are 16 valid register pairs for 40-bit and 64-bit data in the C62x/C67x cores, and 32 valid register pairs for 40-bit and 64-bit data in the C64x core, as shown in Table 2–1. In assembly language syntax, a colon between the register names denotes the register pairs, and the odd-numbered register is specified first.

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

Register Files		Applicable
A	B	Devices
A1:A0	B1:B0	C62x/C64x/C67x
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	
A17:A16	**B17:B16**	**C64x only**
A19:A18	**B19:B18**	
A21:A20	**B21:B20**	
A23:A22	**B23:B22**	
A25:A24	**B25:B24**	
A27:A26	**B27:B26**	
A29:A28	**B29:B28**	
A31:A30	**B31:B30**	

Figure 2–4 illustrates the register storage scheme for 40-bit long data. Operations requiring a long input ignore the 24 MSBs of the odd-numbered register. Operations producing a long result zero-fill the 24 MSBs of the odd-numbered register. The even-numbered register is encoded in the opcode.

Figure 2–4. Storage Scheme for 40-Bit Data in a Register Pair



2.2 Functional Units

The eight functional units in the C6000 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.

Besides being able to perform all the C62x instructions, the C64x also contains many 8-bit to 16-bit extensions to the instruction set. For example, the **MPYU4** instruction performs four 8x8 unsigned multiplies with a single instruction on an .M unit. The **ADD4** instruction performs four 8-bit additions with a single instruction on an .L unit. The additional C64x operations are shown in boldface 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 32-bit logical operations Leftmost 1 or 0 counting for 32 bits Normalization count for 32 and 40 bits **Byte shifts** **Data packing/unpacking** **5-bit constant generation** **Dual 16-bit arithmetic operations** **Quad 8-bit arithmetic operations** **Dual 16-bit min/max operations** **Quad 8-bit min/max 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 control register file (.S2 only) **Byte shifts** **Data packing/unpacking** **Dual 16-bit compare operations** **Quad 8-bit compare operations** **Dual 16-bit shift operations** **Dual 16-bit saturated arithmetic operations** **Quad 8-bit saturated arithmetic operations**	Compare Reciprocal and reciprocal square-root operations Absolute value operations SP → DP conversion operations

Table 2–2. Functional Units and Operations Performed (Continued)

Functional Unit	Fixed-Point Operations	Floating-Point Operations
.M unit (.M1, .M2)	16 x 16 multiply operations **16 x 32 multiply operations** **Quad 8 x 8 multiply operations** **Dual 16 x 16 multiply operations** **Dual 16 x 16 multiply with add/subtract operations** **Quad 8 x 8 multiply with add operation** **Bit expansion** **Bit interleaving/de-interleaving** **Variable shift operations** **Rotation** **Galois Field Multiply**	32 X 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 5-bit constant offset Loads and stores with 15-bit constant offset (.D2 only) **Load and store double words with 5-bit constant** **Load and store non-aligned words and double words** **5-bit constant generation** **32-bit logical operations**	Load doubleword with 5-bit constant offset

Note: Fixed-point operations are available on all three devices. Floating-point operations and 32 x 32-bit fixed-point multiply are available only on the C67x. Additonal C64x functions are shown in bold.

Most data lines in the CPU support 32-bit operands, and some support long (40-bit) and double word (64-bit) operands. Each functional unit has its own 32-bit write port into a general-purpose register file (Refer to Figure 2–3). 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, when performing 32-bit operations all eight units can be used in parallel every cycle. Since each C64x multiplier can return up to a 64-bit result, an extra write port has been added from the multipliers to the register file as compared to the C62x.

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 register file. The 1X cross path allows the functional units of data path A to read their source from register file B, and the 2X cross path allows the functional units of data path B to read their source from register file A.

On the C62x/C67x six of the eight functional units have access to the register file on the opposite side, via a cross path. The .M1, .M2, .S1 and .S2 units' *src2* units are selectable between the cross path and the same side register file. In the case of the .L1 and .L2, both *src1* and *src2* inputs are also selectable between the cross path and the same-side register file.

On the C64x all eight of the functional units have access to the register file on the opposite side, via a cross path. The *src2* inputs of .M1, .M2, .S1, .S2, .D1, and .D2 units are selectable between the cross path and the same-side register file. In the case of the .L1 and .L2, both *src1* and *src2* inputs are also selectable between the cross path and the same-side register file.

Only two cross paths, 1X and 2X, exist in the C6000 architecture. Thus the limit is one source read from each data path's opposite register file per cycle, or a total of two cross path source reads per cycle. In the C62x/C67x only one functional unit per data path, per execute packet, can get an operand from the opposite register file. In the C64x, multiple units on a side may read the same cross path source simultaneously. Thus the C64x cross path operand for one side may be used by up to two functional units on that side in an execute packet.

On the C64x a delay clock cycle is introduced whenever an instruction attempts to read a register via a cross path that was updated in the previous cycle. This is known as a cross path stall. This stall is inserted automatically by the hardware, no **NOP** instruction is needed. It should be noted that no stall is introduced if the register being read is the destination for data placed by an LDx instruction. For more information see Chapter 5, section 5.6.2 *Cross Path Stalls*. Techniques for avoiding this stall are discussed in the *TMS320C6000 Programmers Guide* (SPRU198).

2.4 Memory, Load, and Store Paths

The C62x has 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. This allows the LDDW instruction to simultaneously load two 32-bit values into register file A and two 32-bit values into register file B. For side A, LD1a is the load path for the 32 LSBs and LD1b is the load path for the 32 MSBs. For side B, LD2a is the load path for the 32 LSBs and LD2b is the load path for the 32 MSBs. There are also two 32-bit paths, ST1 and ST2, for storing register values to memory from each register file.

The C64x supports double word loads and stores. There are four 32-bit paths for loading data from memory to the register file. For side A, LD1a is the load path for the 32 LSBs and LD1b is the load path for the 32 MSBs. For side B, LD2a is the load path for the 32 LSBs and LD2b is the load path for the 32 MSBs. There are also four 32-bit paths, for storing register values to memory from each register file. ST1a is the write path for the 32 LSBs on side A and ST1b is the write path for the 32 MSBs on side A. For side B, ST2a is the write path for the 32 LSBs and ST2b is the write path for the 32 MSBs.

On the C6000 architecture, some of the ports for long and double word operands are shared between functional units. This places a constraint on which long or double word operations can be scheduled on a data path in the same execute packet. See Chapter 5, section 5.6.4, *Constraints on Long (40-Bit) Data.*

2.5 Data Address Paths

The data address paths DA1 and DA2 are each connected to the .D units in both data paths. This allows data addresses generated by any one path to access data to or from any register.

The DA1 and DA2 resources and their associated data paths are specified as T1 and T2 respectively. T1 consists of the DA1 address path and the LD1 and ST1 data paths. For the C64x and C67x, LD1 is comprised of LD1a and LD1b to support 64-bit loads. For the C64x, ST1 is comprised of ST1a and ST1b to support 64-bit stores. Similarly, T2 consists of the DA2 address path and the LD2 and ST2 data paths. For the C64x and C67x, LD2 is comprised of LD2a and LD2b to support 64-bit loads. For the C64x, ST2 is comprised of ST2a and ST2b to support 64-bit stores. The T1 and T2 designations appear in functional unit fields for load and store instructions.

For example, the following load instructions uses the .D1 unit to generate the address but is using the LD2 path resource from DA2 to place the data in the B register file. The use of the DA2 resource is indicated with the T2 designation.

```
LDW  .D1T2   *A0[3],B1
```

2.6 TMS320C6000 Control Register File

One unit (.S2) can read from and write to the control register file, as shown in this section. 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/C64x/C67x Fixed-Point Instruction Set*, for information on how to use this instruction.

Additionally, some of the control register bits are specially accessed in other ways. For example, arrival of a maskable interrupt on an external interrupt pin, INT*m*, triggers the setting of flag bit IFR*m*. Subsequently, when that interrupt is processed, this triggers the clearing of IFR*m* and the clearing of the global interrupt enable bit, GIE. Finally, when that interrupt processing is complete, the **B IRP** instruction in the interrupt service routine restores the pre-interrupt value of the GIE. Similarly, saturating instructions like **SADD** set the SAT (saturation) bit in the CSR (Control Status Register).

Table 2–3. Control Registers Common to C62x/C67x and C64x Cores

Abbreviation	Register 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	2-14
CSR	Control status register	Contains the global interrupt enable bit, cache control bits, and other miscellaneous control and status bits	2-17
IFR	Interrupt flag register	Displays status of interrupts	8-14
ISR	Interrupt set register	Allows manually setting pending interrupts	8-14
ICR	Interrupt clear register	Allows manually clearing pending interrupts	8-14
IER	Interrupt enable register	Allows enabling/disabling of individual interrupts	8-13
ISTP	Interrupt service table pointer	Points to the beginning of the interrupt service table	8-8
IRP	Interrupt return pointer	Contains the address to be used to return from a maskable interrupt	8-17
NRP	Nonmaskable interrupt return pointer	Contains the address to be used to return from a nonmaskable interrupt	8-16
PCE1	Program counter, E1 phase	Contains the address of the fetch packet that is in the E1 pipeline stage	2-19

2.6.1 Pipeline/Timing of Control Register Accesses

As shown in this section, all **MVC** are single-cycle instructions that complete their access of the explicitly named registers in the E1 pipeline phase. This is true whether **MVC** is moving a general register to a control register, or vice versa. In all cases the source register content is read, moved through the .S2 unit, and written to the destination register in the E1 pipeline phase.

Pipeline Stage	**E1**
Read	*src2*
Written	*dst*
Unit in use	.S2

Even though **MVC** modifies the particular target control register in a single cycle, it can take extra clocks to complete modification of the non-explicitly named register. For example, the **MVC** cannot modify bits in the IFR directly. Instead, **MVC** can only write 1's into the ISR or the ICR to specify setting or clearing, respectively, of the IFR bits. **MVC** completes this ISR/ICR write in a single (E1) cycle (as described above) but the modification of the IFR bits themselves occur one clock later. For more information on the manipulation of ISR, ICR, and IFR see these control hardware sections in Chapter 8: section 8.3.2 *Status of, Setting, and Clearing Interrupts,* and section 8.3.3 *Returning from Interrupt Servicing.*

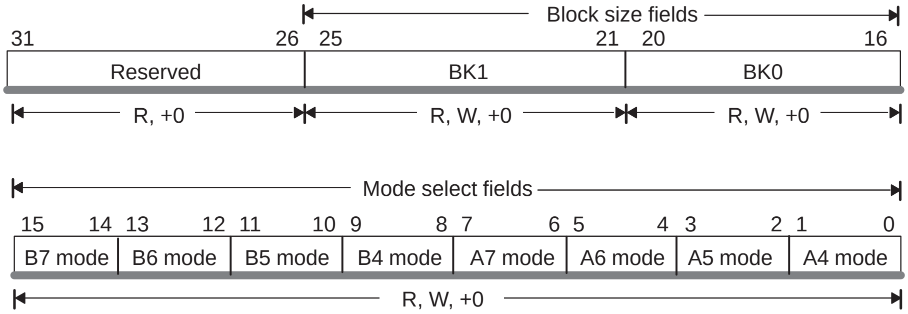
Saturating instructions, such as **SADD**, set the saturation flag bit (SAT) in the Control Status Register (CSR) indirectly. As a result, several of these instructions update the SAT bit one full clock cycle after their primary results are written to the register file. For example, the **SMPY** instruction writes its result at the end of pipeline stage E2; its primary result is available after one delay slot. In contrast, the SAT bit in the CSR is updated one cycle later than the result is written; this update occurs after two delay slots. (For the specific behavior of an instruction, refer to the description of that individual instruction).

The **B IRP** and **B NRP** instructions directly update the GIE and NMIE, respectively. Because these branches directly modify the CSR and IER (Interrupt Enable Register) respectively, there are no delay slots between when the branch is issued and when the control register updates take effect.

2.6.2 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 2–5, and the mode select field encoding is shown in Table 2–4.

Figure 2–5. 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

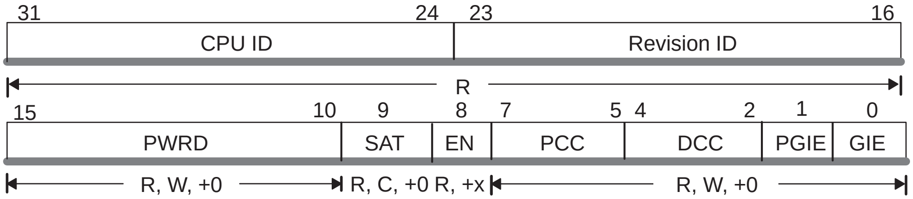
N	Block Size	N	Block Size
00000	2	10000	131 072
00001	4	10001	262 144
00010	8	10010	524 288
00011	16	10011	1 048 576
00100	32	10100	2 097 152
00101	64	10101	4 194 304
00110	128	10110	8 388 608
00111	256	10111	16 777 216
01000	512	11000	33 554 432
01001	1 024	11001	67 108 864
01010	2 048	11010	134 217 728
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

Note: When N is 11111, the behavior is identical to linear addressing.

2.6.3 Control Status Register (CSR)

The CSR, shown in Figure 2–6, 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 the data sheet of your specific device to see if it supports the options that these fields control. Also see the *C6000 Peripherals Reference Guide* for more information on these options.

Figure 2–6. 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

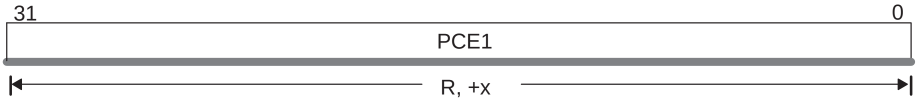
<table>
<tr><th>Bit Position</th><th>Width</th><th>Field Name</th><th>Function</th></tr>
<tr><td></td><td></td><td>CPU ID and Revision ID</td><td></td></tr>
<tr><td>31-24</td><td>8</td><td>CPU ID</td><td>CPU ID identifies which of these CPUs:
CPU ID = 00b: indicates C62x,
CPU ID= 10b: indicates C67x,
CPU ID = 1000b: indicates C64x</td></tr>
<tr><td>23-16</td><td>8</td><td>Revision ID</td><td>Revision ID defines silicon revision of the CPU.
<table>
<tr><th>Devices</th><th>CPU</th><th>Core Voltage</th><th>Bits 31:16 of CSR</th></tr>
<tr><td>C6201</td><td>C62x</td><td>2.5V</td><td>0x0001</td></tr>
<tr><td>C6201B, C6202, C6211</td><td>C62x</td><td>1.8V</td><td>0x0002</td></tr>
<tr><td>C6202B, C6203, C6204, C6205</td><td>C62x</td><td>1.5V</td><td>0x0003</td></tr>
<tr><td>C6701 revision 0 (early CPU)</td><td>C67x</td><td>1.8V</td><td>0x0201</td></tr>
<tr><td>C6701, C6711, C6712</td><td>C67x</td><td>1.8V</td><td>0x0202</td></tr>
<tr><td>C64xx</td><td>C64x</td><td>1.5V</td><td>0x0801</td></tr>
</table></td></tr>
<tr><td>15-10</td><td>6</td><td>PWRD</td><td>Control power-down modes; the values are always read as zero.[†]</td></tr>
<tr><td>9</td><td>1</td><td>SAT</td><td>The saturate bit, set when any unit performs a saturate, can be cleared only by the <b>MVC</b> instruction and can be set only by a functional unit. The set by a functional unit has priority over a clear (by the <b>MVC</b> 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.</td></tr>
<tr><td>8</td><td>1</td><td>EN</td><td>Endian bit: 1 = little endian, 0 = big endian [†]</td></tr>
<tr><td>7-5</td><td>3</td><td>PCC</td><td>Program cache control mode[†]</td></tr>
<tr><td>4-2</td><td>3</td><td>DCC</td><td>Data cache control mode[†]</td></tr>
<tr><td>1</td><td>1</td><td>PGIE</td><td>Previous GIE (global interrupt enable); saves GIE when an interrupt is taken.</td></tr>
<tr><td>0</td><td>1</td><td>GIE</td><td>Global interrupt enable; enables (1) or disables (0) all interrupts except the reset interrupt and NMI (nonmaskable interrupt).</td></tr>
</table>

[†] See the *TMS320C6000 Peripherals Reference Guide* for more information.

2.6.4 E1 Phase Program Counter (PCE1)

The PCE1, shown in Figure 2–7, contains the 32-bit address of the fetch packet in the E1 pipeline phase.

Figure 2–7. E1 Phase Program Counter (PCE1)



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

2.7 TMS320C67x Control Register File Extensions

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.

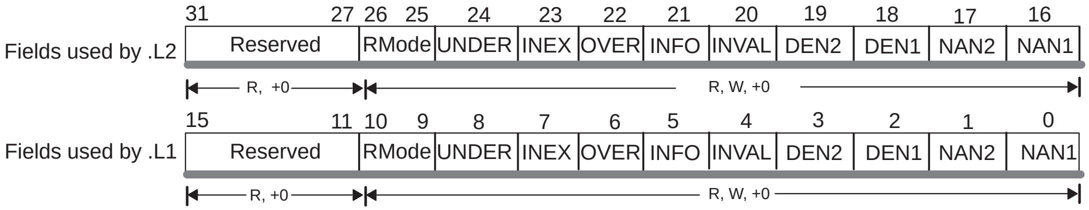
Table 2–7. Control Register File Extensions

Register		**Description**	**Page**
Abbreviation	**Name**		
FADCR	Floating-point adder configuration register	Specifies underflow mode, rounding mode, NaNs, and other exceptions for the .L unit.	2-20
FAUCR	Floating-point auxiliary configuration register	Specifies underflow mode, rounding mode, NaNs, and other exceptions for the .S unit.	2-22
FMCR	Floating-point multiplier configuration register	Specifies underflow mode, rounding mode, NaNs, and other exceptions for the .M unit.	2-24

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 2–8 shows the layout of FADCR. The functions of the fields in the FADCR are shown in Table 2–8.

Figure 2–8. Floating-Point Adder Configuration Register (FADCR)



Fields used by .L2	31–27	26–25	24	23	22	21	20	19	18	17	16
	Reserved	RMode	UNDER	INEX	OVER	INFO	INVAL	DEN2	DEN1	NAN2	NAN1
	R, +0	R, W, +0									

Fields used by .L1	15–11	10–9	8	7	6	5	4	3	2	1	0
	Reserved	RMode	UNDER	INEX	OVER	INFO	INVAL	DEN2	DEN1	NAN2	NAN1
	R, +0	R, W, +0									

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 2–9 shows the layout of FAUCR. The functions of the fields in the FAUCR are shown in Table 2–9.

Figure 2–9. Floating-Point Auxiliary Configuration Register (FAUCR)

	31 27	26	25	24	23	22	21	20	19	18	17	16
Fields used by .S2	Reserved	DIV0	UNORD	UND	INEX	OVER	INFO	INVAL	DEN2	DEN1	NAN2	NAN1
	R, +0	R, W, +0										

	15 11	10	9	8	7	6	5	4	3	2	1	0
Fields used by .S1	Reserved	DIV0	UNORD	UND	INEX	OVER	INFO	INVAL	DEN2	DEN1	NAN2	NAN1
	R, +0	R, W, +0										

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 2–10 shows the layout of FMCR. The functions of the fields in the FMCR are shown in Table 2–10.

Figure 2–10. Floating-Point Multiplier Configuration Register (FMCR)

	31 27	26 25	24	23	22	21	20	19	18	17	16
Fields used by .M2	Reserved	RMode	UNDER	INEX	OVER	INFO	INVAL	DEN2	DEN1	NAN2	NAN1
	R, +0	R, W, +0									

	15 11	10 9	8	7	6	5	4	3	2	1	0
Fields used by .M1	Reserved	RMode	UNDER	INEX	OVER	INFO	INVAL	DEN2	DEN1	NAN2	NAN1
	R, +0	R, W, +0									

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

2.8 TMS320C64x Control Register File Extensions

One new control register in the C64x is the Galois Field Polynomial Generator Function Register (GFPGFR).

2.8.1 Galois Field

Modern digital communication systems typically make use of error correction coding schemes to improve system performance under imperfect channel conditions. The scheme most commonly used is the Reed-Solomon code, due to its robustness against burst errors and its relative ease of implementation.

The C64x contains Galois Field Multiply hardware that can be used for Reed-Solomon encode and decode functions. To understand the relevance of the Galois Field Multiply hardware, it is necessary to first define some mathematical terms.

Two kinds of number systems that are common in algorithm development are integers and real numbers. For integers the addition, subtraction and multiplication operations can be performed. Division can also be performed if a non-zero remainder can be allowed. For real numbers all four of these operations can be performed, even if there is a non-zero remainder for division operations.

Real numbers can belong to a mathematical structure called a field. A field consists of a set of data elements along with addition, subtraction, multiplication, and division. A field of integers can also be created if modulo arithmetic is performed.

An example is doing arithmetic using integers modulo 2. Perform the operations using normal integer arithmetic and then take the result modulo 2. Table 2–11 describes addition, subtraction and multiplication modulo 2.

Table 2–11. Modulo 2 Arithmetic

Addition

+	0	1
0	0	1
1	1	0

Subtraction

–	0	1
0	0	1
1	1	0

Multiplication

×	0	1
0	0	0
1	0	1

Note that addition and subtraction results are the same, and in fact are equivalent to the **XOR** (exclusive **OR**) operation in binary. Also, the multiplication result is equal to the **AND** operation in binary. These properties are unique to modulo 2 arithmetic, but modulo 2 arithmetic is used extensively in error correction coding. Another more general property is that division by any non-zero element is now defined. Division can always be performed if every element other than zero has a multiplicative inverse, i.e.:

$x \cdot x^{-1} = 1.$

Another example, arithmetic modulo 5, illustrates this concept more clearly. The addition, subtraction and multiplication tables are given in Table 2–12.

Table 2–12. Modulo 5 Arithmetic

Addition

+	0	1	2	3	4
0	0	1	2	3	4
1	1	2	3	4	0
2	2	3	4	0	1
3	3	4	0	1	2
4	4	0	1	2	3

Subtraction

–	0	1	2	3	4
0	0	4	3	2	1
1	1	0	4	3	2
2	2	1	0	4	3
3	3	2	1	0	4
4	4	3	2	1	0

Multiplication

×	0	1	2	3	4
0	0	0	0	0	0
1	0	1	2	3	4
2	0	2	4	1	3
3	0	3	1	4	2
4	0	4	3	2	1

In the rows of the multiplication table, it is clear that the element 1 appears in every non-zero row and column. Every non-zero element can be multiplied by at least one other element to get a result equal to 1. Therefore, division always works and arithmetic over integers modulo 5 forms a field. Fields generated in this manner are called finite fields or Galois fields and are written as GF(X), such as GF(2) or GF(5). They only work when the arithmetic performed is modulo a prime number.

Galois fields can also be formed where the elements are vectors instead of integers if polynomials are used. Finite fields therefore can be found with a number of elements equal to any power of a prime number. Typically we are interested in implementing error correction coding systems using binary arithmetic. All of the fields that are dealt with in Reed Solomon coding systems are of the form GF(2^m). This allows performing addition using **XORs** on the coefficients of the vectors, and multiplication using a combination of **ANDs** and **XORs**.

A final example considers the field GF(2^3), which had 8 elements. This can be generated by arithmetic modulo the (irreducible) polynomial $P(x) = x^3 + x + 1$. Elements of this field look like vectors of three bits.

Table 2–13 shows the addition and multiplication tables for field GF(2^3).

Table 2–13. Modulo Arithmetic for Field GF(2^3)

Addition

+	000	001	010	011	100	101	110	111
000	000	001	010	011	100	101	110	111
001	001	000	011	010	101	110	111	110
010	010	011	000	001	110	111	100	101
011	011	010	001	000	111	110	101	100
100	100	101	110	111	000	001	010	011
101	101	100	111	110	001	000	011	010
110	110	111	100	101	010	011	000	001
111	111	110	101	100	011	010	001	000

Multiplication

×	000	001	010	011	100	101	110	111
000	000	000	000	000	000	000	000	000
001	000	001	010	011	100	101	110	111
010	000	010	100	110	011	001	111	101
011	000	011	110	101	111	100	001	010
100	000	100	011	111	110	010	101	001
101	000	101	001	100	010	111	011	110
110	000	110	111	001	101	011	010	100
111	000	111	101	010	001	110	100	011

Note that the value 1 (001) appears in every non-zero row of the multiplication table, which indicates that this is a valid field.

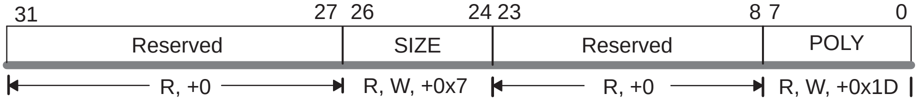
The channel error can now be modeled as a vector of bits, with a one in every bit position that an error has occurred, and a zero where no error has occurred. Once the error vector has been determined, it can be subtracted from the received message to determine the correct code word.

The Galois Field Multiply hardware on the C64x is named **GMPY4**. This instruction performs four parallel operations on 8-bit packed data on the .M unit. The Galois Field Multiplier can be programmed to perform all Galois Multiplies for fields of the form GF(2^m), where m can range between 1 and 8 using any generator polynomial. The field size and the polynomial generator are controlled by the Galois Field Polynomial Generator Function Register (GFPGFR).

The GFPGFR, shown in Figure 2–11, contains the Galois field polynomial generator and the field size control bits. These bits control the operation of the **GMPY4** instruction. This register can only be set via the **MVC** instruction.

The default function after reset for the **GMPY4** instruction is field size=7 and polynomial=0x1D.

Figure 2–11. Galois Field Polynomial Generator Function Register (GFPGFR)



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

Table 2–14. GFPGFR Field Descriptions

Bit Position	Width	Field Name	Function
31–27	5	Reserved	Reserved. Read as zero. Ignored on write.
26–24	3	SIZE	Field size.
23–8	16	Reserved	Reserved. Read as zero. Ignored on write.
7–0	8	POLY	Polynomial Generator.

2.8.2 Special Timing Considerations

If the next execute packet after an **MVC** instruction that changes the GFPGFR value contains a **GMPY4** instruction, then the **GMPY4** will be controlled by the newly loaded GFPGFR value.

2.9 Summary of TMS320C64x Architecture Key Extensions

This summary lists the key extensions of the C64x architecture, as compared to the C62x architecture. The C64x is a code-compatible member of the C6000 DSP family. The C64x provides a superset of the C62x architecture and also complete code compatibility for existing C62x code running on a C64x device. The C64x provides extensions to the existing C62x architecture in these areas:

- ❑ Register file enhancements
- ❑ Data path extensions
- ❑ Quad 8-bit and dual 16-bit extensions with data flow enhancements
- ❑ Additional functional unit hardware
- ❑ Increased orthogonality of the instruction set
- ❑ Additional instructions that reduce code size and increase register flexibility.

These areas are described in more detail below.

Register File Enhancements:

- ❑ The C64x has twice as many registers. The C64x has 64 general-purpose registers, whereas the C62x has 32 general-purpose registers.

- ❑ The C62x uses A1, A2, B0, B1 and B2 as conditional registers. In addition to these, the C64x can also use A0 as a conditional register.

- ❑ The C62x register file supports packed 16-bit data types in addition to 32-bit and 40-bit data types. The C64x register file extends this by supporting packed 8-bit types and 64-bit types.

Data Path Extensions:

- ❑ The .D unit in the C64x can load and store double words (64 bits) with a single instruction, whereas the .D unit in the C62x cannot with a single instruction.

- ❑ The .D unit in the C64x can now access operands via a data cross path similar to the .L, .M, and .S functional units. In the C62x only address cross paths on the .D unit are supported.

- ❑ The C64x pipelines data cross path accesses over multiple cycles. This allows the same register to be used as a data cross path operand by multiple functional units in the same execute packet. In the C62x, only one cross operand is allowed per side. This additional pipelining by the C64x can cause a delay clock cycle known as a cross path stall under certain conditions. See Chapter 5, section 5.6.2 *Cross Path Stalls*.
- ❑ In the C64x, up to two long sources and two long results can be accessed on each data path every cycle. In the C62x only one long source and one long result per data path could occur every cycle.

Additional Functional Unit Hardware:

- ❑ Each .M unit can now perform two 16x16 bit multiplies or four 8x8 bit multiplies every clock cycle.
- ❑ The .D units in the C64x can now access words and double words on any byte boundary by using non-aligned load and store instructions. The C62x only provides aligned load and store instructions.
- ❑ The .L units can perform byte shifts and the .M units can perform bi-directional variable shifts, in addition to the .S unit's ability to do shifts. The bi-directional shifts directly assist voice-compression codecs (vocoders).
- ❑ The .L units can perform quad 8-bit subtends with absolute value. This absolute difference instruction greatly aids motion estimation algorithms.
- ❑ Special communications-specific instructions, such as **SHFL, DEAL** and **GMPY4** have been added to the .M unit to address common operations in error-correcting codes.
- ❑ Bit-count and rotate hardware on the .M unit extends support for bit-level algorithms; such as binary morphology, image metric calculations and encryption algorithms.

Increased Orthogonality of the Instruction Set:

- ❑ The .D unit can now perform 32-bit logical instructions, in addition to the .S and .L units.
- ❑ The .D unit in the C64x now directly supports load and store instructions for double word data values. The C62x does not directly support loads and stores of double words, and the C67x only directly supports loads of double words.
- ❑ The .L, and .D units can now be used to load 5-bit constants, in addition to the .S unit's ability to load 16-bit constants.

Quad 8-bit and Dual 16-bit Extensions with Data Flow Enhancements:

❑ Extensive collection of PACK and byte shift instructions simplifies manipulation of packed data types.

❑ Instructions have been added in the C64x that operate directly on packed data to streamline data flow and increase instruction set efficiency. The C64x has a comprehensive collection of 8-bit and 16-bit instruction set extensions, which are shown in Table 2–15.

Table 2–15. C64x 8-Bit and 16-Bit Instruction Set Extensions.

Operation	Quad 8-bit	Dual 16-bit
Multiply	X	X
Multiply with Saturation		X
Addition/Subtraction	X	X[†]
Addition with Saturation	X	X
Absolute Value		X
Subtract with Absolute Value	X	
Compare	X	X
Shift		X
Data Pack/Unpack	X	X
Data Pack with Saturation	X	X
Dot-product with Optional Negate	X[‡]	X
Min/Max/Average	X	X
Bit-expansion (Mask Generation)	X	X

[†] The C62x provides support for 16-bit data with the ADD2/SUB2 instructions. The C64x extends this support to include 8-bit data.

[‡] Dot-product with negate is not available for 8-bit data.

Additional Instructions that Reduce Code Size and Increase Register Flexibility:

❑ **BDEC** and **BPOS** combine a branch instruction with the decrement/test positive of a destination register respectively. These instructions help reduce the number of instructions needed to decrement a loop counter and conditionally branch based upon the value of that counter. Any register can be used as the loop counter, which can free the standard conditional registers (A0–A2 and B0–B2) for other uses.

❑ The **ADDKPC** instruction helps reduce the number of instructions needed to set up the return address for a function call.

❑ The **BNOP** instruction helps reduce the number of instructions required to perform a branch when **NOPs** are needed to fill the delay slots of a branch.

❑ Execute packet boundary restrictions are removed, thereby eliminating all of the **NOPs** added to pad fetch packets.

Chapter 3

TMS320C62x/C64x/C67x Fixed-Point Instruction Set

The TMS320C62x™, TMS320C64x™, and the TMS320C67x™ share an instruction set. All of the instructions valid for the C62x™ are also valid for the C64x™ and 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. Similarly, the C64x adds functionality to the C62x with some unique instructions. This chapter describes the assembly language instructions that are common to the C62x, C64x, 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.

Instructions unique to the C64x are described in Chapter 5.

Topic **Page**

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
→	Assignment
+	Addition
×	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/C64x/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/C64x/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/C64x/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/C64x/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/C64x/C67x Opcode Map

Table 3–4 and the instruction descriptions in this chapter explain the field syntaxes and values. The C62x, C64x, and C67x opcodes are mapped in Figure 3–1.

Table 3–4. TMS320C62x/C64x/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/C64x/C67x Opcode Map

Operations on the .L unit

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					

Operations on the .M unit

31 29	28	27 23	22 18	17 13	12	11 7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1/cst*	x	*op*	0	0	0	0	0	*s*	*p*
3		5	5	5		5							

Operations on the .D unit

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

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

31 29	28	27 23	22 8	7	6 4	3	2	1	0
creg	*z*	*dst/src*	*ucst15*	y	*ld/st*	1	1	*s*	*p*
3		5	15		3				

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

31 29	28	27 23	22 18	17 13	12 9	8	7	6 4	3	2	1	0
creg	*z*	*dst/src*	*baseR*	*offsetR/ucst5*	*mode*	*r*	y	*ld/st*	0	1	*s*	*p*
3		5	5	5	4			3				

Operations on the .S unit

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

ADDK on the .S unit

31 29	28	27 23	22 7	6	5	4	3	2	1	
creg	*z*	*dst*	*cst*	1	0	1	0	0	*s*	*p*
3		5	16							

Figure 3–1. TMS320C62x/C64x/C67x Opcode Map (Continued)

Field operations (immediate forms) on the .S unit

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*	*op*	0	0	1	0	*s*	*p*
3		5	5	5	5	2						

MVK and MVKH on the .S unit

31 29	28	27 23	22 7	6	5	4	3	2	1	0
creg	z	*dst*	*cst*	*h*	1	0	1	0	*s*	*p*
3		5	16							

Bcond disp on the .S unit

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

IDLE

31 18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	*s*	*p*
14																		1

NOP

31 18	17	16 13												0
Reserved	0	*src*	0	0	0	0	0	0	0	0	0	0	0	0	*p*
14		4												1

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, C64x, 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

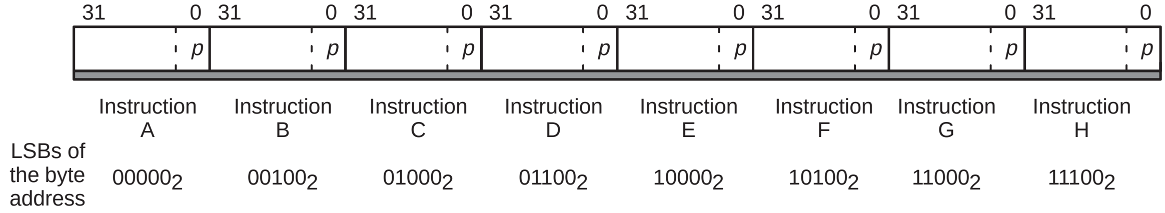
Instruction Type	Delay Slots	Functional Unit Latency	Read Cycles[†]	Write Cycles[†]	Branch Taken[†]
NOP (no operation)	0	1			
Store	0	1	i	i	
Single cycle	0	1	i	i	
Multiply (16 × 16)	1	1	i	i + 1	
Load	4	1	i	i, i + 4[§]	
Branch	5	1	i[‡]		i + 5

[†] 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.

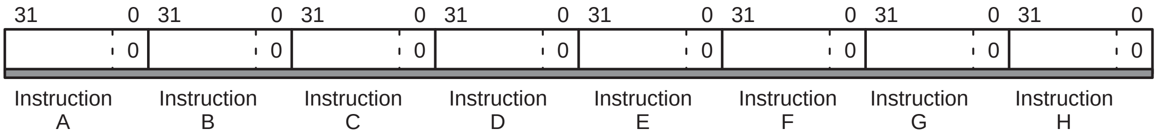
An execute packet cannot cross an 8-word boundary. Therefore, the last *p*-bit in a fetch packet is always set to 0, and each fetch packet starts a new execute packet. There are three types of *p*-bit patterns for fetch packets. These three *p*-bit patterns result in the following execution sequences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



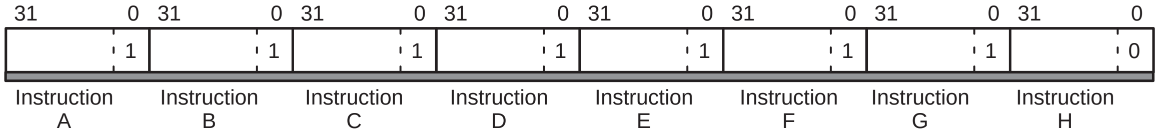
results in this execution sequence:

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:



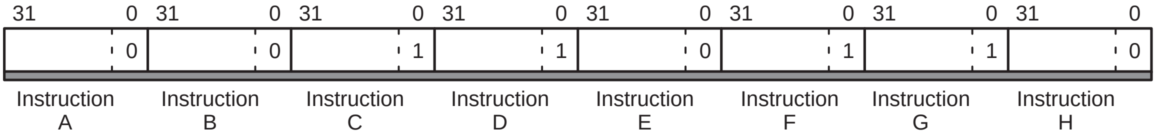
results in this execution sequence:

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 6, *TMS320C62x/C64x Pipeline*, and Chapter 7, *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. The C64x can also use A0 as a conditional register. Please see Chapter 5, section 5.5 *Conditional Operations.*

[†] This value is reserved for software breakpoints that are used for emulation purposes.

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. Chapter 5, section 5.6 discusses the differences between the C62x/C67x and the C64x with respect to resource constaints.

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 2-5 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, C64x, 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.

Additional information on addressing modes for the C64x can be found in Chapter 5.

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 2.6.2, on page 2-14, 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 × 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 3–4 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
```

	Before LDW	**1 cycle after LDW**	**5 cycles after LDW**
A4	0000 0100h	0000 0104h	0000 0104h
A1	XXXX XXXXh	XXXX XXXXh	1234 5678h
mem 104h	1234 5678h	1234 5678h	1234 5678h

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 × 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 3–5 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
```

	Before ADDAH	**1 cycle after ADDAH**
A4	0000 0100h	0000 0106h
A1	0000 0013h	0000 0013h

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, C64x, 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, C64x, 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.

Example

The way each instruction is described.

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 → dst
else         nop
```

Execution for .D1, .D2 Opcodes

```
if (cond)    src2 + src1 → 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 6.2 on page 6-14 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.

ABS — *Integer Absolute Value With Saturation*

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

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

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

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

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

ADD(U)

Signed or Unsigned Integer Addition Without Saturation

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

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					

Opcode .S unit

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

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

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

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    A1,26,A6
```

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

ADDAB/ADDAH/ADDAW *Integer Addition Using Addressing Mode*

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

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

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 2.6.2). *src1* is left shifted by 1 or 2 for halfword and word data sizes respectively. Byte, halfword, and word mnemonics are **ADDAB**, **ADDAH**, and **ADDAW**, respectively. The result is placed in *dst*.

Execution

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

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	0000 000Bh
A4	0000 0100h	0000 0103h
AMR	0002 0001h	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	0000 000Bh
A4	0000 0100h	0000 0106h
AMR	0002 0001h	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	0002 0000h
AMR	0002 0001h	0002 0001h

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

ADDK — *Integer Addition Using Signed 16-Bit Constant*

Syntax

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

.unit = .S1 or .S2

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

Opcode

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

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
```

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

ADD2 — *Two 16-Bit Integer Adds on Upper and Lower Register Halves*

Syntax

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

.unit = .S1 or .S2

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

Opcode

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

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	

AND

Bitwise AND

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:

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					

.S unit form:

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

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

B — *Branch Using a Displacement*

Syntax

B (.unit) label

.unit = .S1 or .S2

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

Opcode

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

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

		Target Instruction					
Pipeline Stage	**E1**	**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	

B — *Branch Using a Register*

Syntax

B (.unit) *src2*

.unit = .S2

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

Opcode

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

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 → 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	

B IRP

Branch Using an Interrupt Return Pointer

Syntax

B (.unit) IRP

.unit = .S2

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

Opcode

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

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 → 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

B NRP

Branch Using NMI Return Pointer

Syntax

B (.unit) NRP

.unit = .S2

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

Opcode

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

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 → 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

CLR — *Clear a Bit Field*

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:

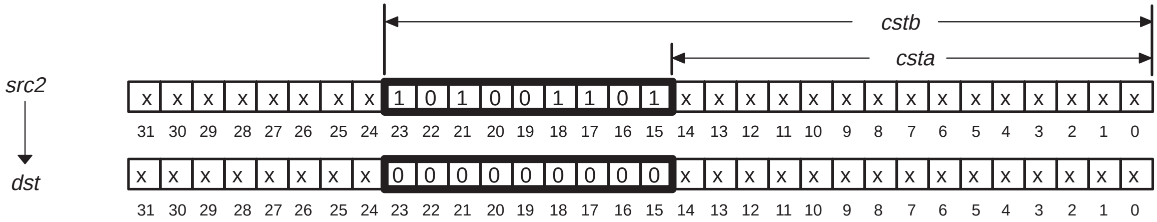
31 29	28	27 23	22 18	17 13	12 8	7 6	5		0
creg	*z*	*dst*	*src2*	*csta*	*cstb*	1 0	0 0 1 0	*s*	*p*
3	1	5	5	5	5	2		1	1

Register form:

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

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	03B6 E7D5h
B2	XXXX XXXXh	03B0 0001h
B3	0000 0052h	0000 0052h

CMPEQ Integer Compare for Equality

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

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 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

CMPGT(U) *Signed or Unsigned Integer Compare for Greater Than*

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

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

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.

Note:

The **CMPGT** instruction allows using a 5-bit constant as *src1.* If *src2* is a 5-bit constant, as in

```
CMPGT  .L1   A4, 5, A0
```

Then to implement this operation, the assembler converts this instruction to

```
CMPLT  .L1   5, A4, A0
```

These two instructions are equivalent, with the second instruction using the conventional operand types for *src1* and *src2.*

Similarly, the **CMPGT** instruction allows a cross path operand to be used as *src2.* If *src1* is a cross path operand as in

```
CMPGT  .L1x   B4, A5, A0
```

Then to implement this operation the assembler converts this instruction to

```
CMPLT  .L1x   A5, B4, A0
```

In both of these operations the listing file (.lst) will have the first implementation, and the second implementation will appear in the debugger.

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

CMPLT(U) *Signed or Unsigned Integer Compare for Less Than*

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

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

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*.

Note:

The **CMPLT** instruction allows using a 5-bit constant as *src1.* If *src2* is a 5-bit constant, as in

```
CMPLT  .L1   A4, 5, A0
```

Then to implement this operation, the assembler converts this instruction to

```
CMPGT  .L1   5, A4, A0
```

These two instructions are equivalent, with the second instruction using the conventional operand types for *src1* and *src2.*

Similarly, the **CMPLT** instruction allows a cross path operand to be used as *src2.* If *src1* is a cross path operand as in

```
CMPLT  .L1x   B4, A5, A0
```

Then to implement this operation, the assembler converts this instruction to

```
CMPGT  .L1x   A5, B4, A0
```

In both of these operations the listing file (.lst) will have the first implementation, and the second implementation will appear in the debugger.

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

EXT Extract and Sign-Extend a Bit Field

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:

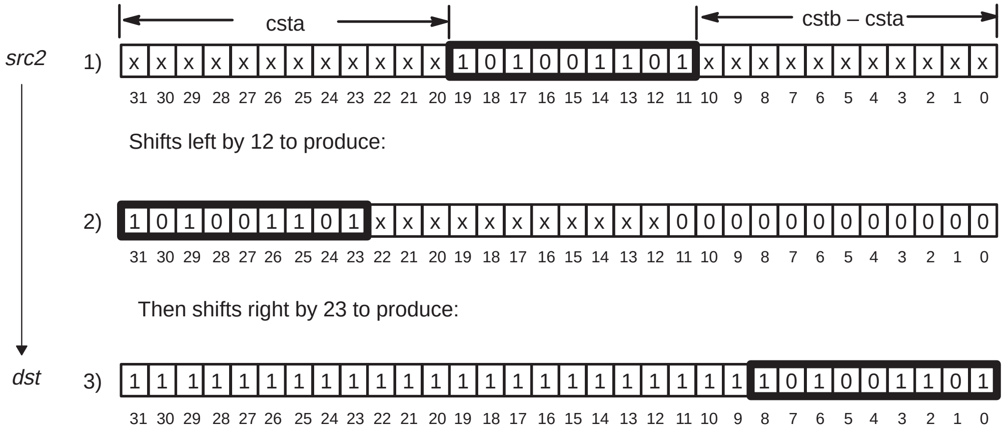
31 29	28	27 23	22 18	17 13	12 8	7 6	5		0
creg	*z*	*dst*	*src2*	*csta*	*cstb*	0 1	0 0 1 0	*s*	*p*
3	1	5	5	5	5	2		1	1

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

EXTU

Extract and Zero-Extend a Bit Field

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:

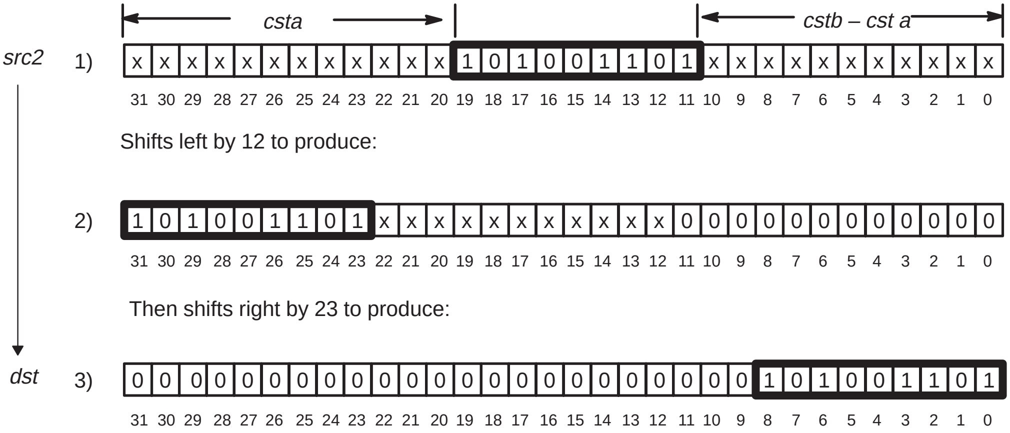
31 29	28	27 23	22 18	17 13	12 8	7 6	5 0
creg	*z*	*dst*	*src2*	*csta*	*cstb*	0 0	0 0 1 0 *s* *p*
3	1	5	5	5	5	2	1 1

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

IDLE *Multicycle NOP With No Termination Until Interrupt*

Syntax **IDLE**

Opcode

31 18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	*s*	*p*
14																		1

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

LDB(U)/LDH(U)/LDW

Load From Memory With a 5-Bit Unsigned Constant Offset or Register Offset

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

31 29	28	27 23	22 18	17 13	12 9	8	7	6 4	3	2	1	0
creg	*z*	*dst*	*baseR*	*offsetR/ucst5*	*mode*	r	y	*ld/st*	0	1	*s*	*p*
3		5	5	5	4			3				

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 2.6.2 on page 2-14).

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 6, *TMS320C62x/C64x Pipeline*, and Chapter 7, *TMS320C67x Pipeline.*

Example 1

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

Before LDW		**1 cycle after LDW**		**5 cycles after LDW**	
B1	0000 0000h	B1	0000 0000h	B1	21F3 1996h
A10	0000 0100h	A10	0000 0100h	A10	0000 0100h
mem 100h	21F3 1996h	mem 100h	21F3 1996h	mem 100h	21F3 1996h

Example 2 `LDB .D1 *–A5[4],A7`

	Before LDB	1 cycle after LDB	5 cycles after LDB
A5	0000 0204h	0000 0204h	0000 0204h
A7	1951 1970h	1951 1970h	FFFF FFE1h
AMR	0000 0000h	0000 0000h	0000 0000h
mem 200h	E1h	E1h	E1h

Example 3 `LDH .D1 *++A4[A1],A8`

	Before LDH	1 cycle after LDH	5 cycles after LDH
A1	0000 0002h	0000 0002h	0000 0002h
A4	0000 0020h	0000 0024h	0000 0024h
A8	1103 51FFh	1103 51FFh	FFFF A21Fh
AMR	0000 0000h	0000 0000h	0000 0000h
mem 24h	A21Fh	A21Fh	A21Fh

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

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

Example 5

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

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

LDB(U)/LDH(U)/LDW *Load From Memory With a 15-Bit Constant Offset*

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

31 29	28	27 23	22 8	7	6 4	3	2	1	0
creg	*z*	*dst*	*ucst15*	y	*ld/st*	1	1	*s*	*p*
3		5	15		3				

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
```

	Before LDB
B1	XXXX XXXXh
B14	0000 0100h
mem 124–127h	4E7A FF12h
mem 124h	12h

	1 cycle after LDB
B1	XXXX XXXXh
B14	0000 0100h
mem 124–127h	4E7A FF12h
mem 124h	12h

	5 cycles after LDB
B1	0000 0012h
B14	0000 0100h
mem 124–127h	4E7A FF12h
mem 124h	12h

LMBD — *Leftmost Bit Detection*

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`	`0000 0001h`
A2	`009E 3A81h`	`009E 3A81h`
A3	`XXXX XXXXh`	`0000 0008h`

MPY(U/US/SU) *Signed or Unsigned Integer Multiply 16lsb x 16lsb*

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

31 29	28	27 23	22 18	17 13	12	11 7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1/cst*	x	*op*	0	0	0	0	0	*s*	*p*
3		5	5	5		5							

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

MPYH(U/US/SU) *Signed or Unsigned Integer Multiply 16msb x 16msb*

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

31 29	28	27 23	22 18	17 13	12	11 7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1/cst*	x	*op*	0	0	0	0	0	*s*	*p*
3		5	5	5		5							

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
```

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

Example 2

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

	Before instruction			2 cycles after instruction	
A1	0023 0000h	35‡	A1	0023 0000h	
A2	FFA7 1234h	65447‡	A2	FFA7 1234h	
A3	XXXX XXXXh		A3	0022 F3D5h	2290645§

Example 3

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

	Before instruction			2 cycles after instruction	
A1	0023 0000h	35†	A1	0023 0000h	
A2	FFA7 FFFFh	65447‡	A2	FFA7 FFFFh	
A3	XXXX XXXXh		A3	0022 F3D5h	2290645

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

MPYHL(U)/MPYHULS/MPYHSLU *Signed or Unsigned Integer Multiply 16msb x 16lsb*

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

31 29	28	27 23	22 18	17 13	12	11 7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1/cst*	x	*op*	0	0	0	0	0	*s*	*p*
3		5	5	5		5							

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
```

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

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

MPYHL(U)/MPYHULS/MPYHSLU *Signed or Unsigned Integer Multiply 16lsb x 16msb*

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

31 29	28	27 23	22 18	17 13	12	11 7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1/cst*	x	*op*	0	0	0	0	0	*s*	*p*
3		5	5	5		5							

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
```

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

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

MV Move From Register to Register (Pseudo-Operation)

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** and **OR** instructions.

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. For the C64x, the operation performed is **OR** (.unit) FFFFh, *src2, dst.* In the case where *dst* is an slong, the C64x will use the **ADD** 0, *src, dst* operation like the C62x/C67x.

Execution

if (cond) 0 + *src* → dst
else nop

Instruction Type

Single-cycle

Delay Slots

0

MVC *Move Between the Control File and the Register File*

Syntax

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

.unit = .S2

Opcode

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

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
```

	Before instruction	1 cycle after instruction
B1	F009 0001h	F009 0001h
AMR	0000 0000h	0009 0001h

Note:

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

MVK — *Move a 16-Bit Signed Constant Into a Register and Sign Extend*

Syntax

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

.unit = .S1 or .S2

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

Opcode

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

Description

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

Execution

```
if (cond)    scst16 → dst
else         nop
```

Pipeline

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

Instruction Type

Single-cycle

Delay Slots

0

Note:

Use the MVK instruction to load 16-bit constants. The assembler will generate a warning for any constant over 16 bits. To load 32-bit constants, such as 0x 1234 5678, use the following pair of instructions:

```
MVKL   0x12345678
MVKH   0x12345678
```

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

```
MVKL   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

MVKH/MVKLH *Move 16-Bit Constant Into the Upper Bits of a Register*

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						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 ($dst_{15..0}$) → *dst*
else nop

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

Pipeline

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

Instruction Type Single-cycle

Delay Slots 0

Note:

Use the MVK instruction to load 16-bit constants. The assembler will generate a warning for any constant over 16 bits. To load 32-bit constants, such as 0x1234 5678, use the following pair of instructions:

```
MVKL   0x12345678
MVKH   0x12345678
```

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

```
MVKL   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

MVKL

Sign Extend 16-Bit Constant, Place In Register (Pseudo Operation)

Syntax

MVKL (.unit) *cst*, *dst*

.unit = .S1 or .S2

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

Opcode

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

Description

This is a pseudo-operation that sign extends the 16-bit constant and places it 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:

```
MVKL   0x12345678
MVKH   0x12345678
```

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

```
MVKL   label
MVKH   label
```

Example 1

```
MVKL .S1   293,A1
```

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

Example 2

```
MVKL .S2   125h,B1
```

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

Example 3

```
MVKL .S1   0FF12h,A1
```

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

NEG Negate (Pseudo-Operation)

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

NOP — *No Operation*

Syntax **NOP** [*count*]

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

Opcode

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

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

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

Execution No operation for *count* cycles

Instruction Type **NOP**

Delay Slots 0

Example 1

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

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

Example 2

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

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

NORM Normalize Integer

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

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

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

src2

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

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

src2

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

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

src2

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

Execution

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

Instruction Type Single-cycle

Pipeline

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

Delay Slots 0

Example 1

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

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

Example 2

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

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

NOT

Bitwise NOT (Pseudo-Operation)

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

OR Bitwise OR

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:

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					

.S unit form:

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

Description

A bitwise **OR** instruction is performed beween *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	A1	08A3 A49Fh
A2	XXXX XXXXh	A2	08FF B7DFh
B1	00FF 375Ah	B1	00FF 375Ah

Example 2

```
OR .L2     –12,B1,B2
```

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

SADD Integer Addition With Saturation to Result Size

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

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

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

SAT — *Saturate a 40-Bit Integer to a 32-Bit 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

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	1 0 0 0 0 0 0	1	1	0	*s*	*p*
3		5	5	5		7					

Description

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

Execution

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

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

SET — *Set a Bit Field*

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:

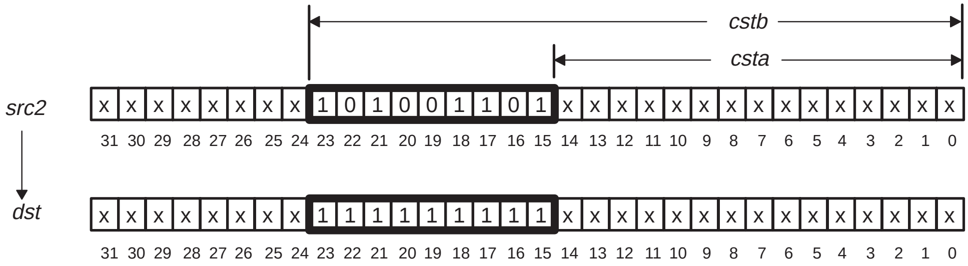
31 29	28	27 23	22 18	17 13	12 8	7 6	5					0
creg	*z*	*dst*	*src2*	*csta*	*cstb*	1 0	0	0	1	0	*s*	*p*
3	1	5	5	5	5	2					1	1

Register form:

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

SHL *Arithmetic Shift Left*

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

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

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 of *src1* are used by the shifter, so any bits set above bit 5 do not affect execution.

Execution

if (cond)	*src2* << *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

```
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

SHR — *Arithmetic Shift Right*

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

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

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 of *src1* are used by the shifter, so any bits set above bit 5 do not affect execution.

Execution

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

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
```

	Before instruction		1 cycle after instruction
A0	F123 63D1h	A0	F123 63D1h
A1	XXXX XXXXh	A1	FFF1 2363h

Example 2

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

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

Example 3

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

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

SHRU *Logical Shift Right*

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

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

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 of *src1* are used by the shifter, so any bits set above bit 5 do not affect execution.

Execution

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

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
```

	Before instruction		1 cycle after instruction
A0	F123 63D1h	A0	F123 63D1h
A1	XXXX XXXXh	A1	00F1 2363h

SMPY (HL/LH/H) *Integer Multiply With Left Shift and Saturation*

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

31 29	28	27 23	22 18	17 13	12	11 7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1/cst*	x	*op*	0	0	0	0	0	*s*	*p*
3		5	5	5		5							

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

SSHL *Shift Left With Saturation*

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

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

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.

Note:

For the C64x, when a register is used to specify the shift, the six least-significant bits specify the shift amount. Valid values are 0 through 63. If the shift count value is greater than 32, then the result is saturate to 32 bits when *src2* is non-zero.

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	A0	02E3 031Ch	A0	02E3 031Ch	
A1	XXXX XXXXh	A1	0B8C 0C70h	A1	0B8C 0C70h	
CSR	0001 0100h	CSR	0001 0100h	CSR	0001 0100h	Not saturated

Example 2

`SSHL .S1   A0,A1,A2`

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

SSUB — *Integer Subtraction With Saturation to Result Size*

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

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

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
```

STB/STH/STW *Store to Memory With a Register Offset or 5-Bit Unsigned Constant Offset*

Syntax

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

.unit = .D1 or .D2

Opcode

31 29	28	27 23	22 18	17 13	12 9	8	7	6 4	3	2	1	0
creg	*z*	*src*	*baseR*	*offsetR/ucst5*	*mode*	*r*	y	*ld/st*	0	1	*s*	*p*
3		5	5	5	4			3				

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 2.6.2 on page 2-14).

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 6, *TMS320C62x/C64x Pipeline,* and Chapter 7, *TMS320C67x Pipeline*.

Example 1

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

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

Example 2

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

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

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

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

Example 4 `STH .D1 A1,*A10––[A11]`

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

STB/STH/STW Store to Memory With a 15-Bit Offset

Syntax

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

.unit = .D2

Opcode

31 29	28	27 23	22 8	7	6 4	3	2	1	0
creg	*z*	*src*	*ucst15*	y	*ld/st*	1	1	*s*	*p*
3		5	15		3				

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]
```

	Before instruction	1 cycle after instruction	3 cycles after instruction
B1	1234 5678h	1234 5678h	1234 5678h
B14	0000 1000h	0000 1000h	0000 1000h
mem 1028h	42h	42h	78h

SUB(U) *Signed or Unsigned Integer Subtraction Without Saturation*

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	0100100	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:

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					

.S unit form:

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

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:

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

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

SUBAB/SUBAH/SUBAW *Integer Subtraction Using Addressing Mode*

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

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

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 2.6.2 on page 2-14). *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
```

SUBC — *Conditional Integer Subtract and Shift – Used for Division*

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

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

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 → dst
             else $src1$ << 1 → 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  |
```

SUB2 — *Two 16-Bit Integer Subtractions on Upper and Lower Register Halves*

Syntax

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

.unit = .S1 or .S2

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

Opcode

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

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

XOR — *Exclusive OR*

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:

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					

.S unit form:

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

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	A1	0721 325Ah
A2	0019 0F12h	A2	0019 0F12h
A3	XXXX XXXXh	A3	0738 3D48h

Example 2

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

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

ZERO

Zero a Register (Pseudo-Operation)

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*. For the C64x, the operation performed is **MVK** 0, *dst.* In the case where *dst* is an slong, the C64x will use the **SUB** operation like the C62x/C67x.

Execution

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

Instruction Type

Single-cycle

Delay Slots

0

Chapter 4

TMS320C67x Floating-Point Instruction Set

The TMS320C67x™ floating-point DSP uses all of the instructions available to the TMS320C62x™, 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

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
→	Assignment
+	Addition
×	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	
DPTRUNC		CMPGTDP	
INTDP		CMPGTSP	
INTDPU		CMPLTDP	
INTSP		CMPLTSP	
INTSPU		RCPDP	
SPINT		RCPSP	
SPTRUNC		RSQRDP	
SUBDP		RSQRSP	
SUBSP		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)

	C67x Functional Units				
Instruction	**.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.

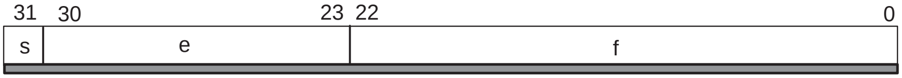
Table 4–4 shows notations used in discussing floating-point numbers.

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

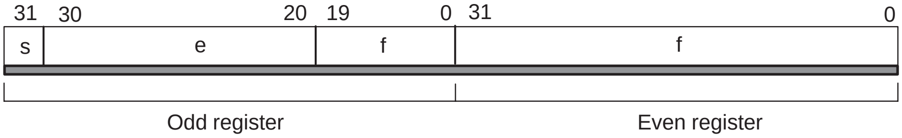
Table 4–6 shows hex and decimal values for some single-precision floating-point numbers.

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)

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

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

ABSDP *Double-Precision Floating-Point Absolute Value*

Syntax

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

.unit = .S1 or .S2

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

Opcode

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

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*) → *dst*
else nop

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

1) If *src2* ≥ 0, then *src2* → *dst*
2) If *src2* < 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	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
```

	Before instruction				**2 cycles after instruction**		
A1:A0	c004 0000h	0000 0000h	–2.5	A1:A0	c004 0000h	0000 0000h	–2.5
A3:A2	XXXX XXXXh	XXXX XXXXh		A3:A2	4004 0000h	0000 0000h	2.5

ABSSP — *Single-Precision Floating-Point Absolute Value*

Syntax

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

.unit = . S1 or .S2

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

Opcode

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

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* ≥ 0, then *src2* → *dst*
2) If *src2* < 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

ADDAD

Integer Addition Using Doubleword Addressing Mode

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

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

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 2.6.2 on page 2-14). *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) → *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

ADDDP *Double-Precision Floating-Point Addition*

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

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

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

ADDSP *Single-Precision Floating-Point Addition*

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

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

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 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

CMPEQDP *Double-Precision Floating-Point Compare for Equality*

Syntax

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

.unit = .S1 or .S2

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

Opcode

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 0 1 0 0 0	1	0	0	0	*s*	*p*
3		5	5	5		6						

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`

	Before instruction				2 cycles after instruction		
A1:A0	4021 3333h	3333 3333h	8.6	A1:A0	4021 3333h	3333 3333h	8.6
A3:A2	c004 0000h	0000 0000h	–2.5	A3:A2	c004 0000h	0000 0000h	–2.5
A4	XXXX XXXXh			A4	0000 0000h	false	

CMPEQSP *Single-Precision Floating-Point Compare for Equality*

Syntax

CMPEQSP (.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

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 0 0	1	0	0	0	*s*	*p*
3		5	5	5		6						

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			Configuration Register	
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 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

CMPGTDP *Double-Precision Floating-Point Compare for Greater Than*

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

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 0 1 0 0 1	1	0	0	0	*s*	*p*
3		5	5	5		6						

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			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	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`

	Before instruction				2 cycles after instruction		
A1:A0	4021 3333h	3333 3333h	8.6	A1:A0	4021 3333h	3333 3333h	8.6
A3:A2	c004 0000h	0000 0000h	–2.5	A3:A2	c004 0000h	0000 0000h	–2.5
A4	XXXX XXXXh			A4	0000 0001h	true	

CMPGTSP *Single-Precision Floating-Point Compare for Greater Than*

Syntax **CMPGTSP** (.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

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 0 1	1	0	0	0	*s*	*p*
3		5	5	5		6						

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			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	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

CMPLTDP *Double-Precision Floating-Point Compare for Less Than*

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

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 0 1 0 1 0	1	0	0	0	*s*	*p*
3		5	5	5		6						

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	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				2 cycles after instruction		
A1:A0	4021 3333h	3333 3333h	8.6	A1:A0	4021 3333h	4021 3333h	8.6
B3:B2	c004 0000h	0000 0000h	–2.5	B3:B2	c004 0000h	0000 0000h	–2.5
A4	XXXX XXXXh			A4	0000 0000h	false	

CMPLTSP *Single-Precision Floating-Point Compare for Less Than*

Syntax

CMPLTSP (.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

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 0	1	0	0	0	*s*	*p*
3		5	5	5		6						

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 $\rightarrow$ dst
              else 0 $\rightarrow$ 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

DPINT *Convert Double-Precision Floating-Point Value to Integer*

Syntax

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

.unit = .L1 or .L2

Opcode map field used...	For operand type...	Unit
src2 *dst*	dp 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 0 0	1	1	0	*s*	*p*
3		5	5	5		7					

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
```

	Before instruction				4 cycles after instruction		
A1:A0	4021 3333h	3333 3333h	8.6	A1:A0	4021 3333h	3333 3333h	8.6
A4	XXXX XXXXh			A4	0000 0009h		9

DPSP

Convert Double-Precision Floating-Point Value to Single-Precision Floating-Point Value

Syntax

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

.unit = .L1 or .L2

Opcode map field used...	For operand type...	Unit
src2 *dst*	dp sp	.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 0 1	1	1	0	*s*	*p*
3		5	5	5		7					

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
```

	Before instruction				4 cycles after instruction		
A1:A0	4021 3333h	3333 3333h	8.6	A1:A0	4021 3333h	4021 3333h	8.6
A4	XXXX XXXXh			A4	4109 999Ah		8.6

DPTRUNC *Convert Double-Precision Floating-Point Value to Integer With Truncation*

Syntax

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

.unit = .L1 or .L2

Opcode map field used...	For operand type...	Unit
src2 *dst*	dp 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 0 0 0 1	1	1	0	*s*	*p*
3		5	5	5		7					

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				4 cycles after instruction		
A1:A0	4021 3333h	3333 3333h	8.6	A1:A0	4021 3333h	3333 3333h	8.6
A4	XXXX XXXXh			A4	0000 0008h		8

INTDP(U) *Convert Integer to Double-Precision Floating-Point Value*

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

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 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

B4	1965 1127h		426053927
A1:A0	XXXX XXXXh	XXXX XXXXh	

5 cycles after instruction

B4	1965 1127h		426053927
A1:A0	41B9 6511h	2700 0000h	4.2605393 E08

Example 2

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

Before instruction

A4	FFFF FFDEh		4294967262
A1:A0	XXXX XXXXh	XXXX XXXXh	

5 cycles after instruction

A4	FFFF FFDEh		4294967262
A1:A0	41EF FFFFh	FBC0 0000h	4.2949673 E09

INTSP(U) *Convert Integer to Single-Precision Floating-Point Value*

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

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 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
```

LDDW

Load Doubleword From Memory With an Unsigned Constant Offset or Register Offset

Syntax

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

.unit = .D1 or .D2

Opcode

31 29	28	27 23	22 18	17 13	12 9	8	7	6 4	3	2	1	0
creg	*z*	*dst*	*baseR*	*offsetR/ucst5*	*mode*	*r*	y	*ld/st*	0	1	*s*	*p*
3		5	5	5	4			3				

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 2.6.2 on page 2-14).

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
```

Before instruction

A1:A0	XXXX XXXXh	XXXX XXXXh	
B10	0000 0010h		16
mem 0x18	3333 3333h	4021 3333h	8.6

5 cycles after instruction

A1:A0	4021 3333h	3333 3333h	8.6
B10	0000 0010h		16
mem 0x18	3333 3333h	4021 3333h	8.6

Little-endian mode

Example 2

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

Before instruction

A1:A0	XXXX XXXXh	XXXX XXXXh	
A10	0000 0010h		16
mem 0x18	4021 3333h	3333 3333h	8.6

1 cycle after instruction

A1:A0	XXXX XXXXh	XXXX XXXXh	
A10	0000 0018h		24
mem 0x18	4021 3333h	3333 3333h	8.6

5 cycles after instruction

A1:A0	4021 3333h	3333 3333h	8.6
A10	0000 0018h		24
mem 0x18	4021 3333h	3333 3333h	8.6

Big-endian mode

MPYDP *Double-Precision Floating-Point Multiply*

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

31 29	28	27 23	22 18	17 13	12	11 7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	0 1 1 1 0	0	0	0	0	0	*s*	*p*
3		5	5	5		5							

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

Execution

if (cond) *src1* × *src2* → *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				10 cycles after instruction		
A1:A0	4021 3333h	3333 3333h	8.6	A1:A0	4021 3333h	4021 3333h	8.6
A3:A2	C004 0000h	0000 0000	–2.5	A3:A2	C004 0000h	0000 0000h	–2.5
A5:A4	XXXX XXXXh	XXXX XXXXh		A5:A4	C035 8000h	0000 0000h	–21.5

MPYI — *32-Bit Integer Multiply – Result Is Lower 32 Bits*

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

31 29	28	27 23	22 18	17 13	12	11 7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1/cst*	x	*op*	0	0	0	0	0	*s*	*p*
3		5	5	5		5							

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	A1	0034 5678h	3430008
B2	0011 2765h	1124197	B2	0011 2765h	1124197
A3	XXXX XXXXh		A3	CBCA 6558h	–875928232

MPYID *32-Bit Integer Multiply – Result Is 64 Bits*

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

31 29	28	27 23	22 18	17 13	12	11 7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1/cst*	x	*op*	0	0	0	0	0	*s*	*p*
3		5	5	5		5							

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

MPYSP

Single-Precision Floating-Point Multiply

Syntax

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

.unit = .M1 or .M2

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

Opcode

31 29	28	27 23	22 18	17 13	12	11 7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	1 1 1 0 0	0	0	0	0	0	*s*	*p*
3		5	5	5		5							

Description

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

Execution

```
if (cond)    src1 × src2 → 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
```

	Before instruction			4 cycles 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	C1AC 0000h	–21.5

RCPDP *Double-Precision Floating-Point Reciprocal Approximation*

Syntax

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

.unit = .S1 or .S2

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

Opcode

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

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 $2^{1022} < \mathit{src2} <$ 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
```

	Before instruction			2 cycles after instruction		
A1:A0	4010 0000h	0000 0000h	A1:A0	4010 0000h	0000 0000h	4.00
A3:A2	XXXX XXXXh	XXXX XXXXh	A3:A2	3FD0 0000h	0000 0000h	0.25

RCPSP *Single-Precision Floating-Point Reciprocal Approximation*

Syntax

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

.unit = .S1 or .S2

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

Opcode

31 29	28	27 23	22 18	17 13	12	11 6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	00000	x	1 1 1 1 0 1	1	0	0	0	*s*	*p*
3		5	5	5		6						

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

```
RCPSP .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

RSQRDP *Double-Precision Floating-Point Square-Root Reciprocal Approximation*

Syntax

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

.unit = .S1 or .S2

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

Opcode

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

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

RSQRSP *Single-Precision Floating-Point Square-Root Reciprocal Approximation*

Syntax

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

.unit = .S1 or .S2

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

Opcode

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

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

SPDP

Convert Single-Precision Floating-Point Value to a Double-Precision Floating-Point Value

Syntax

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

.unit = .S1 or .S2

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

Opcode

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

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
```

	Before instruction				2 cycles after instruction		
B2	4109 999Ah		8.6	B2	4109 999Ah		8.6
A1:A0	XXXX XXXXh	XXXX XXXXh		A1:A0	4021 3333h	4000 0000h	8.6

SPINT *Convert Single-Precision Floating-Point Value to Integer*

Syntax

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

.unit = .L1 or .L2

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

SPTRUNC *Convert Single-Precision Floating-Point Value to Integer With Truncation*

Syntax

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

.unit = .L1 or .L2

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 1	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*. 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
```

SUBDP — *Double-Precision Floating-Point Subtract*

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

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

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	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

SUBSP *Single-Precision Floating-Point Subtract*

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

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

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
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

TMS320C64x Fixed-Point Instruction Set

The TMS320C64x™ fixed-point DSP uses all of the instructions available to the TMS320C62x™, but it also uses other instructions that are specific to the C64x™. These specific instructions include 8-bit and 16-bit extensions, nonaligned word loads and stores, data packing/unpacking operations. This chapter describes these C64x-specific instructions.

Instructions that are common to both the C62x™ and C64x are described in Chapter 3.

Topic **Page**

5.1 Instruction Operation and Execution Notations

Table 5–1 explains the symbols used in the new instruction descriptions.

Table 5–1. New Instruction Operation and Execution Notations

Symbol	Meaning
abs(x)	Absolute value of x
and	Bitwise AND
b_i	Select bit i of source/destination b
bit_count	Count the number of bits that are 1 in a specified byte
bit_reverse	Reverse the order of bits in a 32-bit register
byte0	8-bit value in the least significant byte position in 32-bit register (bits 0-7)
byte1	8 -bit value in the next to least significant byte position in 32-bit register (bits 8-15)
byte2	8-bit value in the next to most signficant byte position in 32-bit register (bits 16-23)
byte3	8-bit value in the most signficant byte position in 32-bit register (bits 24-31)
bv2	Bit Vector of two flags for s2 or u2 data type
bv4	Bit Vector of four flags for s4 or u4 data type
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)
dst_h or *dst_o*	msb32 of *dst* (placed in odd-numbered register of 64-bit register pair)
dst_l or *dst_e*	lsb32 of *dst* (placed in even-numbered register of a 64-bit register pair)
dws4	Four packed signed 16-bit integers in a 64-bit register pair
dwu4	Four packed unsigned 16-bit integers in a 64-bit register pair
gmpy	Galois Field Multiply
i2	Two packed 16-bit integers in a single 32-bit register
i4	Four packed 8-bit integers in a single 32-bit register
int	32-bit integer value
lsbn or LSBn	n least significant bits (for example, lsb16)

Table 5–1. New Instruction Operation and Execution Notations (Continued)

Symbol	Meaning
msbn or MSBn	n most significant bits (for example, msb16)
nop	No operation
or	Bitwise OR
R	Any general-purpose register
ROTL	Rotate left
sat	Saturate
sbyte0	Signed 8-bit value in the least significant byte position in 32-bit register (bits 0–7)
sbyte1	Signed 8-bit value in the next to least significant byte position in 32-bit register (bits 8–15)
sbyte2	Signed 8-bit value in the next to most significant byte position in 32-bit register (bits 16–23)
sbyte3	Signed 8-bit value in the most significant byte position in 32-bit register (bits 24–31)
scstn	Signed n-bit constant field (for example, scst7)
se	Sign-extend
sint	Signed 32-bit integer value
slsb16	Signed 16-bit integer value in lower half of 32-bit register
smsb16	Signed 16-bit integer value in upper half of 32-bit register
s2	Two packed signed 16-bit integers in a single 32-bit register
s4	Four packed signed 8-bit integers in a single 32-bit register
sllong	Signed 64-bit integer value
ubyte0	Unsigned 8-bit value in the least significant byte position in 32-bit register (bits 0–7)
ubyte1	Unsigned 8-bit value in the next to least significant byte position in 32-bit register (bits 8–15)
ubyte2	Unsigned 8-bit value in the next to most significant byte position in 32-bit register (bits 16–23)
ubyte3	Unsigned 8-bit value in the most significant byte position in 32-bit register (bits 24–31)
ucstn	n-bit unsigned constant field (for example, ucstn5)
uint	Unsigned 32-bit integer value

Table 5–1. New Instruction Operation and Execution Notations (Continued)

Symbol	Meaning
ullong	Unsigned 64-bit integer value
ulsb16	Unsigned 16-bit integer value in lower half of 32-bit register
umsb16	Unsigned 16-bit integer value in upper half of 32-bit register
u2	Two packed unsigned 16-bit integers in a single 32-bit register
u4	Four packed unsigned 8-bit integers in a single 32-bit register
xsint	Signed 32-bit integer value that can optionally use cross path
xs2	Two packed signed 16-bit integers in a single 32-bit register that can optionally use cross path
xs4	Four packed signed 8-bit integers in a single 32-bit register that can optionally use cross path
xuint	Unsigned 32-bit integer value that can optionally use cross path
xu2	Two packed unsigned 16-bit integers in a single 32-bit register that can optionally use cross path
xu4	Four packed unsigned 8-bit integers in a single 32-bit register that can optionally use cross path
→	Assignment
+	Addition
++	Increment by one
x	Multiplication
–	Subtraction
>	Greater than
<	Less than
<<	Shift left
>>	Shift right
>=	Greater than or equal to
<=	Less than or equal to
==	Equal to
~	Logical Inverse
&	Logical And

5.2 Mapping Between Instructions and Functional Units

Table 5–2 shows the mapping between instructions and functional units. Table 5–3 shows the mapping between functional units and instructions.

Table 5–2. Instruction to Functional Unit Mapping

.L unit	.M unit		.S unit		.D unit
ABS2	AVG2	SHFL	ADD2	SUB2	ADD2
ADD2	AVGU4	SMPY2	ADDKPC	SWAP2	ADDAD
ADD4	BITC4	SSHVL	AND	UNPKHU4	AND
AND	BITR	SSHVR	ANDN	UNPKLU4	ANDN
ANDN	DEAL	XPND2	BDEC	XOR	LDDW
MAX2	DOTP2	XPND4	BNOP		LDNDW
MAXU4	DOTPN2		BPOS		LDNW
MIN2	DOTPNRSU2		CMPEQ2		MVK
MINU4	DOTPNRUS2		CMPEQ4		OR
MVK	DOTPRSU2		CMPGT2		STDW
OR	DOTPRUS2		CMPGTU4		STNDW
PACK2	DOTPSU4		CMPLT2		STNW
PACKH2	DOTPUS4		CMPLTU4		SUB2
PACKH4	DOTPU4		MVK		XOR
PACKHL2	GMPY4		OR		
PACKL4	MPY2		PACK2		
PACKLH2	MPYHI		PACKH2		
SHLMB	MPYIH		PACKHL2		
SHRMB	MPYHIR		PACKLH2		
SUB2	MPYIHR		SADD2		
SUB4	MPYLI		SADDU4		
SUBABS4	MPYIL		SADDSU2		
SWAP2	MPYLIR		SADDUS2		
SWAP4	MPYILR		SHLMB		
UNPKHU4	MPYSU4		SHR2		
UNPKLU4	MPYUS4		SHRMB		
XOR	MPYU4		SHRU2		
	MVD		SPACK2		
	ROTL		SPACKU4		

Table 5–3. Functional Unit to Instruction Mapping

Instruction	.L unit	.M unit	.S unit	.D unit
ABS2	√			
ADD2ψ	√		√	√
ADD4	√			
ADDAD				√
ADDKPC			√	
ANDψ	√		√	√
ANDN	√		√	√
AVG2		√		
AVGU4		√		
BDEC			√	
BITC4		√		
BITR		√		
BNOP			√	
BNOP reg			√	
BPOS			√	
CMPEQ2			√	
CMPEQ4			√	
CMPGT2			√	
CMPGTU4			√	
CMPLT2			√	
CMPLTU4			√	
DEAL		√		
DOTP2		√		
DOTPN2		√		
DOTPNRSU2		√		
DOTPNRUS2		√		
DOTPRSU2		√		
DOTPRUS2		√		
DOTPSU4		√		

Table 5–3. Functional Unit to Instruction Mapping (Continued)

Instruction	.L unit	.M unit	.S unit	.D unit
DOTPUS4		√		
DOTPU4		√		
GMPY4		√		
LDDW				√
LDNDW				√
LDNW				√
MAX2	√			
MAXU4	√			
MIN2	√			
MINU4	√			
MPY2		√		
MPYHI		√		
MPYIH		√		
MPYHIR		√		
MPYIHR		√		
MPYLI		√		
MPYIL		√		
MPYLIR		√		
MPYILR		√		
MPYSU4		√		
MPYUS4		√		
MPYU4		√		
MVD		√		
MVKΨ	√		√	√
ORΨ	√		√	√
PACK2	√		√	
PACKH2	√		√	
PACKH4	√			
PACKHL2	√		√	

Table 5–3. Functional Unit to Instruction Mapping (Continued)

Instruction	.L unit	.M unit	.S unit	.D unit
PACKL4	√			
PACKLH2	√		√	
ROTL		√		
SADD2			√	
SADDSU2			√	
SADDU4			√	
SADDUS2			√	
SHFL		√		
SHLMB	√		√	
SHR2			√	
SHRMB	√		√	
SHRU2			√	
SMPY2		√		
SPACK2			√	
SPACKU4			√	
SSHVL		√		
SSHVR		√		
STDW				√
STNDW				√
STNW				√
SUB2ψ	√		√	√
SUB4	√			
SUBABS4	√			
SWAP2	√		√	
SWAP4	√			
UNPKHU4	√		√	
UNPKLU4	√		√	
XORψ	√		√	√

Table 5–3. Functional Unit to Instruction Mapping (Continued)

Instruction	.L unit	.M unit	.S unit	.D unit
XPND2		√		
XPND4		√		

Note: Ψ indicates instructions that exist on C62x/C67x but are now also available on one or more additional functional units.

5.3 TMS320C64x Opcode Map Symbols

Table 5–4 and the instruction descriptions in this chapter explain the field syntaxes and values.

Table 5–4. TMS320C64x 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
sc	scaling mode bit
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

5.4 Delay Slots

The execution of the additional 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 **CMPGT2**), source operands read in cycle i produce a result that can be read in cycle i + 1. For a two-cycle instruction (such as **AVGU4**), source operands read in cycle i produce a result that can be read in cycle i + 2. For a four-cycle instruction (such as **DOTP2**), source operands read in cycle i produce a result that can be read in cycle i + 4. Table 5–5 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 additional instructions have a functional unit latency of 1. This means that a new instruction can be started on the functional unit every cycle. Single-cycle throughput is another term for single-cycle functional unit latency.

Table 5–5. 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	
Two cycle	1	1	i	i + 1	
Multiply (16x16)	1	1	i	i + 1	
Four cycle	3	1	i	i + 3	
Load	4	1	i	i, i + 4[‡]	
Branch	5	1	i[§]		i + 5

[†] Cycle i is in the E1 pipeline phase.
[‡] For loads, any address modification happens in cycle i. The loaded data is written into the register file in cycle i + 4.
[§] The branch to label, branch to IRP, and branch to NRP instructions do not read any general-purpose registers.

5.5 Conditional Operations

The C64x handles conditional operations in the same manner as the TMS320C62x™/TMS320C67x™. In addition, the C64x can use A0 as a conditional register as well as A1,A2, B0, B1 and B2. The creg field is encoded in the instruction opcode as shown in Table 5–6.

Table 5–6. 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
A0		1	1	0	z
Reserved		1	1	x	x

† This value is reserved for software breakpoints that are used for emulation purposes.

5.6 Resource Constraints

This section will cover the differences between the C62x and the C64x with respect to resource constraints.

If no differences are cited, then the same constraints covered in Chapter 3, section 3.7 apply.

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

Units in one data path may read a single operand from the opposite data path by using a cross path (**1X** and **2X**). The C62x/C67x™ architectures provided cross path access to the .L, .S and .M units. One functional unit per data path per execute packet may read an operand via a cross path on these architectures. The C64x adds data cross path access to the .D unit for arithmetic and logical operations. Also, multiple units in a given path may read an operand via the same cross path on the C64x, provided that each unit is reading the same operand.

For example, the .S1 unit can read both its operands from the A register file; or it can read an operand from the B register file using the 1X cross path and the other from the A register file. The use of a cross path is denoted by an X following the functional unit name in the instruction syntax (as in S1X).

The following execute packet is invalid because the 1X cross path is being used for two different B register operands:

```
  MV .S1X B0, A0 ; \ Invalid.  Instructions are using the 1X cross path
||MV .L1X B1, A1 ; / with different B registers
```

The following execute packet is valid because all uses of the 1X cross path are for the same B register operand, and all uses of the 2X cross path are for the same A register operand:

```
   ADD .L1X A0,B1,A1 ; \ Instructions use the 1X with B1
|| SUB .S1X A2,B1,A2 ; / 1X cross paths using B1
|| AND .D1 A4,A1,A3 ;
|| MPY .M1 A6,A1,A4 ;
|| ADD .L2 B0,B4,B2 ;
|| SUB .S2X B4,A4,B3 ; / 2X cross paths using A4
|| AND .D2X B5,A4,B4 ; / 2X cross paths using A4
|| MPY .M2 B6,B4,B5  ;
```

As in the C62x core, when an operand comes from a register file opposite of the destination register, the x bit in the instruction field is set.

5.6.2 Cross Path Stalls

The C64x introduces a delay clock cycle whenever an instruction attempts to read a register via a cross path that was updated in the previous cycle. This is known as a cross path stall. This stall is inserted automatically by the hardware, no **NOP** instruction is needed. It should be noted that no stall is introduced if the register being read has data placed by a load instruction, or if an instruction reads a result one cycle after the result is generated.

Here are some examples:

```
ADD  .S1   A0, A0, A1; / Stall is introduced; A1 is  updated
                     ;   1 cycle before it is used as a cross path
ADD  .S2X  A1, B0, B1; \ source


ADD  .S1   A0, A0, A1 ; / No stall is introduced; A0 not updated
                      ;   1 cycle before it is used as a cross
ADD  .S2X  A0, B0, B1 ; \ path source


LDW  .D1  *++A0[1], A1; / No stall is introduced; A1 is the load
                      ;   destination
NOP   4               ;   NOP 4 represents 4 instructions to
ADD  .S2X  A1, B0, B1 ; \ be executed between the load and add.


LDW  .D1  *++A0[1], A1 ; / Stall is introduced; A0 is updated
ADD  .S2X  A0, B0, B1  ;   1 cycle  before it is used as a cross  .
                       ; \ path source
```

This cross path stall does not occur on the C62x/C67x. However, all code written for the C62x/C67x that contains cross paths used in the manner above and that runs on the C64x will exhibit this behavior. The code will still run correctly but it will take more clock cycles.

It is possible to avoid the cross path stall by scheduling an instruction that reads an operand via the cross path at least one cycle after the operand is updated. With appropriate scheduling, the C64x can provide one cross path operand per data path per cycle with no stalls. In many cases, the TMS320C6000 Optimizing C Compiler and Assembly Optimizer automatically perform this scheduling.

5.6.3 Constraints on Loads and Stores

The data address paths named DA1 and DA2 are each connected to the .D units in both data paths. Load/store instructions can use an address register 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 DA1 and DA2 resources and their associated data paths are specified as T1 and T2, respectively. T1 consists of the DA1 address path and the LD1a, LD1b, ST1a, and ST1b data paths. Similarly, T2 consists of the DA2 address path and the LD2a, LD2b, ST2a, and ST2b data paths. The T1 and T2 designations appear in functional unit fields for load and store instructions.

The C64x can access words and double words at any byte boundary using non-aligned loads and stores. As a result, word and double word data does not need alignment to 32-bit or 64-bit boundaries. No other memory access may be used in parallel with a non-aligned memory access. The other .D unit can be used in parallel, as long as it is not performing a memory access.

The following execute packet is invalid:

```
   LDNW .D2T2 *B2[B12],B13 ; \ Two memory operations,
|| LDB  .D1T1 *A2,A14      ; / one non-aligned
```

The following execute packet is valid:

```
    LDNW .D2T2 *B2[B12], A13; \ One non-aligned memory
                            ;   operation,
||  ADD  .D1x A12, B13, A14 ;   one non-memory .D unit
                            ; / operation
```

5.6.4 Constraints on Long (40-Bit) Data

Some of the ports for long and double word operands are shared between functional units. This creates constraints on when long or double word operations can be scheduled in a data path. The restrictions are different between the C64x, the C62x, and the C67x. These differences are summarized in Table 5–7.

Table 5–7. Constraint Differences Between C62x/C67x and C64x Registers

Situation	C62x	C67x	C64x
Both .S and .L units on the same side use a long source.	Conflict	Conflict	OK
Both .S and .L units on the same side write a long result.	Conflict	Conflict	OK
A store reads the data to store from the same side that the .S unit reads a long source.	Conflict	Conflict	Conflict
A store (other than STDW) reads the data to store from the same side that the .L unit reads a long source.	Conflict	Conflict	OK
An STDW reads the data to store from the same side that the .L unit reads a long source.	N/A	N/A	Conflict
An LDDW writes back data (loaded from memory) to the same side that the .S unit writes a long result.	N/A	Conflict	Conflict
An LDDW writes back data (loaded from memory) to the same side that the .L unit writes a long result.	N/A	Conflict	OK

Legend: Conflict = Conflict exists
OK = No conflict exists
N/A = Not Applicable; instruction not available to core

On the C62x and C67x, only one long result may be issued per register file in an execute packet. On the C64x, the .L, .S and .D units can operate independently, as long as the .D unit does loads of data smaller than double words. Up to two instructions with long results may be issued per side in an execute packet. Double word load instructions conflict with long results from the .S units. All stores conflict with a long source on the .S unit. Only double word stores conflict with a long source on the .L unit.

The following execute packet is invalid on the C62x/C67x and the C64x cores, because the .D unit store on the T1 path conflicts with the long source on the .S1 unit:

```
   ADD .S1   A1,A5:A4, A3:A2; \ Long source on .S unit and a
|| STW .D1T1 A8,*A9;        / store on the T1 path of the .D unit
```

The following code sequence is invalid on the C64x and C67x cores. This example is not applicable to the C62x core because it uses the **LDDW** instruction that is not supported on the C62x core.

```
LDDW .D1T1 *A16,A11:A10 ; \ Double word load written to
                        ;   A11:A10 on .D1
NOP 3                   ;   conflicts after 3 cycles
SHL  .S1   A8,A9,A7:A6  ; / with write to A7:A6 on .S1
```

The following code sequences are invalid on the C64x core. These examples are not applicable to the C62x and C67x cores, because the **STDW** instruction is not supported on the C62x or C67x cores.

```
   ADD  .L1   A1,A5:A4,A3:A2  ; \ Long source on .L1 conflicts
|| STDW .D1T1 A13:A12,*A16    ;   with double word store on the
                              ; / T1 path of .D1

   SHL  .S1   A5:A4,A1,A3:A2 ; \  Long source on .S1 conflicts
|| STDW .D1T1 A9:A8,*A19     ;     with double word store on the
                             ;  /  T1 path of .D1
```

The following code sequences are invalid on the C62x and C67x cores but valid on the C64x core:

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

   LDW .D1T1 *A13,A11   ; \ Word read on T1 path of .D1 doesn’t
   NOP 3                ; / conflict after 3 cycles with long
   ADD .L1 A4,A1,A3:A2  ; \ write on .L1 unit and long write
|| SHL .S1 A8,A2,A7:A6  ; / on .S1 unit

   ADD .L1   A1,A5:A4,A3:A2 ; \ Long source operand on .L1
|| STW .D1T1 A8,*A9         ;   doesn’t conflict with store
                            ; / word on the T1 path of .D1
```

The following execute packets are valid on all cores:

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

   ADD .L1    A4, A1, A3:A2  ; \ No long read with
|| STW .D1T1  A8,*A9         ; / the store on T1 path of .D1
```

The following execute packet is invalid on all cores:

```
   SHR .S1   A5:A4, A1, A3:A2 ; \ No long read on .S1 with
|| STW .D1T1  A8,*A9          ; / the store on T1 path of .D1
```

5.7 Addressing Modes

The addressing modes on the C6000™ DSPs 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, LDNDW, LDNW, STNDW, STNW, LDDW, STDW, 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. There is no **SUBAD** instruction.

5.7.1 Linear Addressing Mode

5.7.1.1 LD/ST Instructions

For load and store instructions, linear mode simply shifts the *offsetR/cst* operand to the left by 3, 2, 1, or 0 for double word, word, halfword, or byte access, respectively; and then performs an add or a subtract to baseR (depending on the operation specified). The **LDNDW** and **STNDW** instructions also support non-scaled offsets. In non-scaled mode, the *offsetR/cst* is not shifted before adding or subtracting from the baseR.

For the pre-increment, pre-decrement, positive offset, and negative offset address generation options, the result of the calculation is the address to be accessed in memory. For post-increment or post-decrement addressing, the value of baseR before the addition or subtraction is the address to be accessed from memory.

5.7.1.2 ADDA/SUBA Instructions

For integer addition and subtraction instructions, linear mode simply shifts the *src1*/cst operand to the left by 3, 2, 1, or 0 for double word, word, halfword, or byte data sizes, respectively, and then performs the add or subtract specified.

5.7.2 Circular Addressing Mode

The BK0 and BK1 fields in the AMR specify block sizes for circular addressing. See the description of the AMR (defined in Chapter 2) for more information.

5.7.2.1 LD/ST Instructions

As with linear address arithmetic, offsetR/cst is shifted left by 3, 2, 1, or 0 according to the data size, and is then added to or subtracted from baseR to produce the final address. Circular addressing modifies this slightly by only allowing bits N through 0 of the result to be updated, leaving bits 31 through N+1 unchanged after address arithmetic. The resulting address is bounded to $2^{(N+1)}$ range, regardless of the size of the *offsetR/cst*.

The circular buffer size specified in the AMR is not scaled. For example, a block-size of 8 is 8 bytes, not 8 times the 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 5–1 shows an LDW performed with register A4 in circular mode and BK0 = 4, so the buffer size is 32 bytes, 16 half words, or 8 words. The value put in the AMR for this example is 00040001h.

Example 5–1. LDW in Circular Mode

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

	Before LDW	1 cycle after LDW	5 cycles after LDW
A4	0000 0100h	0000 0104h	0000 0104h
A1	XXXX XXXXh	XXXX XXXXh	1234 5678h
mem 104h	1234 5678h	1234 5678h	1234 5678h

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).

5.7.2.2 ADDA/SUBA Instructions

As with linear address arithmetic, *offsetR/cst* is shifted left by 3, 2, 1, or 0 according to the data size, and is then added to or subtracted from baseR to produce the final address. Circular addressing modifies this slightly by only allowing bits N through 0 of the result to be updated, leaving bits 31 through N+1 unchanged after address arithmetic. The resulting address is bounded to $2^{(N+1)}$ range, regardless of the size of the *offsetR/cst*.

The circular buffer size in the AMR is not scaled. For example, a block size of 8 is 8 bytes, not 8 times the 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 5–2 shows an **ADDAH** performed with register A4 in circular mode and BK0=4, so the buffer size is 32 bytes, 16 half words, or 8 words. The value put in the AMR for this example is 00040001h.

Example 5–2. ADDAH in Circular Mode

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

	Before ADDAH		1 cycle after ADDAH
A4	0000 0100h	A4	0000 0106h
A1	0000 0013h	A1	0000 0013h

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).

5.7.2.3 Circular Addressing Considerations with Non-Aligned Memory

Circular addressing may be used with non-aligned accesses. When circular addressing is enabled, address updates and memory accesses occur in the same manner as for the equivalent sequence of byte accesses. The only restriction is that the circular buffer size be at least as large as the data size being accessed. Non-aligned access to circular buffers that are smaller than the data being read will cause undefined results.

Non-aligned accesses to a circular buffer apply the circular addressing calculation to *logically adjacent* memory addresses. The result is that non-aligned accesses near the boundary of a circular buffer will correctly read data from both ends of the circular buffer, thus seamlessly causing the circular buffer to “wrap around” at the edges.

Consider, for example, a circular buffer size of 16 bytes. A circular buffer of this size at location 0x20, would look like this in physical memory:

1	1	1	1	1	1	1	1	1	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	3	3	3	3	3	3	3	3	3
7	8	9	A	B	C	D	E	F	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0	1	2	3	4	5	6	7	8
x	x	x	x	x	x	x	x	x	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	x	x	x	x	x	x	x	x	x

The effect of circular buffering is to make it so that memory accesses and address updates in the 0x20–0x2F range stay completely inside this range. Effectively, the memory map behaves in this manner:

2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
7	8	9	A	B	C	D	E	F	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0	1	2	3	4	5	6	7	8
h	i	j	k	l	m	n	o	p	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	a	b	c	d	e	f	g	h	i

Example 5–3 shows an LDNW performed with register A4 in circular mode and BK0 = 3, so the buffer size is 16 bytes, 8 half words, or 4 words. The value put in the AMR for this example is 00030001h. The buffer starts at address 0x0020 and ends at 0x002F. The register A4 is initialized to the address 0x002A.

Example 5–3. LDNW in Circular Mode

```
LDNW   .D1    *++A4[2],A1
```

	Before LDNW	1 cycle after LDNW	5 cycles after LDNW
A4	0000 002Ah	0000 0022h	0000 0022h
A1	XXXX XXXXh	XXXX XXXXh	5678 9ABCh
mem 0022h	5678 9ABCh	5678 9ABCh	5678 9ABCh

Note: 2h words is 8h bytes. 8h bytes is 3 bytes beyond the 16-byte (10h) boundary starting at address 002Ah; thus, it is wrapped around to 0022h (002Ah + 8h = 0022h).

5.8 Individual Instruction Descriptions

This section gives detailed information on the fixed-point instruction set for the C64x. Each instruction presents the following information:

❑ Assembler syntax
❑ Functional units
❑ Operands
❑ Opcode
❑ Description
❑ Execution
❑ Instruction type
❑ Delay slots
❑ See Also lists instructions of similar type
❑ Examples

ABS2 *Absolute Value With Saturation, Signed Packed 16-Bit*

Syntax

ABS2 (.unit) *src2, dst*
.unit = .L1, .L2

Opcode map field used...	For operand type...	Unit	Opfield
src2 *dst*	xs2 s2	.L1, .L2	00100

Opcode

31 29	28	27 23	22 18	17 13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src/cst*	*op*	x	0	0	1	1	0	1	0	1	1	0	*s*	*p*
3	1	5	5	5	1	10										1	1

Description

In the **ABS2** instruction, the absolute values of the upper and lower halves of the *src2* operand are placed in the upper and lower halves of the *dst.*

Execution

```
if (cond)    {
               (abs(lsb16(src2)) → lsb16(dst)
               (abs(msb16(src2)) → msb16(dst)
               }
else nop
```

Specifically, this instruction performs the following steps for each half-word of *src2,* then writes its result to the appropriate half-word of *dst*:

1) If the value is between 0 and 2^{15}, then value → *dst*

2) If the value is less than 0 and not equal to -2^{15}, then –value → *dst*

3) If the value is equal to -2^{15}, then $2^{15} -1$ → *dst*

Note:

This operation is performed on each 16-bit value separately. This instruction does not affect the SAT bit in the CSR.

Pipeline

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

Instruction Type Single-cycle

Delay Slots 0

See Also **ABS**

Example 1 `ABS2 .L1 A0,A15`

```
        Before instruction                          1 cycle after instruction

A0   FF68 4E3Dh    –152 20029               A0   FF68 4E3Dh    –152 20029

A2   XXXX XXXXh                             A2   0098 4E3Dh    152 20029
```

Example 2 `ABS2 .L1 A0,A15`

```
        Before instruction                          1 cycle after instruction

A0   3FF6 F105h    16374 –3835              A0   3FF6 F105h    16374 –3835

A2   XXXX XXXXh                             A2   3FF6 0EFBh    16374 3835
```

ADD2 Two 16-Bit Integer Adds on Upper and Lower Register Halves

Syntax

ADD2 (.unit) *src1, src2, dst*
.unit = .S1, .S2, .L1, .L2, .D1 or .D2

Opcode map field used...	For operand type...	Unit	Opfield
src1 *src2* *dst*	i2 xi2 i2	.S1, .S2	000001
src1 *src2* *dst*	i2 xi2 i2	.L1, .L2	0000101
src1 *src2* *dst*	i2 xi2 i2	.D1, .D2	0100

Opcode

.S unit

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

.L Unit

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

.D unit

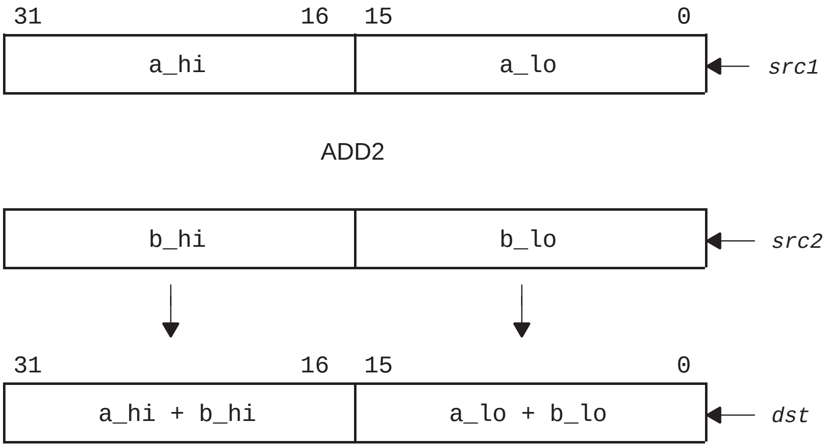
31 29	28	27 23	22 18	17 13	12	11	10	9 6	5	4	3	2	1	0
creg	*z*	dst	src2	src1	x	1	0	*op*	1	1	0	0	*s*	*p*
3	1	5	5	5	1	2		4	4				1	1

Description

In the **ADD2** instruction, the upper and lower halves of the *src1* operand are added to the upper and lower halves of the *src2* operand. The values in *src1* and *src2* are treated as signed, packed 16-bit data and the results are written in signed, packed 16-bit format into *dst*.

For each pair of signed packed 16-bit values found in the *src1* and *src2*, the sum between the 16-bit value from *src1* and the 16-bit value from *src2* is calculated to produce a16-bit result. The result is placed in the corresponding positions in the *dst*. The carry from the lower half add does not affect the upper half add.

This is the same **ADD2** instruction that is found on the C62x, but with the added flexibility of being able to perform this operation on the .L and .D units as well as the .S unit.



Execution

```
if (cond)   {
            msb16(src1) + msb16(src2) → msb16(dst);
            lsb16(src1) + lsb16(src2) → lsb16(dst);
            }
else nop
```

Pipeline

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

Instruction Type Single-cycle

Delay Slots 0

See Also **ADD, ADD4, SUB2**

Example

```
ADD2  .L1   A0,A1,A2
```

	Before instruction			**1 cycle after instruction**	
A0	0021 37E1h	33 14305 signed	A0	0021 37E1h	33 14305 signed
A1	039A E4B8h	922 –6984 signed	A1	039A E4B8h	922 –6984 signed
A2	XXXX XXXXh		A2	03BB 1C99h	955 7321 signed

ADD4 *Add Four 8-Bit Pairs For Four 8-Bit Results*

Syntax

ADD4 (.unit) *src1, src2, dst*
unit = .L1, .L2

Opcode map field used...	For operand type...	Unit	Opfield
src1 *src2* *dst*	i4 xi4 i4	.L1, .L2	1100101

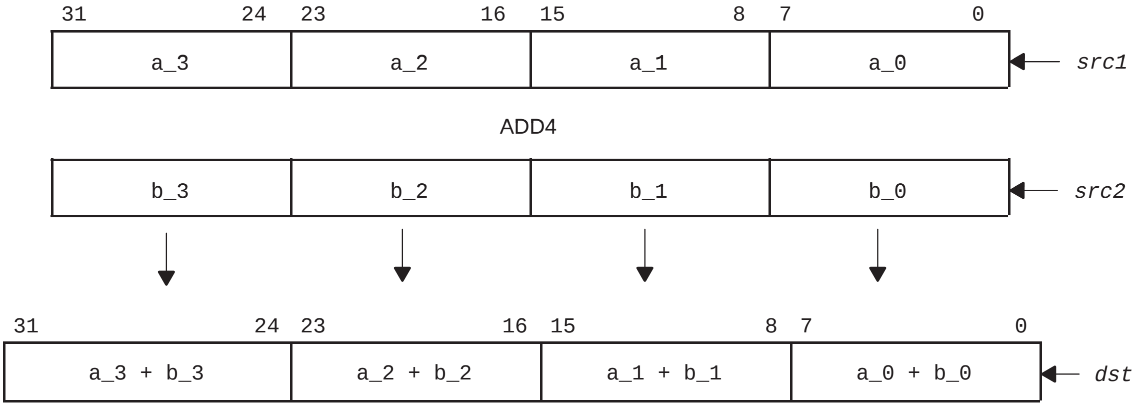
Opcode

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

Description

The **ADD4** instruction performs 2s-complement addition between packed 8-bit quantities. The values in *src1 and src2* are treated as packed 8-bit data and the results are written in packed 8-bit format.

For each pair of packed 8-bit values found in *src1* and *src2*, the sum between the 8-bit value from *src1* and the 8-bit value from *src2* is calculated to produce an 8-bit result. The result is placed in the corresponding positions in *dst.* No saturation is performed. The carry from one 8-bit add does not affect the add of any other 8-bit add.



Execution

```
if (cond)   {
              byte0(src1) + byte0(src2) → byte0(dst)
              byte1(src1) + byte1(src2) → byte1(dst)
              byte2(src1) + byte2(src2) → byte2(dst)
              byte3(src1) + byte3(src2) → byte3(dst)
            }
else nop
```

Pipeline

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

Instruction Type Single-cycle

Delay Slots 0

See Also **ADD, ADD2, SUB4**

Example 1

```
ADD4  .L1   A0,A1,A2
```

	Before instruction		**1 cycle after instruction**
A0	FF 68 4E 3Dh	A0	FF 68 4E 3Dh
A1	3F F6 F1 05h	A1	3F F6 F1 05h
A2	XXXX XXXXh	A2	3E 5E 3F 42h

Example 2

```
ADD4   .L1    A0,A1,A2
```

	Before instruction		1 cycle after instruction
A0	4A E2 D3 1Fh	A0	4A E2 D3 1Fh
A1	32 1A C1 28h	A1	32 1A C1 28h
A2	XXXX XXXXh	A2	7C FC 94 47h

ADDAD

Integer Addition Using Doubleword Addressing Mode

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

.D unit

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

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 2.6.2 on page 2-14). *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) → *dst*
else nop

Pipeline

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

Instruction Type Single-cycle

Delay Slots 0

See Also **ADDAB, ADDAH, ADDAW**

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

ADDKPC *Add a Signed 7-bit Constant to Program Counter*

Syntax

ADDKPC (.unit) *src1, dst, src2*
.unit = S2

Opcode map field used...	For operand type...	Unit
src1 *src2* *dst*	scst7 ucst3 uint	.S2

Opcode

31 29	28	27 23	22 16	15 13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*scst7*	*ucst3*	0	0	0	0	1	0	1	1	0	0	0	*s*	*p*
3	1	5	7	3	1	10										1	1

Description

In the **ADDKPC** instruction, a 7-bit signed constant is shifted 2 bits to the left, then added to the address of the first instruction of the fetch packet that contains the **ADDKPC** instruction (PCE1). The result is placed in the destination register. The 3-bit unsigned constant specifies the number of NOP cycles to insert after the current instruction. This instruction helps reduce the number of instructions needed to set up the return address for a function call.

The following code:

```
B      .S2   func
MVKL   .S2   LABEL, B3
MVKH   .S2   LABEL, B3
NOP     3
```

LABEL

could be replaced by:

```
B       .S2  func
ADDKPC .S2  LABEL, B3, 4
```

LABEL

Only one **ADDKPC** instruction can be executed per cycle. An **ADDKPC** instruction cannot be paired with any relative branch instruction in the same execute packet. If an **ADDKPC** and a relative branch are in the same execute packet, and if the **ADDKPC** instruction is executed when the branch is taken, behavior is undefined.

Execution if (cond) (scst7 << 2) + PCE1 → *dst*
else nop

Pipeline

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

Instruction Type Single-cycle

Delay Slots 0

See Also **B, BNOP**

Example

```
ADDKPC  .S2   30h,B3,5
```

	Before instruction		1 cycle after instruction
PCE1	0100 0000h		
B3	XXXX XXXXh	B3	0100 00C0h

AND Bitwise AND

Syntax **AND** (.unit) *src1, src2, dst*
.unit = .L1, .L2, .S1, .S2, .D1, .D2

Opcode map field used...	For operand type...	Unit	Opfield
src1 *src2* *dst*	uint xint uint	.L1, .L2	1111001
src1 *src2* *dst*	scst5 xuint uint	.L1, .L2	1111010
src1 *src2* *dst*	uint xunit uint	.S1, .S2	011111
src1 *src2* *dst*	scst5 xuint uint	.S1, .S2	011110
src *src2* *dst*	uint xuint uint	.D1, .D2	0110
src1 *src2* *dst*	scst5 xuint uint	.D1, .D2	0111

Opcode

.L unit form:

31 29	28	27 23	22 18	17 13	12	11 5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1lcst*	x	*op*	1	1	0	*s*	*p*
3		5	5	5	1	7		3		1	1

.S unit form:

31 29	28	27 23	22 18	17 13	12	11 6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1lcst*	x	*op*	1	1	0	0	*s*	*p*
3		5	5	5	1	6		4			1	1

.D unit form:

31 29	28	27 23	22 18	17 13	12	11	10	9 6	5	4	3	2	1	0
creg	z	*dst*	*src2*	*src1/cst5*	x	1	0	*op*	1	1	0	0	s	p
3	1	5	5	5	1	2		4	4				1	1

Description In this instruction a bitwise **AND** is performed between *src1* and *src2*. The result is placed in *dst.* The *scst5* operands are sign extended to 32 bits. This is the same **AND** instruction that is found on the C62x, but with the added flexibility of being able to perform this operation on the .D unit as well as the .L and .S units.

Execution if (cond) *src1* and *src2* → *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

See Also **ANDN, OR, XOR**

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

ANDN Bit-Wise Logical AND Invert

Syntax

ANDN (.unit) *src1, src2, dst*
unit = .L1, .L2, .D1, .D2, S1, .S2

Opcode map field used...	For operand type...	Unit	Opfield
src1 *src2* *dst*	uint xuint uint	.L1, .L2	1111100
src1 *src2* *dst*	uint xunit uint	.D1, .D2	0000
src1 *src2* *dst*	uint xuint uint	.S1, .S2	0110

Opcode

.D unit form:

31 29	28	27 23	22 18	17 13	12	11	10	9 6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	1	0	*op*	1	1	0	0	*s*	*p*
3	1	5	5	5	1	2		4	4				1	1

.L unit form:

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

.S unit form:

31 29	28	27 23	22 18	17 13	12	11	10	9 6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	1	1	*op*	1	1	0	0	*s*	*p*
3	1	5	5	5	1	2		4	4				1	1

Description

In the **ANDN** instruction, a bitwise logical AND is performed between *src1* and the bitwise logical inverse of *src2.* The result is placed in *dst.*

Execution

if (cond) *src1* and ~ *src2* → *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

See Also **AND, OR, XOR**

Example

```
ANDN    .L1     A0,A1,A2
```

	Before instruction			1 cycle after instruction
A0	1957 21ABh		A0	1957 21ABh
A1	081C 17E6h	F7E3E819	A1	081C 17E6h
A2	XXXX XXXXh		A2	1143 2009h

AVG2 *Average, Signed Packed 16-Bit*

Syntax

AVG2 (.unit) *src1*,*src2, dst*
.unit = .M1, .M2

Opcode map field used...	For operand type...	Unit	Opfield
src1 *src2* *dst*	s2 xs2 s2	.M1, .M2	10011

Opcode

31 29	28	27 23	22 18	17 13	12	11	10 6	5	4	3	2	1	0
creg	z	*dst*	*src2*	*src1*	x	0	*op*	1	1	0	0	*s*	*p*
3	1	5	5	5	1	1	5	4				1	1

Description

The **AVG2** instruction performs an averaging operation on packed 16-bit data. For each pair of signed 16-bit values found in *src1* and *src2*, **AVG2** calculates the average of the two values and returns a signed 16-bit quantity in the corresponding position in the *dst*.

Execution

```
if (cond)     {
              ((lsb16(src1) + lsb16(src2) + 1) >> 1) → lsb16(dst);
              ((msb16(src1) + msb16(src2) + 1) >> 1) → msb16(dst);
              }
```

The averaging operation is performed by adding *1* to the sum of the two 16-bit numbers being averaged. The result is then right-shifted by 1 to produce a 16-bit result.

Note: No overflow conditions exist.

Pipeline

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

Instruction Type Two-cycle

Delay Slots 1

See Also **AVGU4**

Example `AVG2 .M1 A0,A1,A2`

	Before instruction			2 cycles after instruction	
A0	6198 4357h	24984 17239	A0	6198 4357h	24984 17239
A1	7582 AE15	30082 –20971	A1	7582 AE15h	30082 –20971
A2	XXXX XXXXh		A2	6B8D F8B6	27533 –1866

AVGU4 *Average, Unsigned Packed 8-Bit*

Syntax

AVGU4 (.unit) *src1*,*src2, dst*
.unit = .M1, .M2

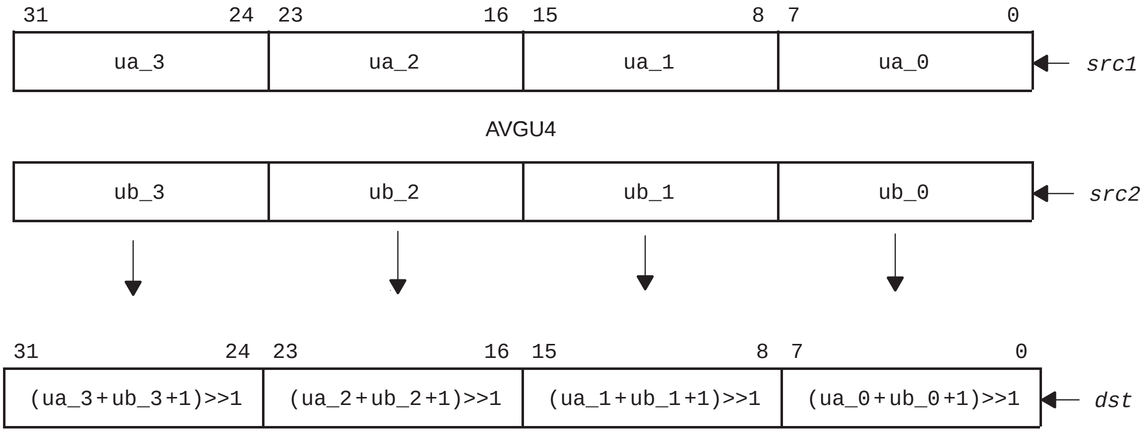
Opcode map field used...	For operand type...	Unit	Opfield
src1 *src2* *dst*	u4 xu4 u4	.M1, .M2	10010

Opcode

31 29	28	27 23	22 18	17 13	12	11	10 6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	0	*op*	1	1	0	0	*s*	*p*
3	1	5	5	5	1	1	5	4				1	1

Description

The **AVGU4** instruction performs an averaging operation on packed 8-bit data. The values in *src1* and *src2* are treated as unsigned, packed 8-bit data and the results are written in unsigned, packed 8-bit format. For each unsigned, packed 8-bit value found in *src1* and *src2*, **AVGU4** calculates the average of the two values and returns an unsigned, 8-bit quantity in the corresponding positions in the *dst*.



Execution

```
if (cond)    {
                ((ubyte0(src1) + ubyte0(src2) + 1) >> 1) → ubyte0(dst);
                (((ubyte1(src1) + ubyte1(src2) + 1) >> 1) → ubyte1(dst);
                (((ubyte2(src1) + ubyte2(src2) + 1) >> 1) → ubyte2(dst);
                (((ubyte3(src1) + ubyte3(src2) + 1) >> 1) → ubyte3(dst)
                }
else  nop
```

The averaging operation is performed by adding 1 to the sum of the two 8-bit numbers being averaged. The result is then right-shifted by 1 to produce an 8-bit result.

Note: No overflow conditions exist.

Pipeline

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

Instruction Type Two-cycle

Delay Slots 1

See Also **AVG2**

Example

```
AVGU4  .M1   A0,A1,A2
```

	Before instruction			2 cycles after instruction	
A0	1A 2E 5F 4Eh	26 46 95 78 unsigned	A0	1A 2E 5F 4Eh	26 46 95 78 unsigned
A1	9E F2 6E 3Fh	158 242 110 63 unsigned	A1	9E F2 6E 3Fh	158 242 110 63 unsigned
A2	XXXX XXXXh		A2	5C 90 67 47h	92 144 103 71 unsigned

BDEC

Branch and Decrement

Syntax

BDEC (.unit) *scst10, dst*
.unit = .S1, .S2

Opcode map field used...	For operand type...	Unit
src *dst*	scst10 int	.S1, .S2

Opcode

31 29	28	27 23	22 13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	z	*dst*	*scst10*	1	0	0	0	0	0	0	1	0	0	0	*s*	*p*
3	1	5	10	1	10										1	1

Description

If the predication and decrement register (*dst*) is positive (greater than or equal to 0), the **BDEC** instruction performs a relative branch and decrements *dst* by one. The instruction performs the relative branch using a 10-bit signed constant specified by the *scst10*. The constant is shifted 2 bits to the left, then added to the address of the first instruction of the fetch packet that contains the **BDEC** instruction (PCE1). The result is placed in the program fetch counter (PFC).

This instruction helps reduce the number of instructions needed to decrement a register and conditionally branch based upon the value of the register. Note also that any register can be used which can free the predicate registers (A0–A2 and B0–B2) for other uses.

The following code:

```
                   CMPLT     .L1   A10,0,A1
    [!A1]          SUB       .L1   A10,1,A10
  ||[!A1]          B         .S1   func
                   NOP  5
```

could be replaced by:

```
  BDEC  .S1   func, A10
  NOP    5
```

Note:

Only one **BDEC** instruction can be executed per cycle. The **BDEC** instruction can be predicated by using any conventional condition register. The conditions are effectively ANDed together. If two branches are in the same execute packet, and if both are taken, behavior is undefined.

Execution

```
if(cond)    {
              if (dst >=0), PFC = ((PCE1 + se(scst10)) <<2);
               if (dst >=0), dst = dst − 1;
              else nop
              }
else nop
```

Pipeline

	Target Instruction						
Pipeline Stage	**E1**	**PS**	**PW**	**PR**	**DP**	**DC**	**E1**
Read	*dst*						
Written	*dst, PC*						
Branch Taken							X
Unit in use	.S						

Instruction Type Branch

Delay Slots 5

Example 1

```
BDEC  .S1  100h,A10
```

	Before instruction		After branch has been taken
PCE1	0100 0000h		
PC	XXXX XXXXh	PC	0100 0400h
A10	0000 000Ah	A10	0000 0009h

Example 2

```
BDEC  .S1   300h,A10 ; 300h is sign extended
```

	Before instruction		After branch has been taken
PCE1	0100 0000h		
PC	XXXX XXXXh	PC	00FF FC00h
A10	0000 0010h	A10	0000 000Fh

BITC4 Bit Count, Packed 8-Bit

Syntax

BITC4 (.unit) *src2, dst*
.unit = .M1, .M2

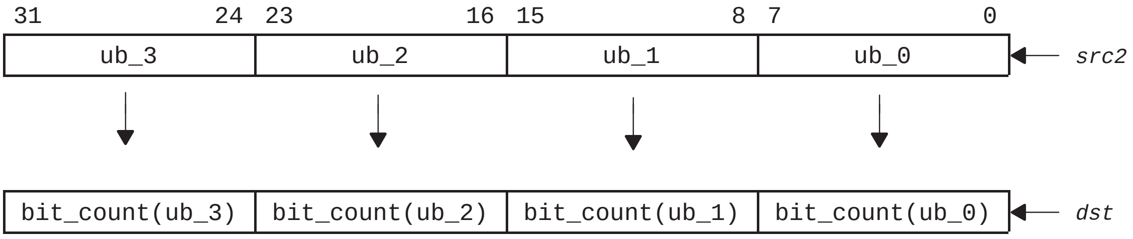
Opcode map field used...	For operand type...	Unit	Opfield
src2 *dst*	xu4 u4	.M1, .M2	11110

Opcode

31 29	28	27 23	22 18	17 13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	z	*dst*	*src*	*op*	x	0	0	0	0	1	1	1	1	0	0	s	p
3	1	5	5	5	1	10										1	1

Description

The **BITC4** instruction performs a bit-count operation on 8-bit quantities. The value in *src2* is treated as packed 8-bit data, and the result is written in packed 8-bit format. For each of the 8-bit quantities in *src2*, the count of the number of “1” bits in that value is written to the corresponding position in *dst*.



Execution

```
if (cond)    {
             ((bit_count src2(ubyte0)) → ubyte0(dst);
             ((bit_count src2(ubyte1)) → ubyte1(dst);
             ((bit_count src2(ubyte2)) → ubyte2(dst);
             ((bit_count src2(ubyte3)) → ubyte3(dst);
             }
else nop
```

Pipeline

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

Instruction Type

Two-cycle

Delay Slots

1

Example

```
BITC4  .M1   A1,A2
```

	Before instruction		2 cycles after instruction
A1	9E 52 6E 30h	A1	9E 52 6E 30h
A2	XXXX XXXX h	A2	05 03 05 02h

BITR Bit Reverse

Syntax

BITR (.unit) *src2, dst*
.unit = .M1, .M2

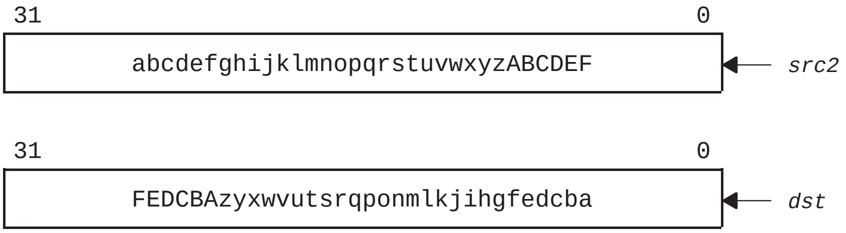
Opcode map field used...	For operand type...	Unit	Opfield
src2 *dst*	xu4 u4	.M1, .M2	11111

Opcode

31 29	28	27 23	22 18	17 13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	z	*dst*	*src*	*op*	x	0	0	0	0	1	1	1	1	0	0	*s*	*p*
3	1	5	5	5	1	10										1	1

Description

The **BITR** instruction implements a bit-reversal. A bit-reversal function reverses the order of bits in a 32-bit word. This means that bit 0 of the source becomes bit 31 of the result, bit 1 of the source becomes bit 30 of the result, bit 2 becomes bit 29, and so on.


31
0
abcdefghijklmnopqrstuvwxyzABCDEF
src2
31
0
FEDCBAzyxwvutsrqponmlkjihgfedcba
dst

Execution

```
if (cond)    {
             bit_reverse (src2) → dst
             }
else  nop
```

Pipeline

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

Instruction Type Two-cycle

Delay Slots 1

Example

```
BITR  .M2   B4,B5
```

	Before instruction		2 cycles after instruction
B4	A6E2 C179h	B4	A6E2 C179h
B5	XXXX XXXXh	B5	9E83 4765h

BNOP *Branch Using a Displacement With NOP*

Syntax

BNOP (.unit) *src2, src1*
.unit = .S1, .S2

Opcode map field used...	For operand type...	Unit
src2	scst12	.S1,
src1	ucst3	.S2

Opcode

31 29	28	27 16	15 13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*scst12*	*ucst3*	0	0	0	0	1	0	0	1	0	0	0	*s*	*p*
3	1	12	3	1	10										1	1

Description

The constant displacement form of the **BNOP** instruction performs a relative branch with **NOP** instructions. The instruction performs the relative branch using the 12-bit signed constant specified by *src2*. The constant is shifted 2 bits to the left, then added to the address of the first instruction of the fetch packet that contains the **BNOP** instruction (PCE1). The result is placed in the program fetch counter (PFC).

The 3-bit unsigned constant specified in *src1* gives the number of delay slot **NOP** instructions to be inserted, from 0 to 5. With *src1* = 0, no **NOP** cycles are inserted.

This instruction helps reduce the number of instructions to perform a branch when **NOP** instructions are required to fill the delay slots of a branch.

The following code:

```
   B         .S1    LABEL
   NOP       N
LABEL: ADD
```

could be replaced by:

```
   BNOP      .S1    LABEL, N
LABEL: ADD
```

Note:

BNOP instructions may be predicated. The predication condition controls whether or not the branch is taken, but does not affect the insertion of **NOP**s. **BNOP** always inserts the number of **NOP**s specified by N, regardless of the predication condition.

Only one branch instruction can be executed per cycle. If two branches are in the same execute packet, and if both are taken, the behavior is undefined. It should also be noted that when a predicated **BNOP** instruction is used with a **NOP** count greater than 5, the C64x will insert the full delay slots requested when the predicated condition is false.

For example, the following set of instructions will insert 7 cycles of **NOP**s:

```
      ZERO  .L1  A0
[A0]  BNOP .S1  LABEL,7; branch is not taken and
                     ; 7 cycles of NOPs are inserted
```

Conversely, when a predicated **BNOP** instruction is used with a **NOP** count greater than 5 and the predication condition is true, the branch will be taken and the multi-cycle **NOP** is terminated when the branch is taken.

For example in the following set of instructions, only 5 cycles of **NOP** are inserted:

```
      MVK   .D1  1,A0
[A0]  BNOP  .S1  LABEL,7; branch is taken and 5 cycles of
                       ; NOPs are inserted
```

Execution

```
if (cond)   {
              PFC = (PCE1 + (se(scst12) << 2));
              nop (src1);
              }
else      nop(src1 +1)
```

Pipeline

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

Instruction Type Branch

Delay Slots 5

See Also **ADDKPC**, **B, NOP**

Example

```
BNOP  .S1   30h,2
```

	Before instruction		**After branch has been taken**
PCE1	0100 0500h		
PC	XXXX XXXXh	PC	0100 1100h

BNOP — *Branch Using a Register With NOP*

Syntax

BNOP (.unit) *src2, src1*
.unit = .S2

Opcode map field used...	For operand type...	Unit
src2	xuint	.S2
src1	ucst3	

Opcode

31 29	28	27				23	22 18	17	16	15 13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	z	0	0	0	0	1	*src2*	0	0	*ucst3*	x	0	0	1	1	0	1	1	0	0	0	1	*p*
3	1	5					5	2		3	1	10										1	1

Description

The register form of the **BNOP** instruction performs an absolute branch with **NOP** instructions. The register specified in *src2* is placed in the program fetch counter (PFC).

The 3-bit unsigned constant specified in *src1* gives the number of delay slots **NOP** instructions to be inserted, from 0 to 5. With *src1* = 0, no NOP cycles are inserted.

This instruction helps reduce the number of instructions to perform a branch when **NOP** instructions are required to fill the delay slots of a branch.

The following code:

```
B     .S2   B3
```

```
NOP   N
```

could be replaced by:

```
BNOP  .S2   B3,   N
```

Note:

BNOP instructions may be predicated. The predication condition controls whether or not the branch is taken, but does not affect the insertion of **NOP**s. **BNOP** always inserts the number of **NOP**s specified by N, regardless of the predication condition.

Only one branch instruction can be executed per cycle. If two branches are in the same execute packet, and if both are taken, the behavior is undefined. It should also be noted that when a predicated **BNOP** instruction is used with a **NOP** count greater than 5, the C64x will insert the full delay slots requested when the predicated condition is false.

For example, the following set of instructions will insert 7 cycles of **NOP**s:

```
       ZERO  .L1  A0
[A0]   BNOP   .S1   B3,7;   branch   is   not   taken   and
                        ;  7 cycles of NOPs are inserted
```

Conversely, when a predicated **BNOP** instruction is used with a **NOP** count greater than 5 and the predication condition is true, the branch will be taken and multi-cycle **NOP** is terminated when the branch is taken.

For example, in the following set of instructions only 5 cycles of **NOP** are inserted:

```
       MVK   .D1  1,A0
[A0]   BNOP   .S1   B3,7; branch is taken and 5 cycles of
                       ; NOPs are inserted
```

Execution

```
if (cond)   {
             src2 → PFC
             nop (src1);
             }
else      nop (src1 +1)
```

Pipeline

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

Instruction Type Branch

Delay Slots 5

See Also **ADDKPC**, **B, NOP**

Example `BNOP  .S2   A5,2`

	Before instruction		After branch has been taken
PCE1	0010 0000h		
PC	XXXX XXXXh	PC	0100 F000h
A5	0100 F000h	A5	0100 F000h

BPOS

Branch Positive

Syntax

BPOS (.unit) *scst10, dst*
.unit = .S1, .S2

Opcode map field used...	For operand type...	Unit
src *dst*	scst10 int	.S1, .S2

Opcode

31 29	28	27 23	22 13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	z	*dst*	*scst10*	0	0	0	0	0	0	0	1	0	0	0	s	p
3	1	5	10	1	10										1	1

Description

If the predication register *(dst)* is positive (greater than or equal to 0), the **BPOS** instruction performs a relative branch. If *dst* is negative, the **BPOS** instruction takes no other action.

The instruction performs the relative branch using a 10-bit signed constant specified by the *scst10*. The constant is shifted 2 bits to the left, then added to the address of the first instruction of the fetch packet that contains the BDEC instruction (PCE1). The result is placed in the program fetch counter (PFC).

Any register can be used which can free the predicate registers (A0–A2 and B0–B2) for other uses.

Note:

Only one **BPOS** instruction can be executed per cycle. The **BPOS** instruction can be predicated by using any conventional condition register. The conditions are effectively ANDed together. If two branches are in the same execute packet, and if both are taken, behavior is undefined.

Execution

```
if (cond)   {
             if (dst >=0), PFC = (PCE1 + (se(scst10) << 2));
            else  nop
            }
else        nop
```

Pipeline

	Target Instruction						
Pipeline Stage	**E1**	**PS**	**PW**	**PR**	**DP**	**DC**	**E1**
Read	*dst*						
Written	*PC*						
Branch Taken							X
Unit in use	.S						

Instruction Type Branch

Delay Slots 5

Example

```
BPOS  .S1   200h,A10
```

	Before instruction		**After branch has been taken**
PCE1	0010 0000h		
PC	XXXX XXXXh	PC	0100 0800h
A10	0000 000Ah	A10	0000 000Ah

CMPEQ2 *Compare if Equal, Packed 16-Bit*

Syntax

CMPEQ2 (.unit) *src1*,*src2, dst*
.unit = .S1, .S2

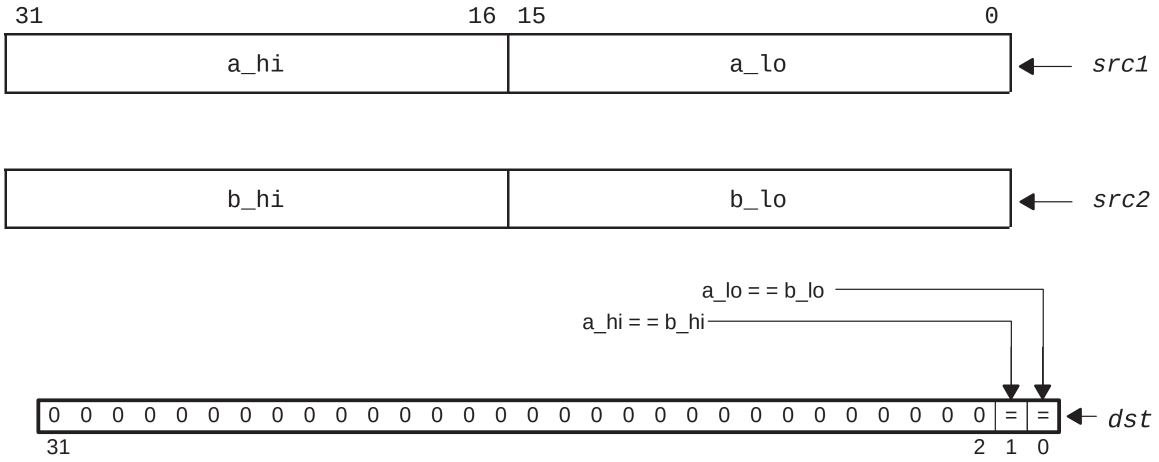
Opcode map field used...	For operand type...	Unit	Opfield
src1 *src2* *dst*	s2 xs2 bv2	.S1, .S2	011101

Opcode

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

Description

The **CMPEQ2** instruction performs equality comparisons on packed 16-bit data. Each 16-bit value in *src1* is compared against the corresponding 16-bit value in *src2*, returning either a 1 if equal or 0 if not equal. The equality results are packed into the two least-significant bits of *dst*. The result for the lower pair of values is placed in bit 0, and the results for the upper pair of values are placed in bit 1. The remaining bits of *dst* are set to 0.



Execution

```
if (cond    {
        if (lsb16(src1) == lsb16(src2), 1 $\rightarrow$ $dst_0$
            else 0 $\rightarrow$ $dst_0$ ;
        if (msb16(src1) == msb16(src2)), 1 $\rightarrow$ $dst_1$
            else 0 $\rightarrow$ $dst_1$
        }
else nop
```

Pipeline

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

Instruction Type Single-cycle

Delay Slots 0

See Also **CMPEQ, CMPEQ4, CMPGT2, XPND2**

Example 1

```
CMPEQ2  .S1   A3,A4,A5
```

	Before instruction		1 cycle after instruction
A3	1105 6E30h	A3	1105 6E30h
A4	1105 6980h	A4	1105 6980h
A5	XXXX XXXXh	A5	0000 0002h

Example 2

```
CMPEQ2  .S2   B2,B8,B15
```

	Before instruction		1 cycle after instruction
B2	F23A 3789h	B2	F23A 3789h
B8	04B8 3789h	B8	04B8 3789h
B15	XXXX XXXXh	B15	0000 0001h

Example 3

```
CMPEQ2  .S2  B2,B8,B15
```

	Before instruction		1 cycle after instruction
B2	01B6 2451h	B2	01B6 2451h
B8	01B6 2451h	B8	01B6 2451h
B15	XXXX XXXXh	B15	0000 0003h

CMPEQ4 *Compare if Equal, Packed 8-Bit*

Syntax

CMPEQ4 (.unit) *src1*,*src2, dst*
.unit = .S1, .S2

Opcode map field used...	For operand type...	Unit	Opfield
src1 *src2* *dst*	s4 xs4 bv4	.S1, .S2	011100

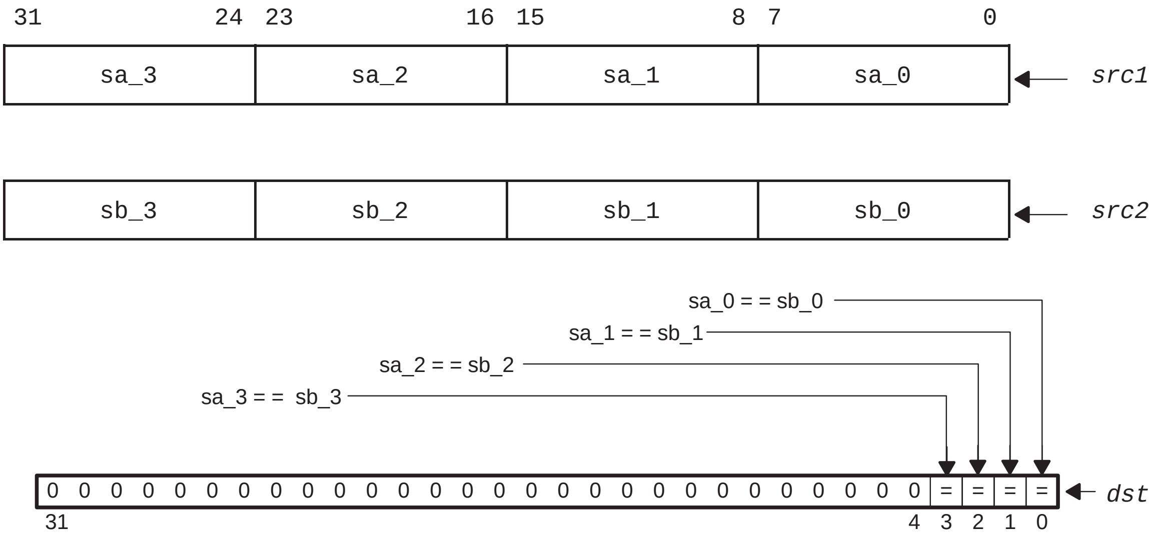
Opcode

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

Description

The **CMPEQ4** instruction performs equality comparisons on packed 8-bit data. Each 8-bit value in *src1* is compared against the corresponding 8-bit value in *src2*, returning a 1 if equal or 0 if not equal. The equality comparison results are packed into the four least-significant bits of *dst*.

The 8-bit values in each input are numbered from 0 to 3, starting with the least-significant byte, then working towards the most- significant byte. The comparison results for byte 0 are written to bit 0 of the result. Likewise the results for byte 1 to 3 are written to bits 1 to 3 of the result, respectively, as shown in the diagram below. The remaining bits of *dst* are set to 0.



Execution

```
if (cond     {
        if (sbyte0(src1)  ==  sbyte0(src2)), 1 → $dst_0$
            else 0 → $dst_0$  ;
        if (sbyte1(src1)  ==  sbyte1(src2)), 1 → $dst_1$
            else 0 →  $dst_1$
        if (sbyte2(src1) == sbyte2(src2)), 1 → $dst_2$
            else 0 →  $dst_2$
        if (sbyte3(src1) == sbyte3(src2)), 1 → $dst_3$
            else 0 →  $dst_3$
         }
else nop
```

Pipeline

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

Instruction Type Single-cycle

Delay Slots 0

See Also **CMPEQ, CMPEQ2, CMPGTU4, XPND4**

Example 1

```
CMPEQ4  .S1   A3,A4, A5
```

	Before instruction		1 cycle after instruction
A3	02 3A 4E 1Ch	A3	02 3A 4E 1Ch
A4	02 B8 4E 76h	A4	02 B8 4E 76h
A5	XXXX XXXXh	A5	0000 000Ah

Example 2

```
CMPEQ4  .S2   B2,B6,B13
```

	Before instruction		1 cycle after instruction
B2	F2 3A 37 89h	B2	F2 3A 37 89h
B8	04 B8 37 89h	B8	04 B8 37 89h
B13	XXXX XXXXh	B13	0000 0003h

Example 3

```
CMPEQ4  .S2   B2,B8,B13
```

	Before instruction		1 cycle after instruction
B2	01 B6 24 51h	B2	01 B6 24 51h
B8	05 B6 24 51h	B8	05 B6 24 51h
B13	XXXX XXXXh	B13	0000 0007h

CMPGT2

Compare for Greater Than, Packed 16-Bit

Syntax

CMPGT2 (.unit) *src1*,*src2, dst*
.unit = .S1, .S2

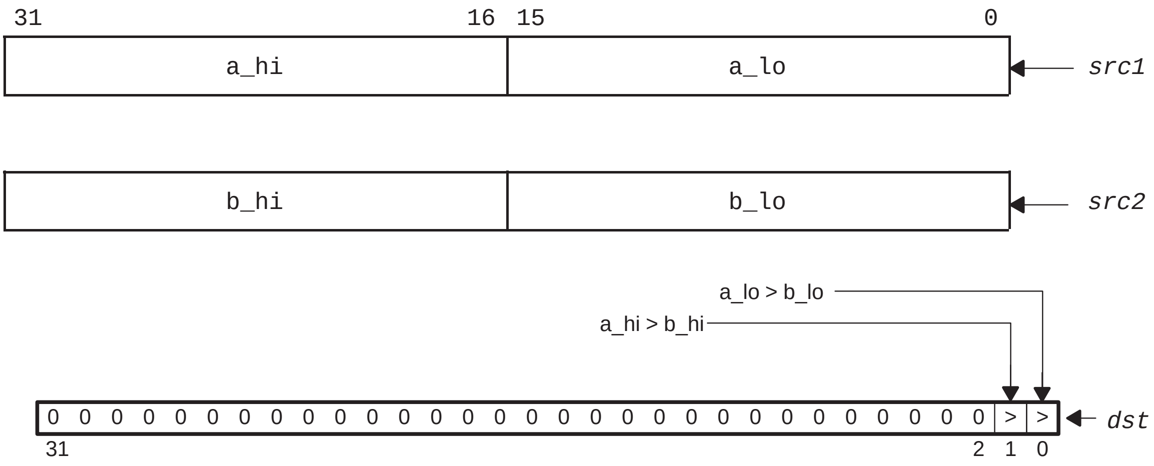
Opcode map field used...	For operand type...	Unit	Opfield
src1 *src2* *dst*	s2 xs2 bv2	.S1, .S2	010100

Opcode

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

Description

The **CMPGT2** instruction performs comparisons for greater than values on signed, packed 16-bit data. Each signed 16-bit value in *src1* is compared against the corresponding signed 16-bit value in *src2*, returning a 1 if *src1* is greater than *src2* or returning a 0 if it is not greater. The comparison results are packed into the two least-significant bits of *dst*. The result for the lower pair of values is placed in bit 0, and the results for the upper pair of values are placed in bit 1. The remaining bits of *dst* are set to 0.



Execution

```
if (cond)   {
                if (lsb16(src1) > lsb16(src2), 1 → $dst_0$
                    else 0 → $dst_0$ ;
                if (msb16(src1) > msb16(src2)), 1 → $dst_1$
                    else 0 → $dst_1$
                }
else nop
```

Pipeline

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

Instruction Type Single-cycle

Delay Slots 0

See Also **CMPEQ2, CMPGT, CMPGTU4, XPND2**

Example 1

```
CMPGT2  .S1   A3,A4,A5
```

```
         Before instruction                   1 cycle after instruction

A3   1105 6E30h    4357 28208        A3   1105 6E30h    4357 28208

A4   1105 6980h    4357 27008        A4   1105 6980h    4357 22008

A5   XXXX XXXXh                      A5   0000 0001h
```

Example 2

```
CMPGT2  .S2   B2,B8,B15
```

```
          Before instruction                   1 cycle after instruction

B2    F23A 3789h    –3526 14217       B2    F23A 3789h    –3526 14217

B8    04B8 4975h    1208 18805        B8    04B8 4975h    1208 18805

B15   XXXX XXXX h                     B15   0000 0000h
```

Example 3

```
CMPGT2  .S2   B2, B8, B15
```

```
          Before instruction                   1 cycle after instruction

B2    01A6 2451h    422 9297          B2    01A6 2451h    422 9297

B8    0124 A051h    292 –24495        B8    0124 A051h    292 –24495

B15   XXXX XXXXh                      B15   0000 0003h
```

CMPGTU4 *Compare for Greater Than, Unsigned Packed 8-Bit*

Syntax

CMPGTU4 (.unit) *src1*,*src2, dst*
.unit = .S1, .S2

Opcode map field used...	For operand type...	Unit	Opfield
src1 *src2* *dst*	u4 xu4 bv4	.S1, .S2	010101

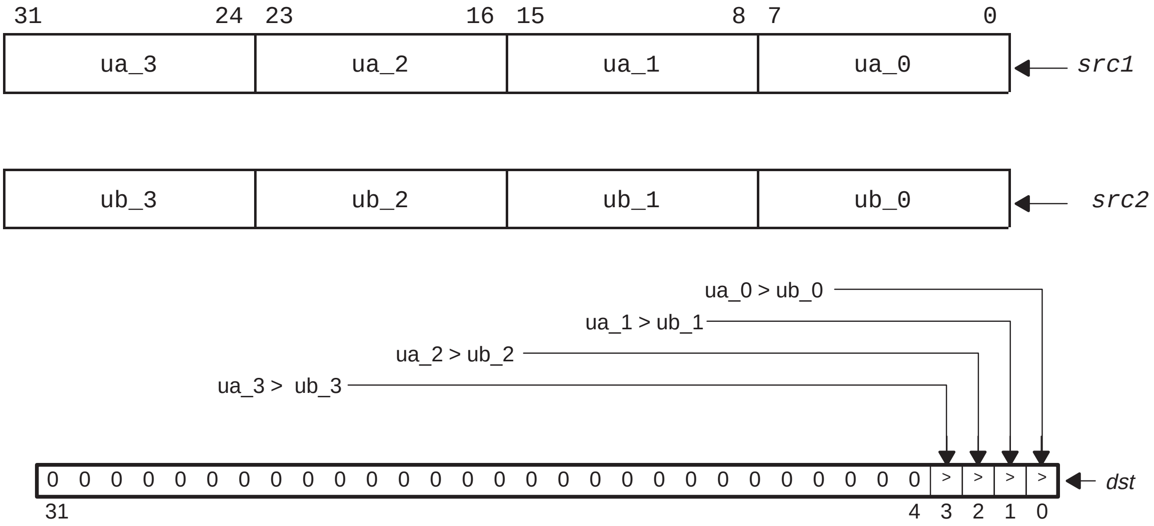
Opcode

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

Description

The **CMPGTU4** instruction performs comparisons for greater than values on packed 8-bit data. Each unsigned 8-bit value in *src1* is compared against the corresponding 8-bit unsigned value in *src2*, returning a 1 if the byte in *src1* is greater than the corresponding byte in *src2* or 0 if is not greater. The comparison results are packed into the four least-significant bits of *dst*.

The 8-bit values in each input are numbered from 0 to 3, starting with the least-significant byte, then working towards the most-significant byte. The comparison results for byte 0 are written to bit 0 of the result. Likewise, the results for byte 1 to 3 are written to bits 1 to 3 of the result, respectively, as shown in the diagram below. The remaining bits of *dst* are set to 0.



Execution

```
if (cond)    {
                 if (ubyte0(src1) > ubyte0(src2)), 1 → $dst_0$
                     else 0 → $dst_0$;
                 if (ubyte1(src1) > ubyte1(src2)), 1 → $dst_1$
                     else 0 → $dst_1$;
                 if (ubyte2(src1) > ubyte2(src2)), 1 → $dst_2$
                     else 0 → $dst_2$;
                 if (ubyte3(src1) > ubyte3(src2)), 1 → $dst_3$
                     else 0 → $dst_3$;
             }
else  nop
```

Pipeline

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

Instruction Type Single-cycle

Delay Slots 0

See Also **CMPEQ4, CMPGT, CMPGT2, XPND4**

Example 1

```
CMPGTU4  .S1   A3,A4,A5
```

	Before instruction			1 cycle after instruction	
A3	25 3A 1C E4h	37 58 28 228	A3	25 3A 1C E4h	37 58 28 228
A4	02 B8 4E 76h	2 184 78 118	A4	02 B8 4E 76h	2 184 78 118
A5	XXXX XXXXh		A5	0000 0009h	

Example 2

```
CMPGTU4  .S2   B2,B8,B13
```

	Before instruction			1 cycle after instruction	
B2	89 F2 3A 37h	137 242 58 55	B2	89 F2 3A 37h	137 242 58 55
B8	04 8F 17 89h	4 143 23 137	B8	04 8F 17 89h	4 143 23 137
B13	XXXX XXXXh		B13	0000 000Eh	

Example 3

```
CMPGTU4  .S2   B2,B8,B13
```

	Before instruction			1 cycle after instruction	
B2	12 33 9D 51h	18 51 157 81	B2	12 33 9D 51h	18 51 157 81
B8	75 67 24 C5h	117 103 36 197	B8	75 67 24 C5h	117 103 36 197
B13	XXXX XXXXh		B13	0000 0002h	

CMPLT2 *Compare for Less Than, Packed 16-Bit (Pseudo-Operation)*

Syntax

CMPLT2 (.unit) *src2, src1*, *dst*
.unit = .S1, .S2

Opcode map field used...	For operand type...	Unit	Opfield
src1 *src2* *dst*	s2 xs2 bv2	.S1, .S2	010100

Opcode

See **CMPGT2** instruction.

Description

The **CMPLT2** instruction is a pseudo-operation used to perform less-than comparisons on signed, packed 16-bit data. Each signed 16-bit value in *src2* is compared against the corresponding signed 16-bit value in *src1*, returning a 1 if *src2* is less than *src1* or returning a 0 if it is not less than. The comparison results are packed into the two least-significant bits of *dst*. The result for the lower pair of values is placed in bit 0, and the results for the upper pair of values are placed in bit 1. The remaining bits of *dst* are set to 0. The assembler uses the operation **CMPGT2** (.unit) *src1, src2, dst* to perform this task.

Execution

```
if (cond)    {
                if (lsb16(src2) < lsb16(src1), 1 → dst_0
                    else 0 → dst_0;
                if (msb16(src2) < msb16(src1)), 1 → dst_1
                    else 0 → dst_1
             }
else nop
```

Pipeline

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

Instruction Type Single-cycle

Delay Slots 0

See Also **CMPEQ2, CMPGT2, CMPLTU4, XPND2**

Example 1 `CMPLT2 .S1 A4,A3,A5; assembler treats as CMPGT2 A3,A4,A5`

	Before instruction			1 cycle after instruction	
A3	1105 6E30h	4357 28208	A3	1105 6E30h	4357 28208
A4	1105 6980h	4357 27008	A4	1105 6980h	4357 27008
A5	XXXX XXXXh		A5	0000 0001h	

Example 2 `CMPLT2 .S2 B8,B2,B15; assembler treats as CMPGT2 B2,B8,B15`

	Before instruction			1 cycle after instruction	
B2	F23A 3789h	–3526 14217	B2	F23A 3789h	–3526 14217
B8	04B8 4975h	1208 18805	B8	04B8 4975h	1208 18805
B15	XXXX XXXXh		B15	0000 0000h	Both are false

Example 3 `CMPLT2 .S2 B8,B2,B12; assembler treats as CMPGT2 B2,B8,B15`

	Before instruction			1 cycle after instruction	
B2	01A6 2451h	422 9297	B2	01A6 2451h	422 9297
B8	0124 A051h	292 –24495	B8	0124 A051h	292 –24495
B12	XXXX XXXXh		B12	0000 0003h	Both are true

CMPLTU4 *Compare for Less Than, Unsigned Packed 8-Bit (Pseudo-Operation)*

Syntax

CMPLTU4 (.unit) *src2*, *src1, dst*
.unit = .S1, .S2

Opcode map field used...	For operand type...	Unit	Opfield
src1 *src2* *dst*	u4 xu4 bv4	.S1, .S2	010101

Opcode

See **CMPGTU4** instruction.

Description

CMPLTU4 is a pseudo-operation that performs less-than comparisons on packed 8-bit data. Each unsigned 8-bit value in *src2* is compared against the corresponding 8-bit unsigned value in *src1*, returning a 1 if the byte in *src2* is less than the corresponding byte in *src1* or 0 it if is not less than. The comparison results are packed into the four least-significant bits of *dst*.

The 8-bit values in each input are numbered from 0 to 3, starting with the least-significant byte, and moving towards the most-significant byte. The comparison results for byte 0 are written to bit 0 of the result. Similarly, the results for byte 1 to 3 are written to bits 1 to 3 of the result, respectively, as shown in the **CMPGTU4** instruction diagram. The remaining bits of *dst* are set to 0.

The assembler uses the operation **CMPGTU4** (.unit) *src1, src2, dst* to perform this task.

Execution

```
if (cond)   {
             if (ubyte0(src2)  <  ubyte0(src1)), 1 → $dst_0$
                  else 0 → $dst_0$;
             if (ubyte1(src2)  <  ubyte1(src1)), 1 → $dst_1$
                  else 0 → $dst_1$;
             if (ubyte2(src2) < ubyte2(src2)), 1 → $dst_2$
                  else 0 → $dst_2$;
             if (ubyte3(src2) < ubyte3(src1)), 1 → $dst_3$
                  else 0 → $dst_3$;
             }
else  nop
```

Pipeline

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

Instruction Type Single-cycle

Delay Slots 0

See Also **CMPEQ4, CMPGT, CMPLT2, XPND4**

Example 1

```
CMPLTU4 .S1 A4,A3,A5; assembler treats as CMPGTU4 A3,A4,A5
```

	Before instruction			1 cycle after instruction	
A3	25 3A 1C E4h	37 58 28 228	A3	02 3A 1C E4h	37 58 28 228
A4	02 B8 4E 76h	2 184 78 118	A4	02 B8 4E 76h	2 184 78 118
A5	XXXX XXXXh		A5	0000 0009h	

Example 2

```
CMPLTU4 .S2 B8,B2,B13; assembler treats as CMPGTU4 B2,B8,B13
```

	Before instruction			1 cycle after instruction	
B2	89 F2 3A 37h	137 242 58 55	B2	89 F2 3A 37h	137 242 58 55
B8	04 8F 17 89h	4 143 23 137	B8	04 8F 17 89h	4 143 23 137
B13	XX XX XX XXh		B13	0000 000Eh	

Example 3

```
CMPLTU4 .S2 B8,B2,B13; assembler treats as CMPGTU4 B2,B8,B13
```

	Before instruction			1 cycle after instruction	
B2	12 33 9D 51h	18 51 157 81	B2	12 33 9D 51h	18 51 157 81
B8	75 67 24 C5h	117 103 36 197	B8	75 67 24 C5h	117 103 36 197
B13	XX XX XX XXh		B13	0000 0002h	

DEAL — *De-Interleave and Pack*

Syntax

DEAL (.unit) *src2, dst*
.unit = .M1, .M2

Opcode map field used...	For operand type...	Unit	Opfield
src2 *dst*	xuint uint	.M1, .M2	11101

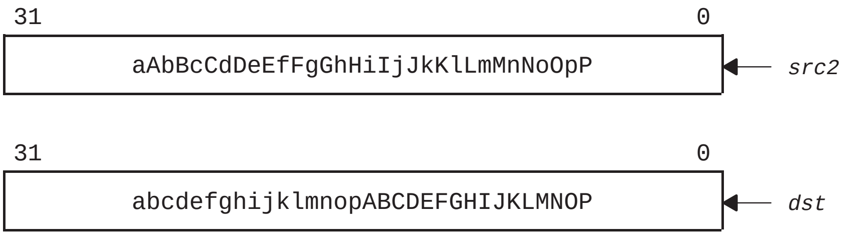
Opcode

31 29	28	27 23	22 18	17 13	12	11	9	9	8	7	6	5	4	3	2	1	0
creg	z	*dst*	*src2*	*op*	x	0	0	0	0	1	1	1	1	0	0	*s*	*p*
3	1	5	5	5	1	10										1	1

Description

The **DEAL** instruction performs a de-interleave and pack operation on the bits in *src2*. The odd and even bits of *src2* are extracted into two separate, 16-bit quantities. These 16-bit quantities are then packed such that the even bits are placed in the lower half-word, and the odd bits are placed in the upper half word.

As a result, bits 0, 2, 4, ... , 28, 30 of *src2* are placed in bits 0, 1, 2, ... , 14, 15 of *dst*. Likewise, bits 1, 3, 5, ... , 29, 31 of *src2* are placed in bits 16, 17, 18, ... , 30, 31 of *dst.*


31
0
aAbBcCdDeEfFgGhHiIjJkKlLmMnNoOpP
src2
31
0
abcdefghijklmnopABCDEFGHIJKLMNOP
dst

Note:

The **DEAL** instruction is the exact inverse of the **SHFL** instruction.

Execution

```
if (cond)    {
                 $src2_{31,29,27...1} \rightarrow dst_{31,30,29...16}$ ;
                 $src2_{30,28,26...0} \rightarrow dst_{15,14,13...0}$ ;
             }
else  nop
```

Pipeline

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

Instruction Type Two-cycle

Delay Slots 1

See Also **SHFL**

Example `DEAL  .M1   A1,A2`

	Before instruction		2 cycles after instruction
A1	`9E52 6E30h`	A1	`9E52 6E30h`
A2	`XXXX XXXXh`	A2	`B174 6CA4h`

DOTP2 *Dot Product, Signed Packed 16-Bit*

Syntax

DOTP2 (.unit) *src1*,*src2, dst*
.unit = .M1, .M2

Opcode map field used...	For operand type...	Unit	Opfield
src1 *src2* *dst*	s2 xs2 int	.M1, .M2	01100
src1 *src2* *dst*	s2 xs2 sllong	.M1, .M2	01011

Opcode

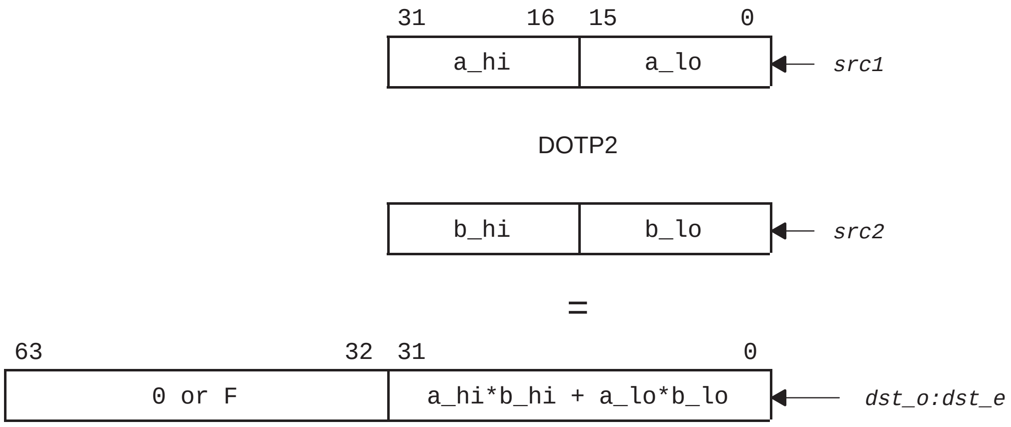
31 29	28	27 23	22 18	17 13	12	11	10 6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	0	*op*	1	1	0	0	*s*	*p*
3	1	5	5	5	1	1	5					1	1

Description

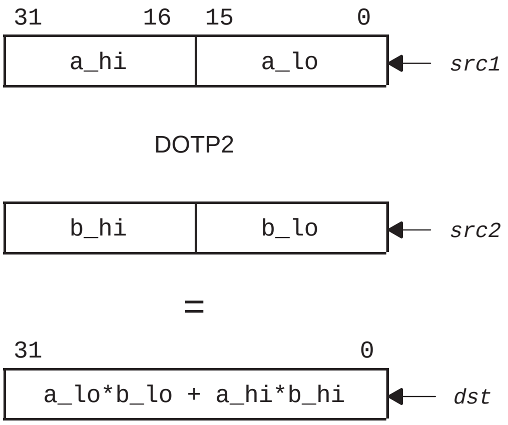
The **DOTP2** instruction returns the dot-product between two pairs of signed, packed 16-bit values. The values in *src1* and *src2* are treated as signed, packed 16-bit quantities. The signed result is written either to a single 32-bit register, or sign-extended into a 64-bit register pair.

The product of the lower half-words of *src1* and *src2* is added to the product of the upper half-words of *src1* and *src2*. The result is then written to the *dst*.

In the 64-bit result version the upper word of the register pair always contains either all 0s or all 1s, depending on whether the result is positive or negative, respectively.



The 32-bit result version returns the same results that the 64-bit result version does in the lower 32 bits. The upper 32-bits are discarded.



Note:

In the overflow case, where all four half-words in *src1* and *src2* are 0x8000, the value 0x80000000 is written into the 32-bit *dst* and 0x0000000080000000 is written into the 64-bit *dst*.

Execution

```
if (cond)    {
                 (lsb16(src1)  x  lsb16(src2)) +
                 (msb16(src1) x msb16(src2)) → dst
                 }
else  nop
```

Pipeline

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

Instruction Type Four-cycle

Delay Slots 3

See Also **DOTPN2**

Example 1

```
DOTP2  .M1   A5,A6,A8
```

	Before instruction			**4 cycles after instruction**	
A5	6A32 1193h	27186 4499	A5	6A32 1193h	27186 4499
A6	B174 6CA4h	–20108 27812	A6	B174 6CA4h	–20108 27812
A8	XXXX XXXXh		A8	E6DF F6D4h	–421529900

Example 2

```
DOTP2  .M1   A5,A6,A9:A8
```

	Before instruction			4 cycles after instruction	
A5	6A32 1193h	27186 4499	A5	6A32 1193h	27186 4499
A6	B174 6CA4h	−20108 27812	A6	B174 6CA4h	−20108 27812
A9:A8	XXXX XXXXh	XXXX XXXXh	A9:A8	FFFF FFFFh	E6DF F6D4h
					−421529900

Example 3

```
DOTP2  .M2   B2,B5,B8
```

	Before instruction			4 cycles after instruction	
B2	1234 3497h	4660 13463	B2	1234 3497h	4660 13463
B5	21FF 50A7h	8703 20647	B5	21FF 50A7h	8703 20647
B8	XXXX XXXXh		B8	12FC 544Dh	318526541

Example 4

```
DOTP2  .M2   B2,B5,B9:B8
```

	Before instruction			4 cycles after instruction	
B2	1234 3497h	4660 13463	B2	1234 3497h	4660 13463
B5	21FF 50A7h	8703 20647	B5	21FF 50A7h	8703 20647
B9:B8	XXXX XXXXh	XXXX XXXXh	B9:B8	0000 0000h	12FC 544Dh
					318526541

DOTPN2 *Dot Product With Negate, Signed Packed 16-Bit*

Syntax

DOTPN2 (.unit) *src1*,*src2, dst*
.unit = .M1, .M2

Opcode map field used...	For operand type...	Unit	Opfield
src1 *src2* *dst*	s2 xs2 int	.M1, .M2	01001

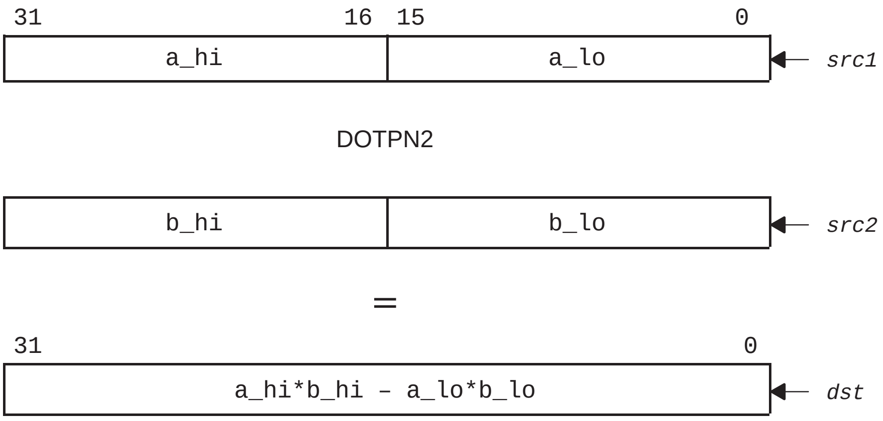
Opcode

31 29	28	27 23	22 18	17 13	12	11	10 6	5 2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	0	*op*	1 1 0 0	*s*	*p*
3	1	5	5	5	1	1	5	4	1	1

Description

The **DOTPN2** instruction returns the dot-product between two pairs of signed, packed 16-bit values where the second product is negated. The values in *src1* and *src2* are treated as signed, packed 16-bit quantities. The signed result is written to a single 32-bit register.

The product of the lower half-words of *src1* and *src2* is subtracted from the product of the upper half-words of *src1* and *src2*. The result is then written to *dst*.



Execution

```
if (cond)    {
             (msb16(src1)  x  msb16(src2))  –
             (lsb16(src1)  x  lsb16(src2))  →  dst
             }
else  nop
```

Note that unlike **DOTP2**, no overflow case exists for this instruction.

Pipeline

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

Instruction Type Four-cycle

Delay Slots 3

See Also **DOTP2**

Example 1

```
DOTPN2  .M1   A5,A6,A8
```

	Before instruction			**4 cycles after instruction**	
A5	3629 274Ah	13865 10058	A5	3629 274Ah	13865 10058
A6	325C 8036h	12892 –32714	A6	325C 8036h	12892 –32714
A8	XXXX XXXXh		A8	1E44 2F20h	507784992

Example 2

```
DOTPN2  .M2   B2,B5,B8
```

```
        Before instruction                       4 cycles after instruction

B2   3FF6 5010h   16374 20496          B2   3FF6 5010h   16374 20496

B5   B1C3 0244h   –20029 580           B5   B1C3 0244h   –20029 580

B8   XXXX XXXXh                        B8   EBBE 6A22h   –339842526
```

DOTPNRSU2

Dot Product With Negate, Shift and Round, Signed by Unsigned Packed 16-Bit

Syntax

DOTPNRSU2 (.unit) *src1*,*src2, dst*
.unit = .M1, .M2

Opcode map field used...	For operand type...	Unit	Opfield
src1	s2	.M1, .M2	00111
src2	xu2		
dst	int		

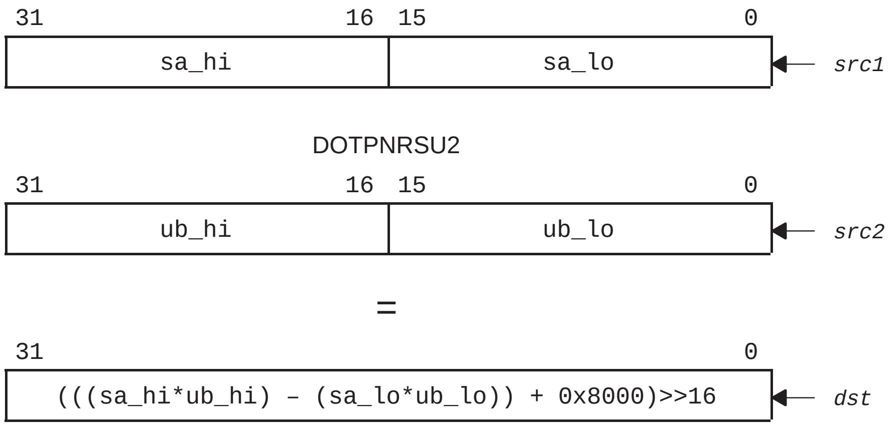
Opcode

31 29	28	27 23	22 18	17 13	12	11	10 6	5			2	1	0
creg	z	*dst*	*src2*	*src1*	x	0	*op*	1	1	0	0	s	p
3	1	5	5	5	1	1	5	4				1	1

Description

The **DOTPNRSU2** instruction returns the dot-product between two pairs of packed 16-bit values, where the second product is negated. This instructiion takes the result of the dot-product and performs an additional round and shift step. The values in *src1* are treated as signed, packed 16-bit quantities; whereas the values in *src2* are treated as unsigned, packed 16-bit quantities. The results are written to *dst*.

The product of the lower half-words of *src1* and *src2* is subtracted from the product of the upper half-words of *src1* and *src2*. The value 2^{15} is then added to this sum, producing an intermediate 32-bit result. The intermediate result is signed shifted right by 16, producing a rounded, shifted result that is sign extended and placed in *dst*.



Note:

The intermediate results of **DOTPNRSU2** are only maintained to 32-bit precision, thus overflow may occur during the rounding step.

Execution

```
if (cond)    {
                 int = (smsb16(src1) x umsb16(src2))
                     – (slsb16(src1) x ulsb16(src2)) + 0x8000;
                 int >> 16 → dst
             }
else nop
```

Pipeline

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

Instruction Type Four-cycle

Delay Slots 3

See Also **DOTP2, DOTPN2, DOTPRSU2**

Example 1

```
DOTPNRSU2  .M1   A5, A6, A8
```

	Before instruction			4 cycles after instruction	
A5	3629 274Ah	13865 10058 signed	A5	3629 274Ah	13865 10058 signed
A6	325C 8036h	12892 32822 unsigned	A6	325C 8036h	12892 32822 unsigned
A8	XXXX XXXXh		A8	FFFF F6FAh	–2310 signed

Example 2

```
DOTPNRSU2  .M2   B2, B5, B8
```

	Before instruction			4 cycles after instruction	
B2	3FF6 5010h	16374 20496 signed	B2	3FF6 5010h	16374 20496 signed
B5	B1C3 0244h	45507 580 unsigned	B5	B1C3 0244h	45507 580 unsigned
B8	XXXX XXXXh		B8	0000 2BB4h	11188 signed

DOTPNRUS2 *Dot Product With Negate, Shift and Round, Unsigned by Signed Packed 16-Bit (Pseudo-Operation)*

Syntax

DOTPNRUS2 (.unit) *src2*, *src1, dst*
.unit = .M1, .M2

Opcode map field used...	For operand type...	Unit	Opfield
src1	s2	.M1, .M2	00111
src2	xu2		
dst	int		

Opcode

See **DOTPNRSU2** instruction.

Description

The **DOTPNRUS2** pseudo-operation performs the dot-product between two pairs of packed 16-bit values, where the second product is negated. This instruction takes the result of the dot-product and performs an additional round and shift step. The values in *src1* are treated as signed, packed 16-bit quantities, whereas the values in *src2* are treated as unsigned, packed 16-bit quantities. The results are written to *dst*. The assembler uses the **DOTPNRSU2** *src1, src2, dst* instruction to perform this task.

The product of the lower half-words of *src1* and *src2* is subtracted from the product of the upper half-words of *src1* and *src2*. The value 2^{15} is then added to this sum, producing an intermediate 32-bit result. The intermediate result is signed shifted right by 16, producing a rounded, shifted result that is sign extended and placed in *dst*.

Execution

```
if (cond)     {
              int = (umsb16(src2)  x  smsb16(src1))
                    – (ulsb16(src2)  x  lsb16(src1)) + 0x8000;
              int >> 16 → dst
              }
else nop
```

Pipeline

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

Instruction Type Four-cycle

Delay Slots 3

See Also **DOTP2, DOTPN2, DOTPNRSU2, DOTPRUS2**

DOTPRSU2 *Dot Product With Shift and Round, Signed by Unsigned Packed 16-Bit*

Syntax

DOTPRSU2 (.unit) *src1*,*src2, dst*
.unit = .M1, .M2

Opcode map field used...	For operand type...	Unit	Opfield
src1	s2	.M1, .M2	01101
src2	xu2		
dst	int		

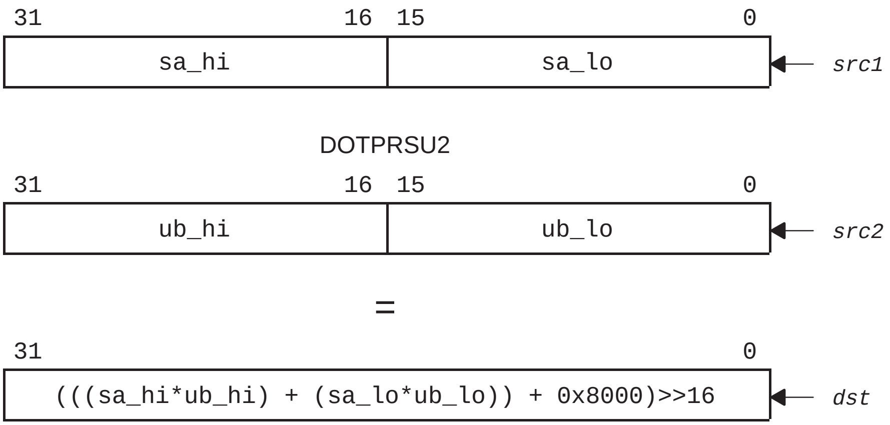
Opcode

31 29	28	27 23	22 18	17 13	12	11	10 6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	0	*op*	1	1	0	0	*s*	*p*
3	1	5	5	5	1	1	5	4				1	1

Description

The **DOTPRSU2** instruction returns the dot-product between two pairs of packed 16-bit values. This instruction takes the result of the dot-product and performs an additional round and shift step. The values in *src1* are treated as signed packed 16-bit quantities, whereas the values in *src2* are treated as unsigned packed 16-bit quantities. The results are written to *dst*.

The product of the lower half-words of *src1* and *src2* is added to the product of the upper half-words of *src1* and *src2*. The value 2^{15} is then added to this sum, producing an intermediate 32-bit result. The intermediate result is signed shifted right by 16, producing a rounded, shifted result that is sign extended and placed in *dst*.



Note:

The intermediate results of **DOTPRSU2** are only maintained to 32-bit precision, and so overflow may occur during the rounding step.

Execution

```
if (cond)    {
                 int = (smsb16(src1) x umsb16(src2)) +
                 (slsb16(src1) x ulsb16(src2)) + 0x8000;
                 int  >> 16 → dst
                 }
else  nop
```

Pipeline

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

Instruction Type Four-cycle

Delay Slots 3

See Also **DOTP2, DOTPN2, DOTPNRSU2**

Example 1

```
DOTPRSU2  .M1   A5, A6, A8
```

	Before instruction			4 cycles after instruction	
A5	3629 274Ah	13865 10058 signed	A5	3629 274Ah	13865 10058 signed
A6	325C 8036h	12892 32822 unsigned	A6	325C 8036h	12892 32822 unsigned
A8	XXXX XXXXh		A8	0000 1E55	7765 signed

Example 2

```
DOTPRSU2  .M2   B2, B5, B8
```

	Before instruction			4 cycles after instruction	
B2	B1C3 0244h	–20029 580 signed	B2	B1C3 0244h	–20029 580 signed
B5	3FF6 5010h	16374 20496 unsigned	B5	3FF6 5010h	16374 20496 unsigned
B8	XXXX XXXXh		B8	FFFF ED29	–4823 signed

DOTPRUS2

Dot Product With Shift and Round, Unsigned by Signed Packed 16-Bit (Pseudo-Operation)

Syntax

DOTPRUS2 (.unit) *src2*, *src1, dst*
.unit = .M1, .M2

Opcode map field used...	For operand type...	Unit	Opfield
src1 *src2* *dst*	s2 xu2 int	.M1, .M2	01101

Opcode

See **DOTPRSU2** instruction.

Description

The **DOTPRUS2** pseudo-operation returns the dot-product between two pairs of packed 16-bit values. This instruction takes the result of the dot-product, and performs an additional round and shift step. The values in *src1* are treated as signed packed 16-bit quantities, whereas the values in *src2* are treated as unsigned packed 16-bit quantities. The results are written to *dst*. The assembler uses the **DOTPRSU2** *src1, src2, dst* instruction to perform this task.

The product of the lower half-words of *src1* and *src2* is added to the product of the upper half-words of *src1* and *src2*. The value 2^{15} is then added to this sum, producing an intermediate 32-bit result. The intermediate result is signed shifted right by 16, producing a rounded, shifted result that is sign extended and placed in *dst*.

Execution

```
if (cond)     {
              int = (umsb16(src2) x smsb16(src1)) + (ulsb16(src2)
                        x slsb16(src1)) + 0 x 8000;
              int  >> 16 → dst
              }
else  nop
```

Pipeline

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

Instruction Type Four-cycle

Delay Slots 3

See Also **DOTP2, DOTPN2, DOTPNRUS2, DOTPRSU2**

DOTPSU4 *Dot Product, Signed by Unsigned Packed 8-Bit*

Syntax

DOTPSU4 (.unit) *src1*,*src2, dst*
.unit = .M1, .M2

Opcode map field used...	For operand type...	Unit	Opfield
src1	s4	.M1, .M2	00010
src2	xu4		
dst	int		

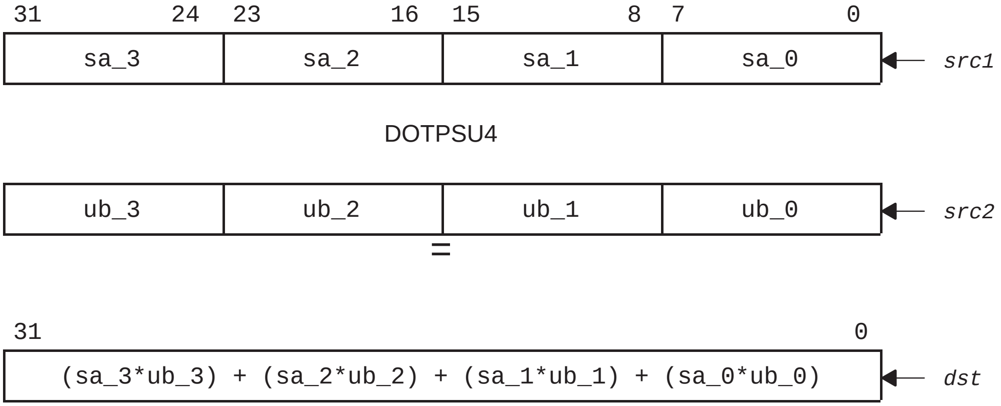
Opcode

31 29	28	27 23	22 18	17 13	12	11	10 6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	0	*op*	1	1	0	0	*s*	*p*
3	1	5	5	5	1	1	5	4				1	1

Description

The **DOTPSU4** instruction returns the dot-product between four sets of packed 8-bit values. The values in *src1* are treated as signed packed 8-bit quantities, whereas the values in *src2* are treated as unsigned 8-bit packed data. The signed result is written into *dst*.

For each pair of 8-bit quantities in *src1* and *src2*, the signed 8-bit value from *src1* is multiplied with the unsigned 8-bit value from *src2*. The four products are summed together, and the resulting dot product is written as a signed 32-bit result to *dst*.



Execution

```
if (cond)    {
                  (sbyte0(src1) x ubyte0(src2)) +
                  (sbyte1(src1) x ubyte1(src2)) +
                  (sbyte2(src1) x ubyte2(src2)) +
                  (sbyte3(src1) x ubyte3(src2)) → dst
                  }
else  nop
```

Pipeline

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

Instruction Type Four-cycle

Delay Slots 3

See Also **DOTPU4**

Example 1

```
DOTPSU4  .M1   A5, A6, A8
```

	Before instruction			**4 cycles after instruction**	
A5	6A 32 11 93h	106 50 17 –109 signed	A5	6A 32 11 93h	106 50 17 –109 signed
A6	B1 74 6C A4h	177 116 108 164 unsigned	A6	B1 74 6C A4h	177 116 108 164 unsigned
A8	XXXX XXXXh		A8	0000 214Ah	8522 signed

Example 2

```
DOTPSU4  .M2   B2, B5, B8
```

	Before instruction			4 cycles after instruction	
B2	3F F6 50 10h	63 -10 80 16 signed	B2	3F F6 50 10h	63 -10 80 16 signed
B5	C3 56 02 44h	195 86 2 68 unsigned	B5	C3 56 02 44h	195 86 2 68 unsigned
B8	XXXX XXXXh		B8	0000 3181h	12673 signed

DOTPUS4 *Dot-Product, Unsigned by Signed Packed 8-Bit (Pseudo-Operation)*

Syntax

DOTPUS4 (.unit) *src2, src1*, *dst*
.unit = .M1, .M2

Opcode map field used...	For operand type...	Unit	Opfield
src1	s4	.M1, .M2	00010
src2	xu4		
dst	int		

Opcode

See **DOTPSU4** instruction.

Description

The **DOTPUS4** pseudo-operation returns the dot-product between four sets of packed 8-bit values. The values in *src1* are treated as signed packed 8-bit quantities, whereas the values in *src2* are treated as unsigned 8-bit packed data. The signed result is written into *dst*. The assembler uses the **DOTPSU4** *src1*, *src2*, *dst* instruction to perform this task.

For each pair of 8-bit quantities in *src1* and *src2*, the signed 8-bit value from *src1* is multiplied with the unsigned 8-bit value from *src2*. The four products are summed together, and the resulting dot-product is written as a signed 32-bit result to *dst*.

Execution

```
if (cond)    {
                (ubyte0(src2) x sbyte0(src1)) +
                (ubyte1(src2) x sbyte1(src1)) +
                (ubyte2(src2) x sbyte2(src1)) +
                (ubyte3(src2) x sbyte3(src1)) → dst
                }
else  nop
```

Pipeline

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

Instruction Type Four-cycle

Delay Slots 3

See Also **DOTPU4, DOTPSU4**

DOTPU4 *Dot Product, Unsigned Packed 8-Bit*

Syntax

DOTPU4 (.unit) *src1*, *src2, dst*
.unit = .M1, .M2

Opcode map field used...	For operand type...	Unit	Opfield
src1 *src2* *dst*	u4 xu4 uint	.M1, .M2	00110

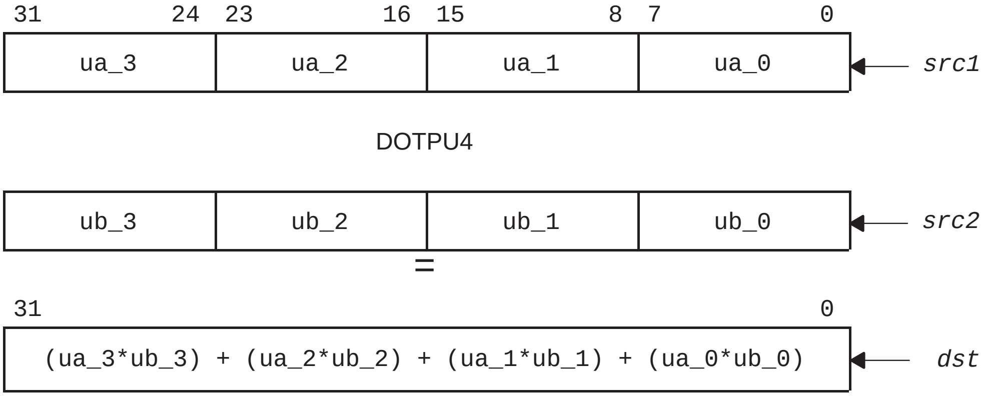
Opcode

31 29	28	27 23	22 18	17 13	12	11	10 6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	0	*op*	1	1	0	0	*s*	*p*
3	1	5	5	5	1	1	5	4				1	1

Description

The **DOTPU4** instruction returns the dot-product between four sets of packed 8-bit values. The values in both *src1* and *src2* are treated as unsigned, 8-bit packed data. The unsigned result is written into *dst*.

For each pair of 8-bit quantities in *src1* and *src2*, the unsigned 8-bit value from *src1* is multiplied with the unsigned 8-bit value from *src2*. The four products are summed together, and the resulting dot-product is written as a 32-bit result to *dst*.



Execution

```
if (cond)    {
             (ubyte0(src1) x ubyte0(src2)) +
             (ubyte1(src1) x ubyte1(src2)) +
             (ubyte2(src1) x ubyte2(src2)) +
             (ubyte3(src1) x ubyte3(src2)) → dst
             }
else nop
```

Pipeline

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

Instruction Type Four-cycle

Delay Slots 3

See Also **DOTPSU4**

Example `DOTPU4 .M1 A5, A6, A8`

```
     Before instruction                         4 cycles after instruction
A5   6A 32 11 93h   106 50 17 147          A5   6A 32 11 93h   106 50 17 147
                    unsigned                                   unsigned
A6   B1 74 6C A4h   177 116 108 164        A6   B1 74 6C A4h   177 116 108 164
                    unsigned               .                   unsigned
A8   XXXX XXXXh                            A8   0000 C54Ah     50506
                                                               unsigned
```

GMPY4 *Galois Field Multiply, Packed 8-Bit*

Syntax

GMPY4 (.unit) *src1*,*src2, dst*
.unit = .M1, .M2

Opcode map field used...	For operand type...	Unit	Opfield
src1	u4	.M1, .M2	10001
src2	xu4		
dst	u4		

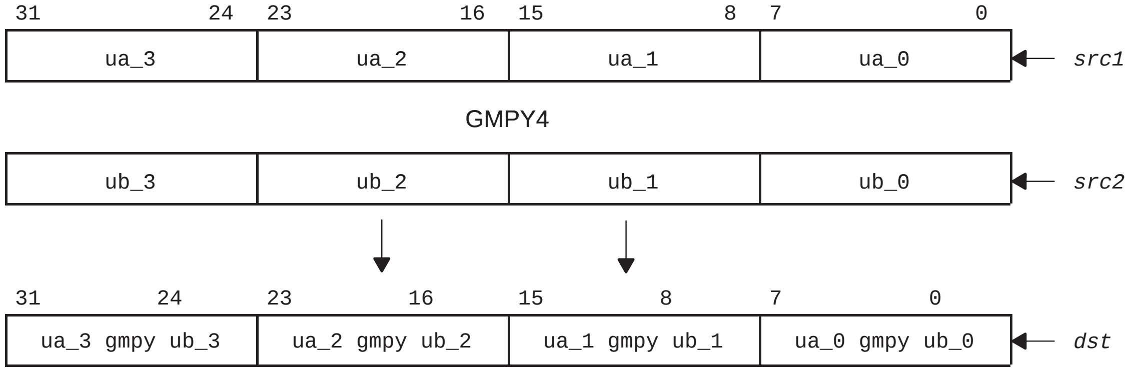
Opcode

31 29	28	27 23	22 18	17 13	12	11	10 6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	*x*	0	*op*	1	1	0	0	*s*	*p*
3	1	5	5	5	1	1	5	4				1	1

Description

The **GMPY4** instruction performs the Galois field multiply on four values in *src1* with four parallel values in *src2*. The four products are packed into dst. The values in both *src1* and *src2* are treated as unsigned, 8-bit packed data.

For each pair of 8-bit quantities in *src1* and *src2*, the unsigned, 8-bit value from *src1* is Galois field multiplied with the unsigned, 8-bit value from *src2*. The product of *src1* byte 0 and *src2* byte 0 is written to byte0 of *dst*. The product of *src1* byte 1 and *src2* byte 1 is written to byte1 of *dst*. The product of *src1* byte 2 and *src2* byte 2 is written to byte2 of *dst*. The product of *src1* byte 3 and *src2* byte 3 is written to the most significant byte in *dst*.



Note: `gmpy` represents the Galois Field Operation

The size and polynomial are controlled by the Galois Field Polynomial Generator Function Register in the control register file. All registers in the control register file can be written using the **MVC** instruction.

The default field generator polynomial is 0x1D, and the default size is 7. This setting is used for many communications standards.

Note that the **GMPY4** instruction is commutative, so:

```
GMPY4 .M1        A10,A12,A13

        ** is equivalent to **

GMPY4 .M1        A12,A10,A13
```

Execution

```
if (cond)    {
             (ubyte0(src1) gmpy ubyte0(src2)) → ubyte0(dst)
             (ubyte1(src1) gmpy ubyte1(src2)) → ubyte1(dst)
             (ubyte2(src1) gmpy ubyte2(src2)) → ubyte2(dst)
             (ubyte3(src1) gmpy ubyte3(src2)) → ubyte3(dst)
             }
else nop
```

Pipeline

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

Instruction Type Four-cycle

Delay Slots 3

See Also **MVC, XOR**

Example 1

GMPY4 .M1 A5,A6,A7; polynomial = 0x11d

	Before instruction			**4 cycles after instruction**	
A5	45 23 00 01h	69 35 0 1 unsigned	A5	45 23 00 01h	69 35 0 1 unsigned
A6	57 34 00 01h	87 52 0 1 unsigned	A6 .	57 34 00 01h	87 52 0 1 unsigned
A7	XXXX XXXXh		A7	72 92 00 01h	114 146 0 1 unsigned

Example 2

GMPY4 .M1 A5,A6,A7; field size is 256

	Before instruction			**4 cycles after instruction**	
A5	FF FE 02 1Fh	255 254 2 31 unsigned	A5	FF FE 02 1Fh	255 254 2 31 unsigned
A6	FF FE 02 01h	255 254 2 1 unsigned	A6 .	FF FE 02 01h	255 254 2 1 unsigned
A7	XXXX XXXXh	XXXX XXXXh	A7	E2 E3 04 1Fh	226 227 4 31 unsigned

LDDW — *Load Doubleword From Memory With an Unsigned Constant Offset or Register Offset*

Syntax

LDDW (.unit) *+*baseR[offsetR/ucst5]*, *dst*
.unit = .D1 or .D2

Opcode

31 29	28	27 23	22 18	17 13	12 9	8	7	6 4	3	2	1	0
creg	*z*	*dst*	*baseR*	*offsetR/ucst5*	*mode*	*r*	*y*	*ld/st*	1	0	*s*	*p*
3	1	4	5	5	4	1	1	3	2		1	1

Description

The **LDDW** instruction loads a 64-bit quantity from memory into a register pair *dst_o:dst_e.* Table 5–8 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*).

Both *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.

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-numbered register and the most significant 32 bits (containing the sign bit and exponent) are loaded into the next register (which is always odd-numbered 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 5–8 summarizes the address generation options supported.

Table 5–8. 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`

Before instruction

A1:A0	XXXX XXXXh	XXXX XXXXh	
B10	0000 0010h		16
mem 0x18	3333 3333h	4021 3333h	8.6

5 cycles after instruction

A1:A0	4021 3333h	3333 3333h	
B10	0000 0010h		16
mem 0x18	3333 3333h	4021 3333h	

Little-endian mode

Example 2

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

Before instruction

A1:A0	XXXX XXXXh	XXXX XXXXh	
A10	0000 0010h		16
mem 0x18	4021 3333h	3333 3333h	8.6

1 cycle after instruction

A1:A0	XXXX XXXXh	XXXX XXXXh	
A10	0000 0018h		24
mem 0x18	4021 3333h	3333 3333h	

5 cycles after instruction

A1:A0	4021 3333h	3333 3333h	
A10	0000 0018h		24
mem 0x18	4021 3333h	3333 3333h	

Big-endian mode

LDNDW

Load Non-Aligned Double Word

Syntax

LDNDW (.unit) *mem*, dst*

.unit = .D1, .D2

Opcode map field used...	For operand type...	Unit	Opfield
baseR *offset* *dst*	uint uint ullong	.D1, .D2	
baseR *offset* *dst*	uint ucst5 ullong	.D1 .D2	

Opcode

31 29	28	27 24	23	22 18	17 13	12 9	8	7	6 4	3	2	1	0
creg	*z*	*dst*	*sc*	*baseR*	*offset*	*mode*	*r*	*y*	*ld/st*	0	1	*s*	*p*
3	1	4	1	5	5	4	1	1	3	2		1	1

Description

The **LDNDW** instruction loads a 64-bit quantity from memory into a register pair, *dst_o:dst_e*. The table below describes the addressing generator options. The LDNDW instruction may read a 64-bit value from any byte boundary. Thus alignment to a 64-bit boundary is not required. The memory effective 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*).

Both *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.

The **LDNDW** instruction supports both scaled offsets and non-scaled offsets. The *sc* field is used to indicate whether the *offsetR*/*ucst5* is scaled or not. If *sc* is 1 (scaled), the *offsetR/ucst5* is shifted left 3 bits before adding or subtracting from the *baseR*. If *sc* is 0 (non-scaled), the *offsetR/ucst5* is not shifted before adding or subtracting from the *baseR*. For the pre-increment, pre-decrement, positive offset, and negative offset address generator options, the result of the calculation is the address to be accessed in memory. For post-increment or post-decrement addressing, the value of *baseR* before the addition or subtraction is the address to be accessed from 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.

The *dst* field of the instruction selects a register pair, a consecutive even-numbered and odd-numbered register pair from the same register file. The instruction can be used to load a pair of 32-bit integers. The least significant 32 bits are loaded into the even-numbered register and the most significant 32 bits are loaded into the next register (which is always an odd-numbered register).

The *dst* can be in either register file, regardless of the .D unit or baseR or offsetR used. The s bit determines which file dst will be loaded into: s = 0 indicates dst will be in the A register file and s = 1 indicates dst will be loaded in the B register file.

Table 5–9. LDNDW 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

Note:

No other memory access may be issued in parallel with a non-aligned memory access. The other .D unit can be used in parallel as long as it is not performing a memory access.

Assembler Notes

When no bracketed register or constant is specified, the assembler defaults increments and decrements to 1 and offsets to 0. Loads that do no modification to the baseR can use the assembler syntax *R. Square brackets, [ ], indicate that the ucst5 offset is left-shifted by 3 for double word loads.

Parentheses, ( ), can be used to tell the assembler that the offset is a non-scaled offset.

For example, **LDNDW** (.unit) *+baseR (14) dst represents an offset of 14 bytes, and the assembler writes out the instruction with *offsetC* = 14 and *sc* = 0.

LDNDW (.unit) *+baseR [16] *dst* represents an offset of 16 double words, or 128 bytes, and the assembler writes out the instruction with *offsetC* = 16 and *sc* = 1.

Either brackets or parentheses must be typed around the specified offset if the optional offset parameter is used.

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

See Also **LDNW, STNDW, STNW**

Example 1

LDNDW .D1 *A0++, A3:A2

	Before instruction			1 cycle after instruction			5 cycles after instruction	
A0	0000 1001h		A0	0000 1009h		A0	0000 1009h	
A3:A2	XXXX XXXXh	XXXh	A3:A2	XXXX XXXXh	XXXX XXXh	A3:A2	5E1C 4F29h	A812 B6C5h
								little endian

Byte Memory Address	100C	100B	100A	1009	1008	1007	1006	1005	1004	1003	1002	1001	1000
Data Value	11	05	69	34	5E	1C	4F	29	A8	12	B6	C5	D4

Example 2

LDNDW .D1 *A0++, A3:A2

	Before instruction			1 cycle after instruction			5 cycles after instruction	
A0	0000 1003h		A0	0000 100Bh		A0	0000 100Bh	
A3:A2	XXXX XXXXh	XXXX XXXXh	A3:A2	XXXX XXXXh	XXXX XXXXh	A3:A2	6934 5E1Ch	4F29 A812h
								little endian

Byte Memory Address	100C	100B	100A	1009	1008	1007	1006	1005	1004	1003	1002	1001	1000
Data Value	11	05	69	34	5E	1C	4F	29	A8	12	B6	C5	D4

LDNW *Load Non-Aligned Word*

Syntax

LDNW (.unit) *mem*, dst*
.unit = .D1, .D2

Opcode map field used...	For operand type...	Unit	Opfield
baseR *offset* *dst*	uint uint int	.D1, .D2	
baseR *offset* *dst*	uint ucst5 int	.D1 .D2	

Opcode

31 29	28	27 23	22 18	17 13	12 9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*baseR*	*offset*	*mode*	*r*	*y*	0	1	1	0	1	*s*	*p*
3	1	5	5	5	4	1	1	3			2		1	1

Description

The **LDNW** instruction loads a 32-bit quantity from memory into a 32-bit register, *dst*. The table below describes the addressing generator options. The **LDNW** instruction may read a 32-bit value from any byte boundary. Thus alignment to a 32-bit boundary is not required. The memory effective 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.

Both *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.

The *offsetR/ucst5* is scaled by a left shift of 2 bits. After scaling, *offsetR/ucst5* is added to, or subtracted from, *baseR*. For the pre-increment, pre-decrement, positive offset, and negative offset address generator options, the result of the calculation is the address to be accessed in memory. For post-increment or post-decrement addressing, the value of *baseR* before the addition or subtraction is the address to be accessed from 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.

The *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. r is always zero.

Table 5–10. LDNW 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

Note:

No other memory access may be issued in parallel with a non-aligned memory access. The other .D unit can be used in parallel, as long as it is not doing a memory access.

Assembler Notes

When no bracketed register or constant is specified, the assembler defaults increments and decrements to 1 and offsets to 0. Loads that do no modification to the baseR can use the assembler syntax *R. Square brackets, [ ], indicate that the ucst5 offset is left-shifted by 2 for word loads.

Parentheses, ( ), can be used to tell the assembler that the offset is a non-scaled, constant offset. The assember right shifts the constant by 2 bits for word loads before using it for the ucst5 field. After scaling by the **LDNW** instruction, this results in the same constant offset as the assembler source if the least significant two bits are zeros.

For example, **LDNW** (.unit) *+baseR (12) dst represents an offset of 12 bytes (3 words), and the assembler writes out the instruction with ucst5 = 3.

LDNW (.unit) *+baseR [12] dst represents an offset of 12 words, or 48 bytes, and the assembler writes out the instruction with ucst5 = 12.

Either brackets or parentheses must be typed around the specified offset if the optional offset parameter is used.

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

See Also **LDNDW, STNDW, STNW**

Example 1

```
LDNW   .D1    *A0++, A2
```

	Before instruction		1 cycle after instruction		5 cycles after instruction	
A0	0000 1001h	A0	0000 1005h	A0	0000 1005h	
A2	XXXX XXXXh	A2	XXXX XXXXh	A2	A812 B6C5h	Little endian

Byte Memory Address	1007	1006	1005	1004	1003	1002	1001	1000
Data Value	1C	4F	29	A8	12	B6	C5	D4

Example 2

```
LDNW   .D1    *A0++, A2
```

	Before instruction		1 cycle after instruction		5 cycles after instruction	
A0	0000 1003h	A0	0000 1007h	A0	0000 1007h	
A2	XXXX XXXXh	A2	XXXX XXXXh	A2	4F29 A812h	Little endian

Byte Memory Address	1007	1006	1005	1004	1003	1002	1001	1000
Data Value	1C	4F	29	A8	12	B6	C5	D4

MAX2 *Maximum, Signed Packed 16-Bit*

Syntax

MAX2 (.unit) *src1*,*src2, dst*
.unit = .L1, .L2

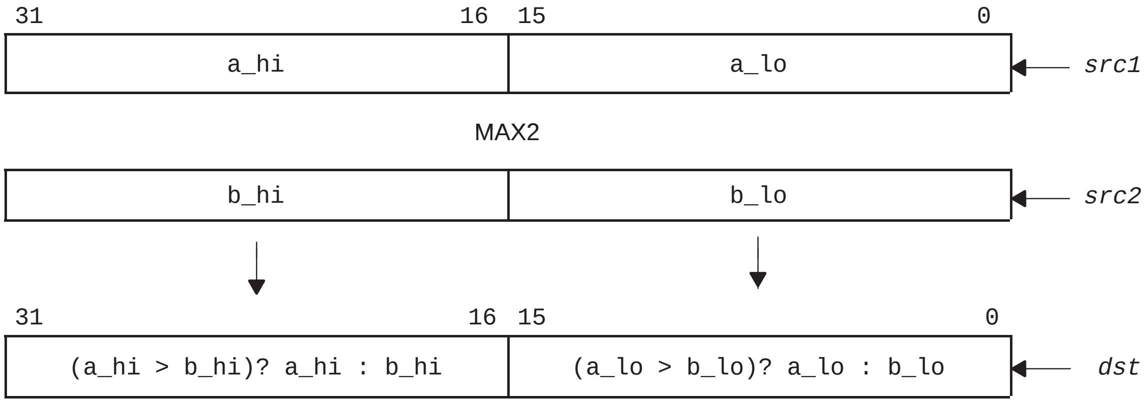
Opcode map field used...	For operand type...	Unit	Opfield
src1	s2	.L1, .L2	1000010
src2	xs2		
dst	s2		

Opcode

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

Description

The **MAX2** instruction performs a maximum operation on packed signed 16-bit values. For each pair of signed 16-bit values in *src1* and *src2*, **MAX2** places the larger value in the corresponding position in *dst*.



Execution

```
if (cond)    {
                 if (lsb16(src1) >= lsb16(src2), lsb16(src1) → lsb16(dst)
                      else lsb16(src2) → lsb16(dst);
                 if (msb16(src1) >= msb16(src2)), msb16(src1) → msb16(dst)
                      else  msb16(src2) → msb16(dst);
                 }
else  nop
```

Pipeline

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

Instruction Type Single-cycle

Delay Slots 0

See Also **MAXU4, MIN2, MINU4**

Example 1

```
MAX2  .L1   A2, A8, A9
```

	Before instruction			1 cycle after instruction	
A2	3789 F23Ah	14217 –3526	A2	3789 F23Ah	14217 –3526
A8	04B8 4975h	1208 18805	A8	04B8 4975h	1208 18805
A9	XXXX XXXXh		A9	3789 4975h	14217 18805

Example 2

```
MAX2  .L2X   A2, B8, B12
```

	Before instruction			1 cycle after instruction	
A2	0124 2451h	292 9297	A2	0124 2451h	292 9297
B8	01A6 A051h	422 –24495	B8	01A6 A051h	422 –24495
B12	XXXX XXXXh		B12	01A6 2451h	422 9297

MAXU4 *Maximum, Unsigned Packed 8-Bit*

Syntax

MAXU4 (.unit) *src1*,*src2, dst*
.unit = .L1, .L2

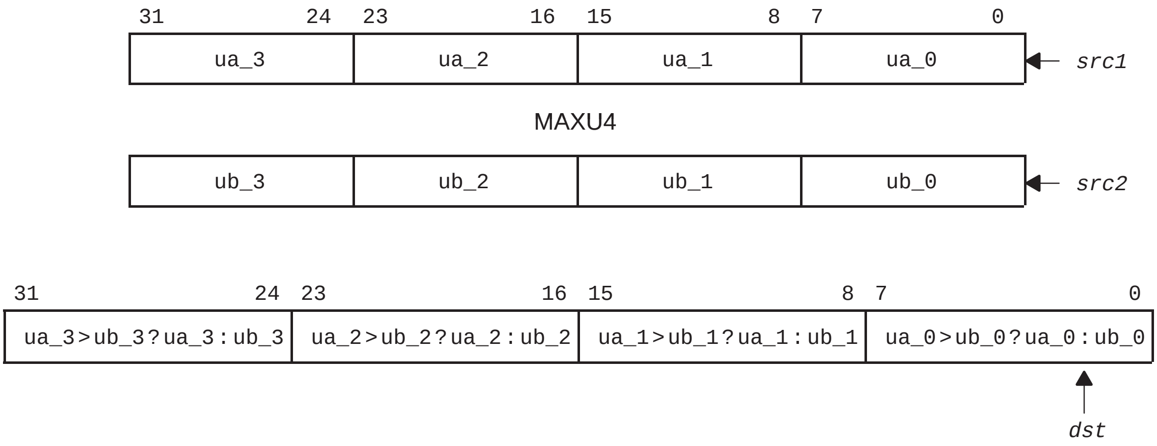
Opcode map field used...	For operand type...	Unit	Opfield
src1 *src2* *dst*	u4 xu4 u4	.L1, .L2	1000011

Opcode

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

Description

The **MAXU4** instruction performs a maximum operation on packed, unsigned 8-bit values. For each pair of unsigned 8-bit values in *src1* and *src2*, **MAXU4** places the larger value in the corresponding position in *dst*.



Execution

```
if (cond)    {
              if (ubyte0(src1) >= ubyte0(src2), ubyte0(src1) → ubyte0(dst)
                   else ubyte0(src2) → ubyte0(dst);
              if (ubyte1(src1) >= ubyte1(src2)), ubyte1(src1) → ubyte1(dst)
                   else ubyte1(src2) → ubyte1(dst);
              if (ubyte2(src1) >= ubyte2(src2)), ubyte2(src1) → ubyte2(dst)
                   else  ubyte2(src2) → ubyte2(dst);
              if (ubyte3(src1) >= ubyte3(src2)), ubyte3(src1) → ubyte3(dst)
                   else  ubyte3(src2) → ubyte3(dst);
              }
else  nop
```

Pipeline

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

Instruction Type Single-cycle

Delay Slots 0

See Also **MAX2, MIN2, MINU4**

Example 1

```
MAXU4  .L1  A2, A8, A9
```

```
        Before instruction                            1 cycle after instruction

A2  | 37 89 F2 3Ah | 55 137 242 58           A2  | 37 89 F2 3Ah | 55 137 242 58
                     unsigned                                     unsigned

A8  | 04 B8 49 75h | 4 184 73 117            A8  | 04 B8 49 75h | 4 184 73 117
                     unsigned                                     unsigned

A9  | XXXX XXXXh   |                         A9  | 37 B8 F2 75h | 55 184 242 117
                                                                  unsigned
```

Example 2

```
MAXU4  .L2X   A2, B8, B12
```

	Before instruction			1 cycle after instruction	
A2	01 24 24 B9h	1 36 36 185 unsigned	A2	01 24 24 B9h	1 36 36 185 unsigned
B8	01 A6 A0 51h	1 166 160 81 unsigned	B8	01 A6 A0 51h	1 166 160 81 unsigned
B12	XXXX XXXXh		B12	01 A6 A0 B9h	1 166 160 185 unsigned

MIN2 *Minimum, Signed Packed 16-Bit*

Syntax

MIN2 (.unit) *src1*,*src2, dst*
.unit = .L1, .L2

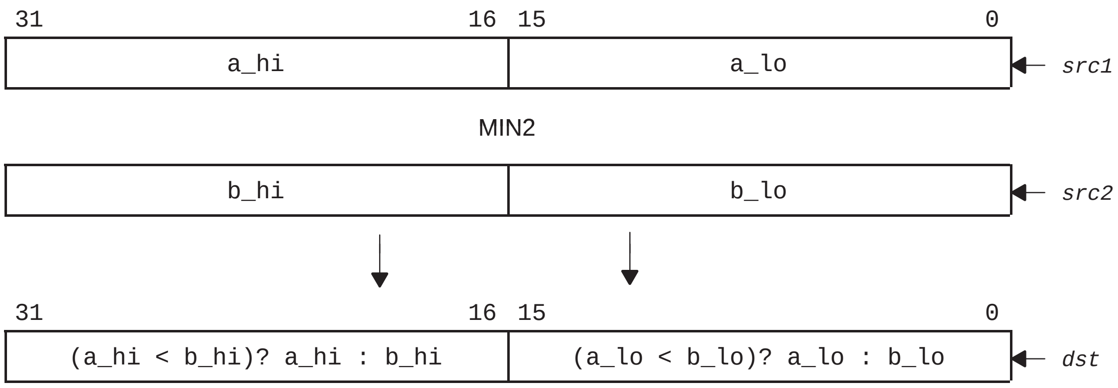
Opcode map field used...	For operand type	Unit	Opfield
src1	s2	.L1, .L2	1000001
src2	xs2		
dst	s2		

Opcode

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

Description

The **MIN2** instruction performs a minimum operation on packed, signed 16-bit values. For each pair of signed 16-bit values in *src1* and *src2*, MAX2 places the smaller value in the corresponding position in *dst*.



Execution

```
if (cond)    {
               if (lsb16(src1) <= lsb16(src2), lsb16(src1) → lsb16(dst)
                   else lsb16(src2) → lsb16(dst);
               if (msb16(src1) <= msb16(src2)), msb16(src1) → msb16(dst)
                   else msb16(src2)→ msb16(dst);
               }
else  nop
```

Pipeline

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

Instruction Type Single-cycle

Delay Slots 0

See Also **MAX2, MAXU4, MINU4**

Example 1

```
MIN2  .L1   A2, A8, A9
```

	Before instruction			1 cycle after instruction	
A2	3789 F23Ah	14217 –3526	A2	3789 F23Ah	14217 –3526
A8	04B8 4975h	1208 18805	A8	04B8 4975h	1208 18805
A9	XXXX XXXX h		A9	04B8 F23Ah	1208 –3526

Example 2

```
MIN2  .L2X   A2, B8, B12
```

```
          Before instruction                       1 cycle after instruction
A2    | 0124 8003h  | 292 –32765          A2    | 0124 8003h | 292 –32765

B8    | 0A37 8001h  | 2615 –32767         B8    | 0A37 8001h | 2615 –32767

B12   | XXXX XXXX h |                     B12   | 0124 8001h | 292 –32767
```

MINU4 Minimum, Unsigned Packed 8-Bit

Syntax

MINU4 (.unit) *src1*,*src2, dst*
.unit = .L1, .L2

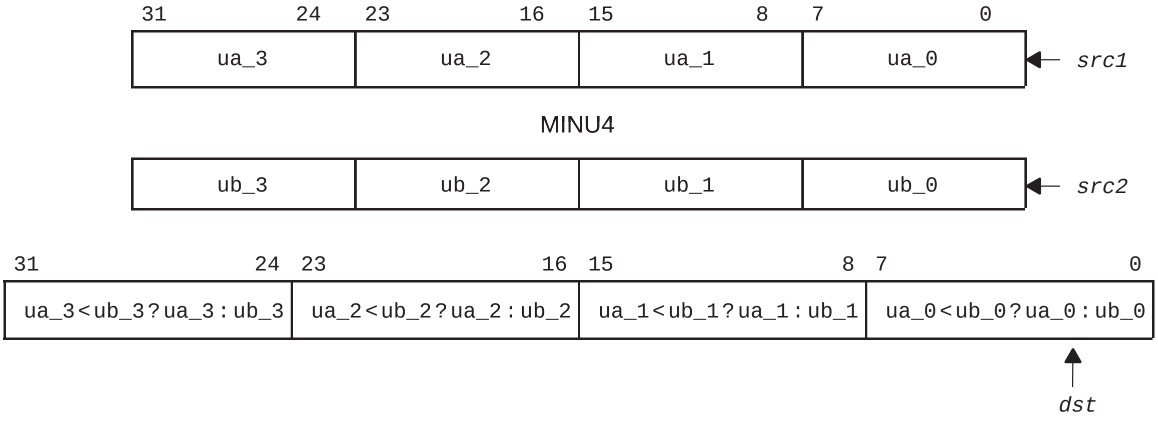
Opcode map field used...	For operand type...	Unit	Opfield
src1 *src2* *dst*	u4 xu4 u4	.L1, .L2	1001000

Opcode

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

Description

The **MINU4** instruction performs a minimum operation on packed, unsigned 8-bit values. For each pair of unsigned 8-bit values in *src1* and *src2*, **MAXU4** places the smaller value in the corresponding position in *dst*.



Execution

```
if (cond)    {
               if (ubyte0(src1) <= ubyte0(src2), ubyte0(src1) → ubyte0(dst)
                   else ubyte0(src2) → ubyte0(dst);
               if (ubyte1(src1) <= ubyte1(src2)), ubyte1(src1) → ubyte1(dst)
                   else  ubyte1(src2) → ubyte1(dst);
               if (ubyte2(src1) <= ubyte2(src2)), ubyte2(src1) → ubyte2(dst)
                   else  ubyte2(src2) → ubyte2(dst);
               if (ubyte3(src1) <= ubyte3(src2)), ubyte3(src1) → ubyte3(dst)
                   else  ubyte3(src2) → ubyte3(dst);
               }
else  nop
```

Pipeline

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

Instruction Type Single-cycle

Delay Slots 0

See Also **MAX2, MAXU4, MIN2**

Example 1

```
MINU4  .L1  A2, A8, A9
```

```
        Before instruction                              1 cycle after instruction

A2   | 37 89 F2 3Ah | 55 137 242 58          A2   | 37 89 F2 3Ah | 55 137 242 58
                      unsigned                                      unsigned

A8   | 04 B8 49 75h | 4 184 73 117           A8   | 04 B8 49 75h | 4 184 73 117
                      unsigned                                      unsigned

A9   | XXXX XXXXh   |                        A9   | 04 89 49 3Ah | 4 137 73 58
                                                                    unsigned
```

Example 2 MINU4 .L2 B2, B8, B12

	Before instruction			1 cycle after instruction	
B2	01 24 24 B9h	1 36 36 185 unsigned	B2	01 24 24 B9h	1 36 36 185 unsigned
B8	01 A6 A0 51h	1 166 160 81 unsigned	B8	01 A6 A0 51h	1 166 160 81 unsigned
B12	XXXX XXXXh		B12	01 24 24 51h	1 36 36 81 unsigned

MPY2 *Multiply Signed by Signed, Packed 16-Bit*

Syntax

MPY2 (.unit) *src1*,*src2, dst*
.unit = .M1, .M2

Opcode map field used...	For operand type...	Unit	Opfield
src1 *src2* *dst*	s2 xs2 ullong	.M1, .M2	00000

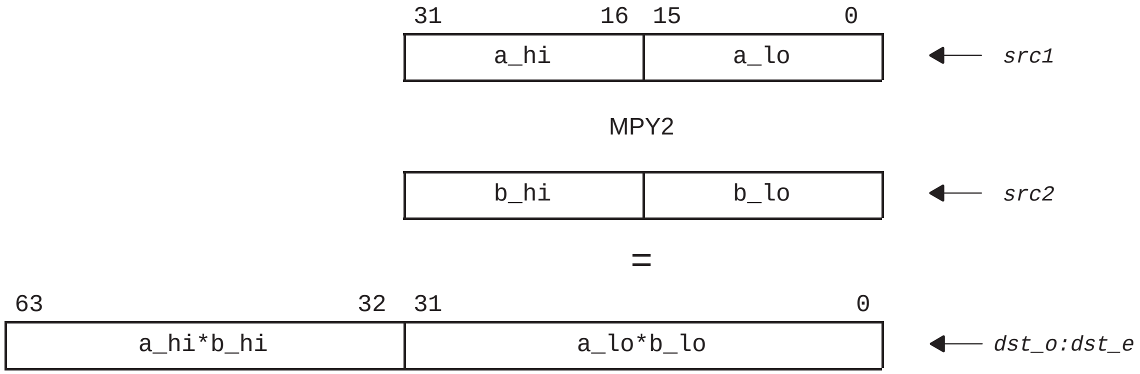
Opcode

31 29	28	27 23	22 18	17 13	12	11	10 6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	0	*op*	1	1	0	0	*s*	*p*
3	1	5	5	5	1	1	5	4				1	1

Description

The **MPY2** instruction performs two 16-bit by 16-bit multiplications between two pairs of signed, packed 16-bit values. The values in *src1* and *src2* are treated as signed, packed 16-bit quantities. The 32-bit results are written into a 64-bit register pair.

The product of the lower half-words of *src1* and *src2* is written to the even destination register, *dst_e*. The product of the upper half-words of *src1* and *src2* is written to the odd destination register, *dst_o*.



This instruction helps reduce the number of instructions required to perform two 16-bit by 16-bit multiplies on both the lower and upper halves of two registers.

The following code:

```
  MPY    .M1     A0, A1, A2
  MPYH   .M1     A0, A1, A3
```

may be replaced by:

```
  MPY2   .M1     A0, A1, A3:A2
```

Execution

```
if (cond)    {
                 (lsb16(src1) x lsb16(src2)) → dst_e
                 (msb16(src1) x msb16(src2)) → dst_o
                 }
else  nop
```

Pipeline

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

Instruction Type Four-cycle

Delay Slots 3

See Also **MPYSU4**

Example 1

```
MPY2  .M1   A5,A6, A9:A8
```

	Before instruction			4 cycles after instruction	
A5	6A32 1193h	27186 4499	A5	6A32 1193h	27186 4499
A6	B174 6CA4h	–20108 27812	A6	B174 6CA4h	–20108 27812
A9:A8	XXXX XXXXh	XXXX XXXXh	A9:A8	DF6A B0A8h	0775 462h
				–546,656,088	125,126,188

Example 2 MPY2 .M2 B2, B5, B9:B8

	Before instruction			4 cycles after instruction	
B2	1234 3497h	4660 13463	B2	1234 3497h	4660 13463
B5	21FF 50A7h	8703 20647	B5	21FF 50A7h	8703 20647
B9:B8	XXXX XXXXh	XXXX XXXXh	B9:B8	026A D5CCh	1091 7E81h
				40,555,980	277,970,561

MPYHI *Multiply 16 MSB x 32-Bit Into 64-Bit Result*

Syntax

MPYHI (.unit) *src1*,*src2, dst*
.unit = .M1, .M2

Opcode map field used...	For operand type	Unit	Opfield
src1 *src2* *dst*	int xint sllong	.M1, .M2	10100

Opcode

31 29	28	27 23	22 18	17 13	12	11	10 6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	0	*op*	1	1	0	0	*s*	*p*
3	1	5	5	5	1	1	5	4				1	1

Description

The **MPYHI** instruction performs a 16-bit by 32-bit multiply. The upper half of *src1* is used as a 16-bit signed input. The value in *src2* is treated as a 32-bit signed value. The result is written into the lower 48 bits of a 64-bit register pair, *dst_o:dst_e*, and sign extended to 64 bits.

Execution

```
if (cond)    {
               ((msb16 (src1)) x src2) → dst_o:dst_e
             }
else nop
```

Pipeline

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

Instruction Type Four-cycle

Delay Slots 3

See Also **MPYLI**

Example 1 `MPYHI  .M1   A5,A6,A9:A8`

	Before instruction			**4 cycles after instruction**	
A5	6A32 1193h	27186	A5	6A32 1193h	27186
A6	B174 6CA4h	–1,317,770,076	A6	B174 6CA4h	–1,317,770,076
A9:A8	XXXX XXXXh	XXXX XXXXh	A9:A8	FFFF DF6Ah	DDB9 2008h
				–358,248,997,286,136	

Example 2 `MPYHI   .M2   B2,B5,B9:B8`

```
               Before instruction                      4 cycles after instruction
B2     | 1234 3497h | 4660               B2     | 1234 3497h |  4660

B5     | 21FF 50A7h | 570,380,455        B5     | 21FF 50A7h |  570,380,455

B9:B8  | XXXX XXXXh | XXXX XXXXh         B9:B8  | 0000 026Ah | | DB88 1FECh    |

                                                         2,657,972,920,300
```

MPYHIR *Multiply 16 MSB x 32-Bit, Shifted by 15 to Produce a Rounded 32-Bit Result*

Syntax

MPYHIR (.unit) *src1*,*src2, dst*
.unit = .M1, .M2

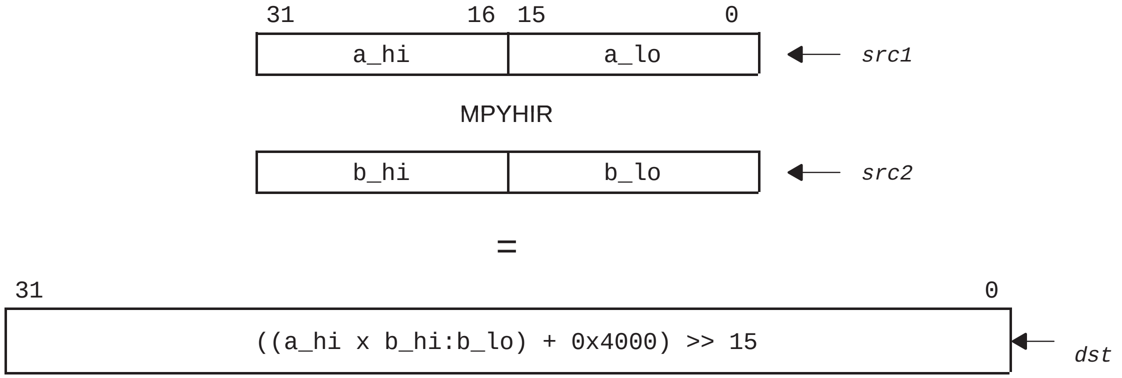
Opcode map field used...	For operand type	Unit	Opfield
src1	int	.M1, .M2	10000
src2	xint		
dst	int		

Opcode

31 29	28	27 23	22 18	17 13	12	11	10 6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	0	*op*	1	1	0	0	*s*	*p*
3	1	5	5	5	1	1	5	4				1	1

Description

The **MPYHIR** instruction performs a 16-bit by 32-bit multiply. The upper half of *src1* is treated as a 16-bit signed input. The value in *src2* is treated as a 32-bit signed value. The product is then rounded to a 32-bit result by adding the value 2^{14} and then this sum is right shifted by 15. The lower 32 bits of the result are written into *dst*.



Execution

```
if (cond)    {
             lsb32(((msb16(src1) x (src2)) + 0x4000)>>15) → dst
             }
else nop
```

Pipeline

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

Instruction Type Four-cycle

Delay Slots 3

See Also **MPYLIR**

Example

```
MPYHIR  .M2   B2,B5,B9
```

	Before instruction			4 cycles after instruction	
B2	1234 3497h	4660	B2	1234 3497h	4660
B5	21FF 50A7h	570,380,455	B5	21FF 50A7h	570,380,455
B9	XXXX XXXXh		B9	04D5 B710h	81,114,896

MPYIH *Multiply 32 x High 16-Bit Into 64-Bit Result (Pseudo-Operation)*

Syntax

MPYIH (.unit) *src2, src1*, *dst*
.unit = .M1, .M2

Opcode map field used...	For operand type	Unit	Opfield
src1 *src2* *dst*	int xint sllong	.M1, .M2	10100

Opcode See **MPYHI** instruction.

Description The **MPYIH** pseudo-operation performs a 16-bit by 32-bit multiply. The upper half of *src1* is used as a 16-bit signed input. The value in *src2* is treated as a 32-bit signed value. The result is written into the lower 48 bits of a 64-bit register pair, *dst_o:dst_e*, and sign extended to 64 bits. The assembler uses the **MPYHI** *src1*, *src2*, *dst* instruction to perform this task.

Execution

```
if (cond)    {
             ($src2$ x msb16 ($src1$)) → $dst$_o:$dst$_e
             }
else nop
```

Pipeline

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

Instruction Type Four-cycle

Delay Slots 3

See Also **MPYHI, MPYIL**

MPYIHR — *Multiply 32-Bit X High 16-bit, Shifted by 15 to Produce a Rounded 32-Bit Result (Pseudo-Operation)*

Syntax

MPYIHR (.unit) *src2*, *src1, dst*
.unit = .M1, .M2

Opcode map field used...	For operand type	Unit	Opfield
src1 *src2* *dst*	int xint int	.M1, .M2	10000

Opcode

See **MPYHIR** instruction.

Description

The **MPYIHR** pseudo-operation performs a 16-bit by 32-bit multiply. The upper half of *src1* is treated as a 16-bit signed input. The value in *src2* is treated as a 32-bit signed value. The product is then rounded to a 32-bit result by adding the value 2^{14} and then this sum is right shifted by 15. The lower 32 bits of the result are written into *dst*. The assembler uses the **MPYHIR** *src1*, *src2*, *dst* instruction to perform this operation.

Execution

```
if (cond)    {
             lsb32((((src2) x msb16(src1)) + 0x4000) >> 15) → dst
             }
else nop
```

Pipeline

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

Instruction Type

Four-cycle

Delay Slots

3

See Also

MPYHIR, MPYILR

MPYIL *Multiply 32 x Low 16-Bit Into 64-Bit Result (Pseudo-Operation)*

Syntax

MPYIL (.unit) *src2, src1*, *dst*
.unit = .M1, .M2

Opcode map field used...	For operand type	Unit	Opfield
src1 *src2* *dst*	int xint sllong	.M1, .M2	10101

Opcode

See **MPYLI** instruction.

Description

The **MPYIL** pseudo-operation performs a 16-bit by 32-bit multiply. The lower half of *src1* is used as a 16-bit signed input. The value in *src2* is treated as a 32-bit signed value. The result is written into the lower 48 bits of a 64-bit register pair, *dst_o:dst_e*, and sign extended to 64 bits. The assembler uses the **MPYLI** *src1*, *src2*, *dst* instruction to perform this operation.

Execution

```
if (cond)    {
             (($src2$) x lsb16($src1$)) → $dst\_o{:}dst\_e$
             }
else nop
```

Pipeline

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

Instruction Type

Four-cycle

Delay Slots

3

See Also

MPYIH, MPYLI

MPYILR

Multiply 32-Bit x Low 16, Shifted by 15 to Produce a Rounded 32-Bit Result (Pseudo-Operation)

Syntax

MPYILR (.unit) *src2*, *src1, dst*
.unit = .M1, .M2

Opcode map field used...	For operand type	Unit	Opfield
src1 *src2* *dst*	int xint int	.M1, .M2	01110

Opcode

See **MPYLIR** instruction.

Description

The **MPYILR** pseudo-operation performs a 16-bit by 32-bit multiply. The lower half of *src1* is used as a 16-bit signed input. The value in *src2* is treated as a 32-bit signed value. The product is then rounded to a 32-bit result by adding the value 2^{14} and then this sum is right shifted by 15. The lower 32 bits of the result are written into *dst.* The assembler uses a **MPYLIR** *src1*, *src2*, *dst* instruction to perform this operation.

Execution

```
if (cond)    {
             lsb32((((src2) x lsb16(src1)) + 0x4000) >> 15) → dst
             }
else nop
```

Pipeline

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

Instruction Type

Four-cycle

Delay Slots

3

See Also

MPYIHR, MPYLIR

MPYLI *Multiply 16 LSB x 32-Bit Into 64-Bit Result*

Syntax

MPYLI (.unit) *src1*,*src2, dst*
.unit = .M1, .M2

Opcode map field used...	For operand type	Unit	Opfield
src1 *src2* *dst*	int xint sllong	.M1, .M2	10101

Opcode

31 29	28	27 23	22 18	17 13	12	11	10 6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	0	*op*	1	1	0	0	*s*	*p*
3	1	5	*5*	5	1	1	5	4				1	1

Description The **MPYLI** instruction performs a 16-bit by 32-bit multiply. The lower half of *src1* is used as a 16 bit signed input. The value in *src2* is treated as a 32-bit signed value. The result is written into the lower 48 bits of a 64-bit register pair, *dst_o:dst_e*, and sign extended to 64 bits.

Execution

```
if (cond)    {
                 (lsb16(src1) x (src2)) → dst_o : dst_e
                 }
else nop
```

Pipeline

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

Instruction Type Four-cycle

Delay Slots 3

See Also **MPYHI**

Example 1 `MPYLI .M1 A5,A6,A9:A8`

```
            Before instruction                          4 cycles after instruction

A5        | 6A32 1193h | 4499                     A5        | 6A32 1193h |  4499

A6        | B174 6CA4h | –1,317,770,076           A6        | B174 6CA4h |  –1,317,770,076
                                                .
A9:A8     | XXXX XXXXh | XXXX XXXXh               A9:A8     | FFFF FA9Bh | | A111 462Ch |

                                                                 –5,928,647,571,924
```

Example 2 `MPYLI .M2 B2,B5,B9:B8`

```
            Before instruction                          4 cycles after instruction

B2        | 1234 3497h | 13463                    B2        | 1234 3497h |  13463

B5        | 21FF 50A7h | 570,380,455              B5        | 21FF 50A7h |  570,380,455

B9:B8     | XXXX XXXXh | XXXX XXXXh               B9:B8     | 0000 06FBh | | E9FA 7E81 |

                                                                   7,679,032,065,665
```

MPYLIR *Multiply 16 LSB x 32-Bit, Shifted by 15 to Produce a Rounded 32-Bit Result*

Syntax

MPYLIR (.unit) *src1*,*src2, dst*
.unit = .M1, .M2

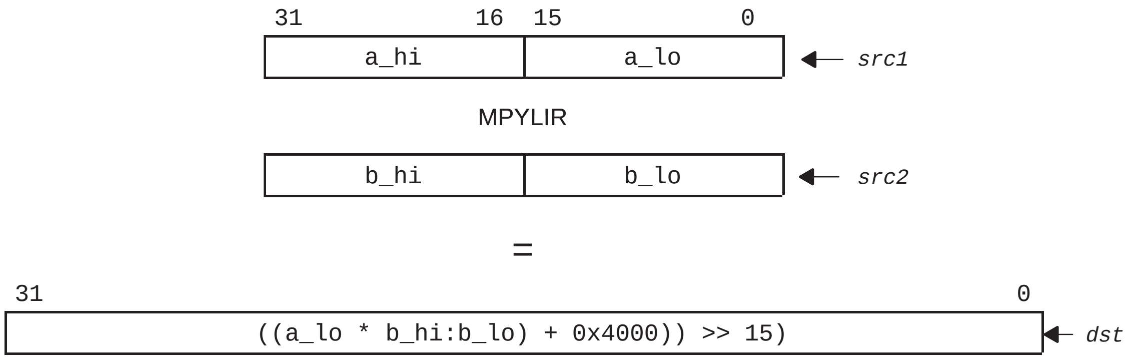
Opcode map field used...	For operand type	Unit	Opfield
src1	int	.M1, .M2	01110
src2	xint		
dst	int		

Opcode

31 29	28	27 23	22 18	17 13	12	11	10 6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	0	*op*	1	1	0	0	*s*	*p*
3	1	5	5	5	1	1	5	4				1	1

Description

The **MPYLIR** instruction performs a 16-bit by 32-bit multiply. The lower half of *src1* is treated as a 16-bit signed input. The value in *src2* is treated as a 32-bit signed value. The product is then rounded into a 32-bit result by adding the value 2^{14} and then this sum is right shifted by 15. The lower 32 bits of the result are written into *dst.*



Execution

```
if (cond)    {
             lsb32(((lsb16(src1) x (src2)) + 0x4000) >> 15) → dst
             }
else nop
```

Pipeline

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

Instruction Type Four-cycle

Delay Slots 3

See Also **MPYHIR**

Example

```
MPYLIR  .M2   B2,B5,B9
```

	Before instruction			4 cycles after instruction	
B2	1234 3497h	13463	B2	1234 3497h	13463
B5	21FF 50A7h	570,380,455	B5	21FF 50A7h	570,380,455
B9	XXXX XXXXh		B9	0DF7 D3F5h	234,345,461

MPYSU4 *Multiply Signed by Unsigned Packed, 8-Bit*

Syntax

MPYSU4 (.unit) *src1*,*src2, dst*

.unit = .M1, .M2

Opcode map field used...	For operand type	Unit	Opfield
src1 *src2* *dst*	s4 xu4 dws4	.M1, .M2	00101

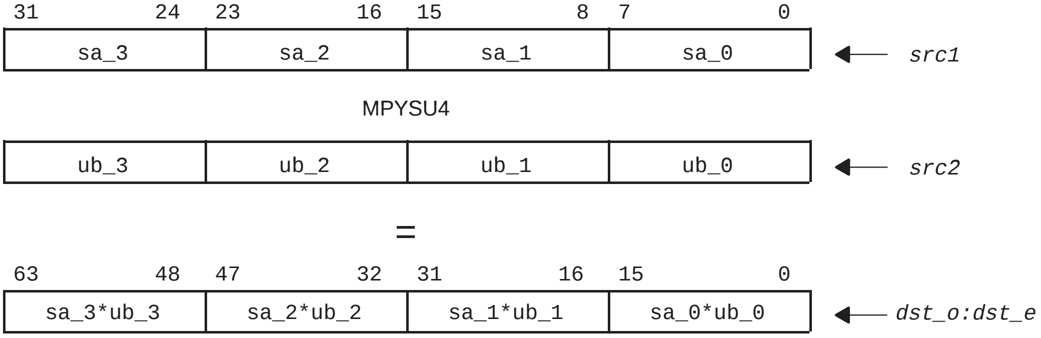
Opcode

31 29	28	27 23	22 18	17 13	12	11	10 6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	0	*op*	1	1	0	0	*s*	*p*
3	1	5	5	5	1	1	5	4				1	1

Description

The **MPYSU4** instruction returns the product between four sets of packed 8-bit values producing four signed 16-bit results. The four signed 16-bit results are packed into a 64-bit register pair, *dst_o:dst_e*. The values in *src1* are treated as signed packed 8-bit quantities, whereas the values in *src2* are treated as unsigned 8-bit packed data.

For each pair of 8-bit quantities in *src1* and *src2*, the signed 8-bit value from *src1* is multiplied with the unsigned 8-bit value from *src2*. The product of *src1* byte 0 and *src2* byte 0 is written to the lower half of *dst_e*. The product of *src1* byte 1 and *src2* byte 1 is written to the upper half of *dst_e*. The product of *src1* byte 2 and *src2* byte 2 is written to the lower half of *dst_o*. The product of *src1* byte 3 and *src2* byte 3 is written to the upper half of *dst_o*.



Execution

```
if (cond)    {
                  (sbyte0(src1) x ubyte0(src2)) → lsb16(dst_e)
                  (sbyte1(src1) x ubyte1(src2)) → msb16(dst_e)
                  (sbyte2(src1) x ubyte2(src2)) → lsb16(dst_o)
                  (sbyte3(src1) x ubyte3(src2)) → msb16(dst_o)
                  }
else nop
```

Pipeline

Pipeline

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

Instruction Type Four-cycle

Delay Slots 3

See Also **MPYU4**

Example 1

```
MPYSU4  .M1  A5,A6,A9:A8
```

	Before instruction			**4 cycles after instruction**	
A5	6A 32 11 93h	106 50 17 –109 signed	A5	6A 32 11 93h	106 50 17 –109 signed
A6	B1 74 6C A4h	177 116 108 164 unsigned	A6 .	B1 74 6C A4h	177 116 108 164 unsigned
A9:A8	XXXX XXXXh	XXXX XXXXh	A9:A8	49A4 16A8h 18762 5800	072C BA2Ch 1386 –17876 signed

Example 2

```
MPYSU4  .M2   B2,B5,B9:B8
```

	Before instruction			4 cycles after instruction	
A5	3F F6 50 10h	63 -10 80 16 signed	A5	3F F6 50 10h	63 -10 80 16 signed
A6	C3 56 02 44h	195 86 2 68 unsigned	A6 .	C3 56 02 44h	195 86 2 68 unsigned
A9:A8	XXXX XXXXh	XXXX XXXXh	A9:A8	2FFD FCA4h 12285 –680	00A0 0440h 160 1088 signed

MPYUS4 *Multiply Unsigned by Signed Packed, 8-Bit (Pseudo-Operation)*

Syntax

MPYUS4 (.unit) *src2, src1*, *dst*
.unit = .M1, .M2

Opcode map field used...	For operand type	Unit	Opfield
src1 *src2* *dst*	s4 xu4 dws4	.M1, .M2	00101

Opcode

See **MPYSU4** instruction.

Description

The **MPYUS4** pseudo-operation returns the product between four sets of packed 8-bit values, producing four signed 16-bit results. The four signed 16-bit results are packed into a 64-bit register pair, *dst_o:dst_e*. The values in *src1* are treated as signed packed 8-bit quantities, whereas the values in *src2* are treated as unsigned 8-bit packed data. The assembler uses the **MPYSU4** *src1*, *src2*, *dst* instruction to perform this operation.

For each pair of 8-bit quantities in *src1* and *src2*, the signed 8-bit value from *src1* is multiplied with the unsigned 8-bit value from *src2*. The product of *src1* byte 0 and *src2* byte 0 is written to the lower half of *dst_e*. The product of *src1* byte 1 and *src2* byte 1 is written to the upper half of *dst_e*. The product of *src1* byte 2 and *src2* byte 2 is written to the lower half of *dst_o*. The product of *src1* byte 3 and *src2* byte 3 is written to the upper half of *dst_o*.

Execution

```
if (cond)   {
            (ubyte0(src2) x sbyte0(src1)) → lsb16(dst_e)
            (ubyte1(src2) x sbyte1(src1)) → msb16(dst_e)
            (ubyte2(src2) x sbyte2(src1)) → lsb16(dst_o)
            (ubyte3(src2) x sbyte3(src1)) → msb16(dst_o)
            }
else nop
```

Pipeline

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

Instruction Type Four-cycle

Delay Slots 3

See Also **MPYSU4, MPYU4**

MPYU4 — *Multiply Unsigned by Unsigned Packed, 8-Bit*

Syntax

MPYU4 (.unit) *src1*,*src2, dst*
.unit = .M1, .M2

Opcode map field used...	For operand type	Unit	Opfield
src1 *src2* *dst*	s4 xu4 dwu4	.M1, .M2	00100

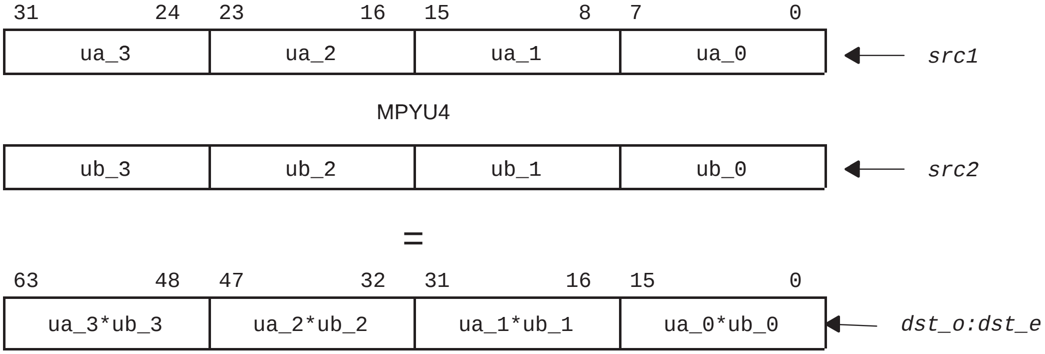
Opcode

31 29	28	27 23	22 18	17 13	12	11	10 6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	0	*op*	1	1	0	0	*s*	*p*
3	1	5	5	5	1	1	5	4				1	1

Description

The **MPYU4** instruction returns the product between four sets of packed 8-bit values producing four unsigned 16-bit results that are packed into a 64-bit register pair, *dst_o:dst_e*. The values in both *src1* and *src2* are treated as unsigned 8-bit packed data.

For each pair of 8-bit quantities in *src1* and *src2*, the unsigned 8-bit value from *src1* is multiplied with the unsigned 8-bit value from *src2*. The product of *src1* byte 0 and *src2* byte 0 is written to the lower half of *dst_e*. The product of *src1* byte 1 and *src2* byte 1 is written to the upper half of *dst_e*. The product of *src1* byte 2 and *src2* byte 2 is written to the lower half of *dst_o*. The product of *src1* byte 3 and *src2* byte 3 is written to the upper half of *dst_o*.



Execution

```
if (cond)    {
             (ubyte0(src1) x ubyte0(src2)) → lsb16(dst_e)
             (ubyte1(src1) x ubyte1(src2)) → msb16(dst_e)
             (ubyte2(src1) x ubyte2(src2)) → lsb16(dst_o)
             (ubyte3(src1) x ubyte3(src2)) → msb16(dst_o)
             }
else nop
```

Pipeline

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

Instruction Type Four-cycle

Delay Slots 3

See Also **MPYSU4**

Example 1 `MPYU4  .M1   A5,A6,A9:A8`

```
          Before instruction                       4 cycles after instruction

A5     | 68 32 C1 93h | 104 50 193 147   A5     | 68 32 C1 93h |  104 50 193 147
                        unsigned                                  unsigned
A6     | B1 74 2C ABh | 177 116 44 171   A6     | B1 74 2C ABh |  177 116 44 171
                        unsigned         .                        unsigned
A9:A8  | XXXX XXXXh   | XXXX XXXXh       A9:A8  | 47E8 16A8h   | | 212C 6231h    |
                                                  18408  5800     8492 25137
                                                                  unsigned
```

Example 2 `MPYU4  .M2   B2,B5,B9:B8`

```
            Before instruction                          4 cycles after instruction

B2      3D E6 50 7Fh   61 230 80 127    B2      3D E6 50 7Fh    61 230 80 127
                       unsigned                                 unsigned

B5      C3 56 02 44h   195 86 2 68      B5      C3 56 02 44h    195 86 2 68
                       unsigned         .                       unsigned

B9:B8   XXXX XXXXh     XXXX XXXXh       B9:B8   2E77 4D44h      00A0 21BCh
                                                11895  19780    160 8636
                                                                unsigned
```

MVD — *Move From Register to Register, Delayed*

Syntax

MVD (.unit) *src2, dst*
.unit = .M1, .M2

Opcode map field used...	For operand type	Unit	Opfield
src2 *dst*	xint int	.M1, .M2	11010

Opcode

31 30 29	28	27 23	22 18	17 13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src*	*op*	x	0	0	0	0	1	1	1	1	0	0	*s*	*p*
3	1	5	5	5	1	10										1	1

Description

The **MVD** instruction moves data from the *src2* register to the *dst* register over 4 cycles. This is done using the multiplier path.

```
MVD    .M2x   A0,B0     ;
NOP                     ;
NOP                     ;
NOP                     ; B0 = A0
```

Execution

if (cond) *src2* → *dst*

else nop

Pipeline

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

Instruction Type Four-cycle

Delay Slots 3

Example `MVD  .M2X   A5,B8`

	Before instruction		4 cycles after instruction
A5	6A32 1193h	A5	6A32 1193h
B8	XXXX XXXXh	B8	6A32 1193h

MVK/MVKL *Move a Signed Constant Into a Register and Sign-Extend*

Syntax

MVK (.unit) cst, *dst*
.unit = .L1 or .L2, .S1 or .S2, .D1 or .D2

Opcode map field used...	For operand type	Unit	Opfield
cst *dst*	scst16 sint	.S1, .S2	
cst *dst*	scst5 sint	.L1, .L2	00101
cst *dst*	scst5 sint	.D1, .D2	000000

Opcode

.S Unit

31 29	28	27 23	22 7	6	5	4	3	2	1	0
creg	*z*	*dst*	*scst16*	0	1	0	1	0	*s*	*p*
3	1	5	16	5					1	1

.L Unit

31 29	28	27 23	22 18	17 13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*scst*	*op*	x	0	0	1	1	0	1	0	1	1	0	*s*	*p*
3	1	5	5	5	1	10										1	1

.D Unit

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

Description

In the **MVK/MVKL** instruction, the *cst* (constant) is sign extended and placed in *dst*. The .S unit form allows for a 16-bit signed constant. This is the same **MVK** instruction that is found on the C62x but with the added flexibility of being able to perform this operation not only on the .S unit but also on the .L and .D units when the constant is limited to a 5-bit signed constant.

Since many non-address constants fall into a 5-bit sign constant range, this allows the flexibility to schedule the **MVK** instruction on the .L or .D units. In the .D unit form, the constant is in the position normally used by *src1*, as for address math. Only the **MVK** instruction is supported on the .L or .D units. The .S unit supports both **MVK** and **MVKL**.

In most cases, the C6000 assembler and linker issue a warning or an error when a constant is outside the range supported by the instruction. In the case of **MVK .S,** a warning is issued whenever the constant is outside the signed 16-bit range, –32768 to 32767 (or 0XFFFF8000 to 0x 00007FFF).

For example:

```
MVK  .S1   0x00008000X, A0
```

will generate a warning, whereas

```
MVK  .S1  0 x FFFF8000, A0
```

will not generate a warning.

The **MVKL** instruction is equivalent to the **MVK** instruction, except that the **MVKL** disables the constant range checking normally performed by the assembler/linker. This allows **MVKL** to be paired with **MVKH** to generate 32-bit constants.

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

```
MVKL  .S1 0x0ABCD, A4
MVKLH .S1 0x1234, A4
```

This could also be used:

```
MVKL  .S1 0x1234ABCD, A4
MVKH  .S1 0x1234ABCD, A4
```

Use this to load the address of a label:

```
MVKL .S2 label, B5
MVKH .S2 label, B5
```

Execution

if (cond) scst → *dst*

else nop

Pipeline

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

Instruction Type Single Cycle

Delay Slots 0

See Also **MVKH, MVKLH**

Example 1

```
MVK   .L2   –5,B8
```

	Before instruction		1 cycle after instruction
B8	XXXX XXXXh	B8	FFFF FFFBh

Example 2

```
MVK   .D2   14,B8
```

	Before instruction		1 cycle after instruction
B8	XXXX XXXXh	B8	0000 000Eh

Example 3

```
MVKL   .S1   5678h,A8
```

	Before instruction		1 cycle after instruction
A8	XXXX XXXXh	A8	0000 5678h

Example 4

```
MVKL   .S1   0C678h,A8
```

	Before instruction		1 cycle after instruction
A8	XXXX XXXXh	A8	FFFF C678h

OR

Bitwise OR

Syntax

OR (.unit) *src1, src2, dst*
.unit =.D1 or .D2, .L1, .L2, .S1, .S2,

Opcode map field used...	**For operand type...**	**Unit**	**Opfield**
src *src2* *dst*	uint xuint uint	.D1, .D2	0001
src1 *src2* *dst*	scst5 xuint uint	.D1, .D2	0011
src1 *src2* *dst*	uint xint uint	.L1, .L2	1111111
src1 *src2* *dst*	scst5 xuint uint	.L1, .L2	1111110
src1 *src2* *dst*	uint xunit uint	.S1, .S2	011011
src1 *src2* *dst*	scst5 xuint uint	.S1, .S2	011010

Opcode

.D unit

31 29	28	27 23	22 18	17 13	12	11	10	9 6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1/cst5*	x	1	0	*op*	1	1	0	0	*s*	*p*
3	1	5	5	5	1	2		4	4				1	1

.L unit

31 29	28	27 23	22 18	17 13	12	11 5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1/cst5*	x	*op*	1	1	0	*s*	*p*
3	1	5	5	5	1	7	3			1	1

.S Unit

31 29	28	27 23	22 18	17 13	12	11 6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1/cst5*	x	*op*	1	0	0	0	*s*	*p*
3	1	5	5	5	1	6	4				1	1

Description A bitwise **OR** instruction is performed between *src1* and *src2*. The result is placed in *dst.* The *scst5* operands are sign extended to 32 bits. This is the same **OR** instruction that is found on the C62x, but with the added flexibility of being able to perform this operation on the .D unit as well as the .L and .S units.

Execution if (cond) *src1* or *src2* → *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

See Also **AND, ANDN, XOR**

Example 1 OR .S1 A3,A4,A5

	Before instruction		**1 cycle after instruction**
A3	08A3 A49Fh	A3	08A3 A49Fh
A4	00FF 375Ah	A4	00FF 375Ah
A5	XXXX XXXX h	A5	08FF B7DFh

Example 2

```
OR  .D2  –12,B2,B8
```

	Before instruction	1 cycle after instruction
B2	0000 3A41h	0000 3A41h
B8	XXXX XXXXh	FFFF FFF5h

PACK2 *Pack 16 LSB, 16 LSB Into Packed 16-Bit*

Syntax

PACK2 (.unit) *src1*,*src2, dst*
.unit = .L1, .L2 or .S1, .S2

Opcode map field used...	For operand type	Unit	Opfield
src1 *src2* *dst*	i2 xi2 i2	.L1, .L2	0000000
src1 *src2* *dst*	i2 xi2 i2	.S1 .S2	1111

Opcode

.L unit

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

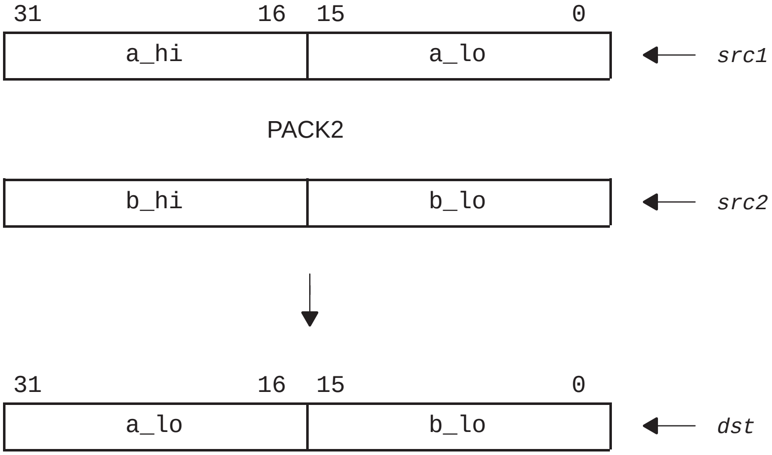
.S Unit

31 30 29	28	27 23	22 18	17 13	12	11	10	9 6	5			2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	1	1	*op*	1	1	0	0	*s*	*p*
3	1	5	5	5	1	2		4		4			1	1

Description

The **PACK2** instruction takes the lower half-words from *src1* and *src2* and packs them both into *dst*. The lower half-word of *src1* is placed in the upper half-word of *dst*. The lower half-word of *src2* is placed in the lower half-word of *dst*.

This instruction is useful for manipulating and preparing pairs of 16-bit values to be used by the packed arithmetic operations, such as **ADD2.**



Execution

```
if (cond)    {
             lsb16(src2) → lsb16(dst)
             lsb16(src1) → msb16(dst);
             }
else nop
```

Pipeline

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

Instruction Type Single-cycle

Delay Slots 0

See Also **PACKH2, PACKHL2, PACKLH2, SPACK2**

Example 1

```
PACK2   .L1   A2,A8,A9
```

	Before instruction		1 cycle after instruction
A2	3789 F23Ah	A2	3789 F23Ah
A8	04B8 4975h	A8	04B8 4975h
A9	XXXX XXXX h	A9	F23A 4975h

Example 2

```
PACK2   .S2   B2,B8,B12
```

	Before instruction		1 cycle after instruction
B2	0124 2451h	B2	0124 2451h
B8	01A6 A051h	B8	01A6 A051h
B12	XXXX XXXXh	B12	2451 A051h

PACKH2

Pack 16 MSB, 16 MSB Into Packed 16-Bit

Syntax

PACKH2 (.unit) *src1*,*src2, dst*
.unit = .L1, .L2 or .S1, .S2

Opcode map field used...	For operand type	Unit	Opfield
src1 *src2* *dst*	i2 xi2 i2	.L1, .L2	0011110
src1 *src2* *dst*	i2 xi2 i2	.S1 .S2	001001

Opcode

.L unit

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

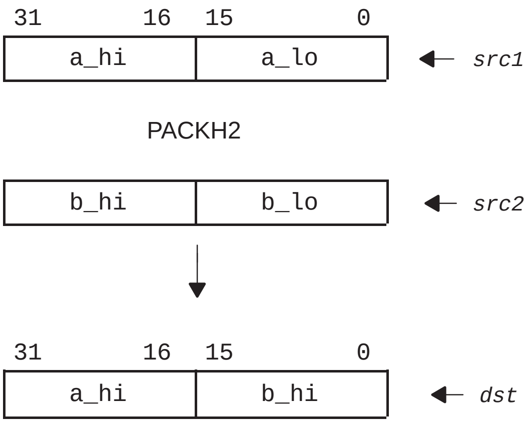
.S unit

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

Description

The **PACKH2** instruction takes the upper half-words from *src1* and *src2* and packs them both into *dst*. The upper half-word of *src1* is placed in the upper half-word of *dst*. The upper half-word of *src2* is placed in the lower half-word of *dst*.

This instruction is useful for manipulating and preparing pairs of 16-bit values to be used by the packed arithmetic operations, such as **ADD2.**



Execution

```
if (cond)    {
                 msb16(src2) → lsb16(dst)
                 msb16(src1) → msb16(dst);
                 }
else nop
```

Pipeline

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

Instruction Type Single-cycle

Delay Slots 0

See Also **PACK2, PACKHL2, PACKLH2, SPACK2**

Example 1

```
PACKH2  .L1   A2,A8,A9
```

	Before instruction		1 cycle after instruction
A2	3789 F23Ah	A2	3789 F23Ah
A8	04B8 4975h	A8	04B8 4975h
A9	XXXX XXXX h	A9	3789 04B8h

Example 2

```
PACKH2  .S2   B2,B8,B12
```

	Before instruction		1 cycle after instruction
B2	0124 2451h	B2	0124 2451h
B8	01A6 A051h	B8	01A6 A051h
B12	XXXX XXXXh	B12	0124 01A6h

PACKH4 *Pack High Bytes of Four Half-Words Into Packed 8-Bit*

Syntax

PACKH4 (.unit) *src1*,*src2, dst*
.unit = .L1, .L2

Opcode map field used...	For operand type	Unit	Opfield
src1 *src2* *dst*	i4 xi4 i4	.L1, .L2	1101001

Opcode

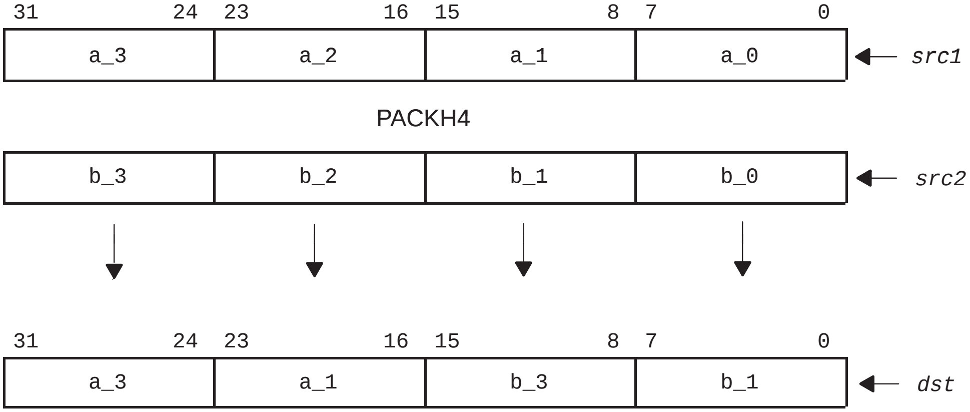
.L unit

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

Description

The **PACKH4** instruction moves the high bytes of the two half-words in *src1* and *src2* and packs them into *dst*. The bytes from *src1* will be packed into the most significant bytes of *dst*, and the bytes from *src2* will be packed into the least significant bytes of *dst*.

Specifically, the high byte of the upper half-word of *src1* is moved to the upper byte of the upper half-word of *dst*. The high byte of the lower half-word of *src1* is moved to the lower byte of the upper half-word of *dst*. The high byte of the upper half-word of *src2* is moved to the upper byte of the lower half-word of *dst*. The high byte of the lower half-word of *src2* is moved to the lower byte of the lower half-word of *dst*.



Execution

```
if (cond)    {
                 byte3(src1) → byte3(dst);
                 byte1(src1) → byte2(dst);
                 byte3(src2) → byte1(dst);
                 byte1(src2) → byte0(dst);
             }
else nop
```

Pipeline

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

Instruction Type Single-cycle

Delay Slots 0

See Also **PACKL4, SPACKU4**

Example 1

```
PACKH4  .L1   A2,A8,A9
```

	Before instruction		**1 cycle after instruction**
A2	37 89 F2 3Ah	A2	37 89 F2 3Ah
A8	04 B8 49 75h	A8	04 B8 49 75h
A9	XXXX XXXXh	A9	37 F2 04 49h

Example 2

```
PACKH4  .L2   B2,B8,B12
```

	Before instruction		1 cycle after instruction
B2	01 24 24 51h	B2	01 24 24 51h
B8	01 A6 A0 51h	B8	01 A6 A0 51h
B12	XXXX XXXXh	B12	01 24 01 A0h

PACKHL2 — *Pack 16 MSB, 16 LSB Into Packed 16-Bit*

Syntax

PACKHL2 (.unit) *src1*,*src2, dst*
.unit = .L1, .L2 or .S1, .S2

Opcode map field used...	For operand type...	Unit	Opfield
src1 *src2* *dst*	i2 xi2 i2	.L1, .L2	0011100
src1 *src2* *dst*	i2 xi2 i2	.S1 . S2	001000

Opcode

.L unit

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

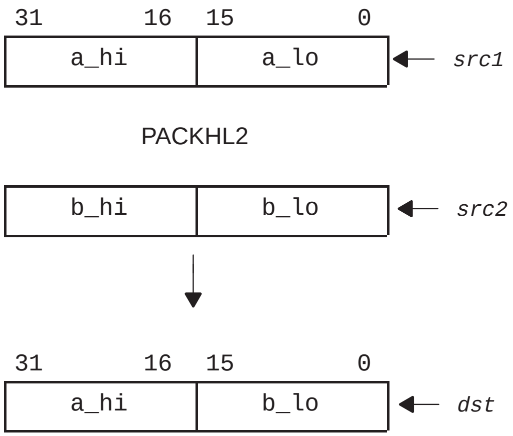
.S unit

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

Description

The **PACKHL2** instruction takes the upper half-word from *src1* and the lower half-word from *src2* and packs them both into *dst*. The upper half-word of *src1* is placed in the upper half-word of *dst*. The lower half-word of *src2* is placed in the lower half-word of *dst*.

This instruction is useful for manipulating and preparing pairs of 16-bit values to be used by the packed arithmetic operations, such as **ADD2**.



Execution

```
if (cond)    {
                 lsb16(src2) $\rightarrow$ lsb16(dst)
                 msb16(src1) $\rightarrow$ msb16(dst);
              }
else nop
```

Pipeline

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

Instruction Type Single-cycle

Delay Slots 0

See Also **PACK2**, **PACKH2**, **PACKLH2**, **SPACK2**

Example 1

```
PACKHL2  .L1   A2,A8,A9
```

	Before instruction		1 cycle after instruction
A2	3789 F23Ah	A2	3789 F23Ah
A8	04B8 4975h	A8	04B8 4975h
A9	XXXX XXXXh	A9	3789 4975h

Example 2

```
PACKHL2  .S2   B2,B8,B12
```

	Before instruction		1 cycle after instruction
B2	0124 2451h	B2	0124 2451h
B8	01A6 A051h	B8	01A6 A051h
B12	XXXX XXXXh	B12	0124 A051h

PACKL4

Pack Low Bytes of Four Half-Words Into Packed 8-Bit

Syntax

PACKL4 (.unit) *src1*,*src2, dst*
.unit = .L1, .L2

Opcode map field used...	For operand type...	Unit	Opfield
src1 *src2* *dst*	i4 xi4 i4	.L1, .L2	1101000

Opcode

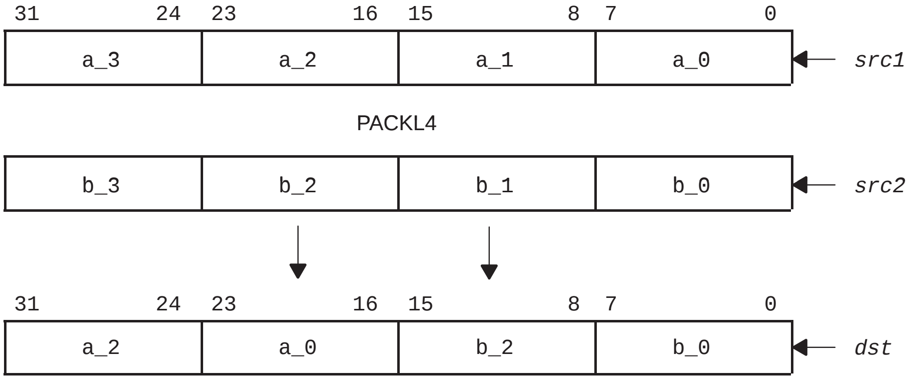
.L unit

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

Description

The **PACKL4** instruction moves the low bytes of the two half-words in *src1* and *src2,* and packs them into *dst*. The bytes from *src1* will be packed into the most significant bytes of *dst*, and the bytes from *src2* will be packed into the least significant bytes of *dst*.

Specifically, the low byte of the upper half-word of *src1* is moved to the upper byte of the upper half-word of *dst*. The low byte of the lower half-word of *src1* is moved to the lower byte of the upper half-word of *dst*. The low byte of the upper half-word of *src2* is moved to the upper byte of the lower half-word of *dst*. The low byte of the lower half-word of *src2* is moved to the lower byte of the lower half-word of *dst*.



Execution

```
if (cond)    {
                 byte2(src1) → byte3(dst)
                 byte0(src1) → byte2(dst)
                 byte2(src2) → byte1(dst)
                 byte0(src2) → byte0(dst);
                 }
else  nop
```

Pipeline

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

Instruction Type Single-cycle

Delay Slots 0

See Also **PACKH4, SPACKU4**

Example 1

```
PACKL4  .L1   A2,A8,A9
```

	Before instruction		1 cycle after instruction
A2	37 89 F2 3Ah	A2	37 89 F2 3Ah
A8	04 B8 49 75h	A8	04 B8 49 75h
A9	XXXX XXXXh	A9	89 3A B8 75h

Example 2

PACKL4 .L2 B2,B8,B12

	Before instruction		1 cycle after instruction
B2	01 24 24 51h	B2	01 24 24 51h
B8	01 A6 A0 51h	B8	01 A6 A0 51h
B12	XXXX XXXXh	B12	24 51 A6 51h

PACKLH2 *Pack 16LSB, 16MSB Into Packed 16-Bit*

Syntax **PACKLH2** (.unit) *src1*,*src2, dst*
.unit = .L1, .L2 or .S1, .S2

Opcode map field used...	For operand type...	Unit	Opfield
src1 *src2* *dst*	i2 xi2 i2	.L1, .L2	0011011
src1 *src2* *dst*	i2 xi2 i2	.S1 .S2	010000

Opcode

.L unit

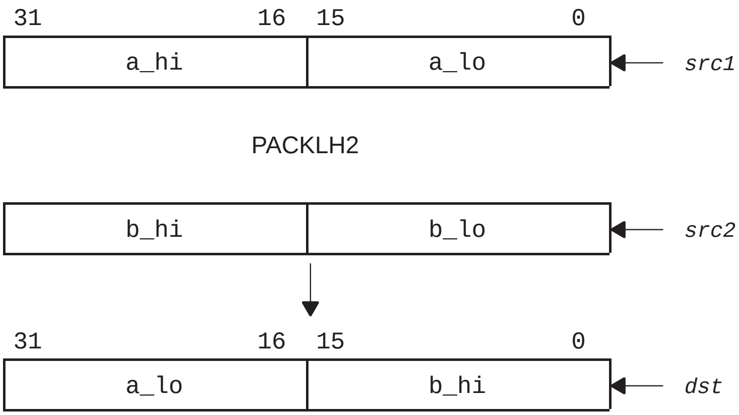
31 29	28	27 23	22 18	17 13	12	11 5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	*op*	1	1	0	*s*	*p*
3	1	5	5	5	1	7		3		1	1

.S unit

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

Description The **PACKLH2** instruction takes the lower half-word from *src1,* and the upper half-word from *src2,* and packs them both into *dst*. The lower half-word of *src1* is placed in the upper half-word of *dst*. The upper half-word of *src2* is placed in the lower half-word of *dst*.

This instruction is useful for manipulating and preparing pairs of 16-bit values to be used by the packed arithmetic operations, such as **ADD2**.



Execution

```
if (cond)    {
             msb16(src2) → lsb16(dst)
             lsb16(src1) → msb16(dst);
             }
else  nop
```

Pipeline

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

Instruction Type Single-cycle

Delay Slots 0

See Also **PACK2, PACKH2, PACKHL2, SPACK2**

Example 1 PACKLH2 .L1 A2,A8,A9

	Before instruction		1 cycle after instruction
A2	3789 F23Ah	A2	3789 F23Ah
A8	04B8 4975h	A8	04B8 4975h
A9	XXXX XXXXh	A9	F23A 04B8h

Example 2 PACKLH2 .S2 B2,B8,B12

	Before instruction		1 cycle after instruction
B2	0124 2451h	B2	0124 2451h
B8	01A6 A051h	B8	01A6 A051h
B12	XXXX XXXX h	B12	2451 01A6h

ROTL

Rotate Left

Syntax

ROTL (.unit) *src2*,*src1, dst*
.unit = .M1, .M2

Opcode map field used...	For operand type...	Unit	Opfield
src1 *src2* *dst*	uint xuint uint	.M1, .M2	11101
src1 *src2* *dst*	ucst5 xuint uint	.M1, .M2	11110

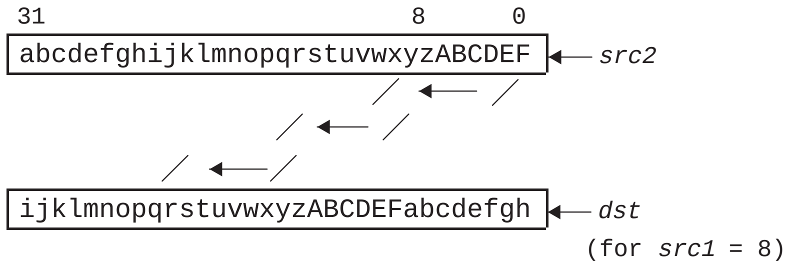
Opcode

31 29	28	27 23	22 18	17 13	12	11	10 6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1/ucst*	x	0	*op*	1	1	0	0	*s*	*p*
3	1	5	5	5	1	1	5	4				1	1

Description

The **ROTL** instruction rotates the 32-bit value of *src2* to the left, and places the result in *dst*. The number of bits to rotate is given in the five least-significant bits of *src1*. Bits 5 through 31 of *src1* are ignored and may be non-zero.

In the example below, *src1* is equal to 8.


31
8
0
abcdefghijklmnopqrstuvwxyzABCDEF
src2
ijklmnopqrstuvwxyzABCDEFabcdefgh
dst
(for src1 = 8)

Note:

The **ROTL** instruction is useful in cryptographic applications.

Execution

```
if (cond)    {
               (src2 << src1) | (src2 >> (32–src1)) → dst
               }
else  nop
```

Pipeline

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

Instruction Type Two-cycle

Delay Slots 1

See Also **SHL, SHLMB, SHRMB, SHR, SHRU**

Example 1

```
ROTL  .M2   B2,B4,B5
```

	Before instruction		2 cycles after instruction
B2	`A6E2 C179h`	B2	`A6E2 C179h`
B4	`1458 3B69h`	B4	`1458 3B69h`
B5	`XXXX XXXXh`	B5	`C582 F34Dh`

Example 2

```
ROTL  .M1   A4,10h,A5
```

	Before instruction		2 cycles after instruction
A4	`187A 65FCh`	A4	`187A 65FCh`
A5	`XXXX XXXXh`	A5	`65FC 187Ah`

SADD2 *Add With Saturation, Signed Packed 16-Bit*

Syntax

SADD2 (.unit) *src1*,*src2, dst*
.unit = .S1, .S2

Opcode map field used...	For operand type...	Unit	Opfield
src1 *src2* *dst*	s2 xs2 s2	.S1, .S2	0000

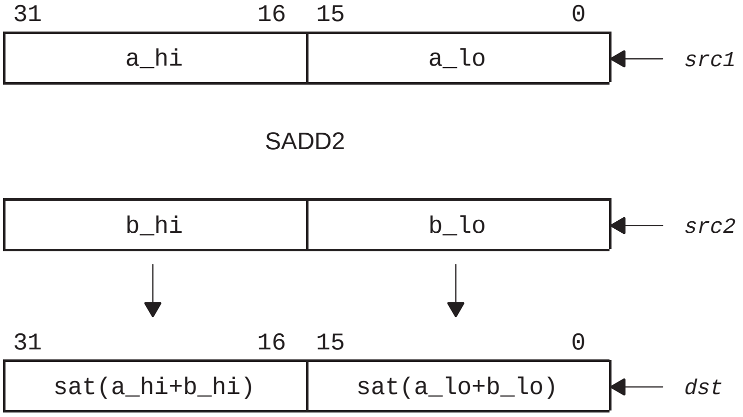
Opcode

31 29	28	27 23	22 18	17 13	12	11	10	9 6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	1	1	*op*	1	1	0	0	*s*	*p*
3	1	5	5	5	1	2		4	4				1	1

Description

The **SADD2** instruction performs 2s-complement addition between signed, packed 16-bit quantities in *src1* and *src2*. The results are placed in a signed, packed 16-bit format into *dst*.

For each pair of 16-bit quantities in *src1* and *src2*, the sum between the signed 16-bit value from *src1* and the signed 16-bit value from *src2* is calculated and saturated to produce a signed 16-bit result. The result is placed in the corresponding position in *dst*.



Note:

This operation is performed on each half-word separately. This instruction does not affect the SAT bit in the CSR.

Execution

```
if (cond)    {
                 sat((msb16(src1) + msb16(src2))) → msb16(dst)
                 sat((lsb16(src1) + lsb16(src2))) → lsb16(dst)
                 }
else  nop
```

Saturation (shown above as sat) is performed on each 16-bit result independently. For each sum, the following tests are applied:

❑ If the sum is in the range -2^{15} to $2^{15}-1$, inclusive, then no saturation is performed and the sum is left unchanged.

❑ If the sum is greater than $2^{15}-1$, then the result is set to $2^{15}-1$.

❑ If the sum is less than -2^{15}, then the result is set to -2^{15}.

Pipeline

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

Instruction Type Single-cycle

Delay Slots 0

See Also **SADD, SADDU4, SADDUS2**

Example 1 SADD2 .S1 A2,A8,A9

	Before instruction			**1 cycle after instruction**	
A2	5789 F23Ah	22409 –3526	A2	5789 F23Ah	22409 –3526
A8	74B8 4975h	29880 18805	A8	74B8 4975h	29880 18805
A9	XXXX XXXXh		A9	7FFF 3BAFh	32767 15279

Example 2 SADD2 .S2 B2,B8,B12

	Before instruction			**1 cycle after instruction**	
B2	0124 847Ch	292 –31260	B2	0124 847Ch	292 –31260
B8	01A6 A051h	422 –24495	B8	01A6 A051h	422 –24495
B12	XXXX XXXXh		B12	02AC 8000h	684 –32768

SADDU4 *Add With Saturation, Unsigned, Packed 8-Bit*

Syntax

SADDU4 (.unit) *src1*,*src2, dst*
.unit = .S1, .S2

Opcode map field used...	For operand type...	Unit	Opfield
src1 *src2* *dst*	u4 xu4 u4	.S1 ,.S2	0011

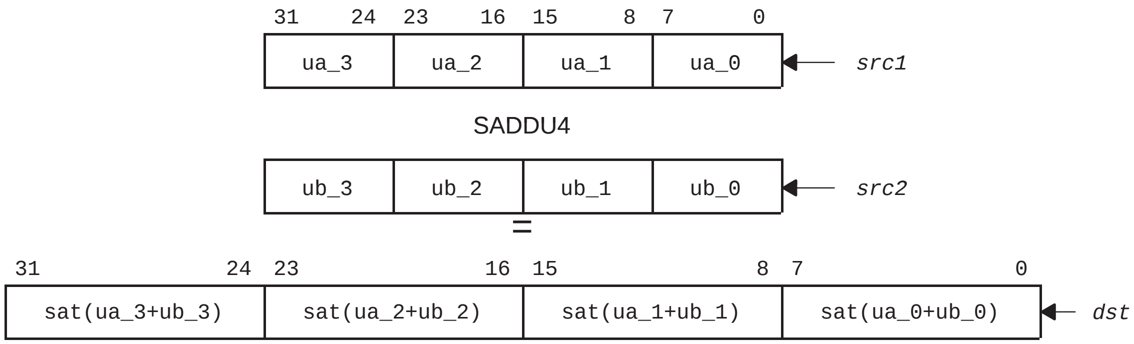
Opcode

31 29	28	27 23	22 18	17 13	12	11	10	9 6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	1	1	*op*	1	1	0	0	*s*	*p*
3	1	5	5	5	1	2		4	4				1	1

Description

The **SADDU4** instruction performs 2s-complement addition between unsigned, packed 8-bit quantities. The values in *src1* and *src2* are treated as unsigned, packed 8-bit quantities. The results are written into *dst* in an unsigned, packed 8-bit format.

For each pair of 8-bit quantities in *src1* and *src2*, the sum between the unsigned 8-bit value from *src1* and the unsigned 8-bit value from *src2* is calculated and saturated to produce an unsigned 8-bit result. The result is placed in the corresponding position in *dst*.



Note:

This operation is performed on each 8-bit quantity separately. This instruction does not affect the SAT bit in the CSR.

Execution

```
if (cond)    {
             sat((ubyte0(src1) + ubyte0(src2))) → ubyte0(dst)
             sat((ubyte1(src1) + ubyte1(src2))) → ubyte1(dst)
             sat((ubyte2(src1) + ubyte2(src2))) → ubyte2(dst)
             sat((ubyte3(src1) + ubyte3(src2))) → ubyte3(dst)
             }
else  nop
```

Saturation (shown above as sat) is performed on each 8-bit result independently. For each sum, the following tests are applied:

- ❑ If the sum is in the range 0 to $2^{8} - 1$, inclusive, then no saturation is performed and the sum is left unchanged.
- ❑ If the sum is greater than $2^{8} - 1$, then the result is set to $2^{8} - 1$.

Pipeline

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

Instruction Type Single-cycle

Delay Slots 0

See Also **SADD, SADD2, SADDUS2**

Example 1 SADDU4 .S1 A2, A8, A9

	Before instruction			1 cycle after instruction	
A2	57 89 F2 3Ah	87 137 242 58 unsigned	A2	57 89 F2 3Ah	87 137 242 58 unsigned
A8	74 B8 49 75h	116 184 73 117 unsigned	A8	74 B8 49 75h	116 184 73 117 unsigned
A9	XXXX XXXXh		A9	CB FF FF AFh	203 255 255 175 unsigned

Example 2 SADDU4 .S2 B2, B8, B12

	Before instruction			1 cycle after instruction	
B2	14 7C 01 24h	20 124 1 36 unsigned	B2	14 7C 01 24h	20 124 1 36 unsigned
B8	A0 51 01 A6h	160 81 1 166 unsigned	B8	A0 51 01 A6h	160 81 1 166 unsigned
B12	XXXX XXXXh		B12	B4 CD 02 CA	180 205 2 202 unsigned

SADDSU2

Add With Saturation, Signed With Unsigned Packed 16-Bit (Pseudo-Operation)

Syntax

SADDSU2(.unit) *src2, src1*, *dst*
.unit = .S1, .S2

Opcode map field used...	For operand type...	Unit	Opfield
src1 *src2* *dst*	u2 xs2 u2	.S1, .S2	0001

Opcode

See **SADDUS2** instruction.

Description

The **SADDSU2** pseudo-operation performs 2s-complement addition between unsigned and signed packed 16-bit quantities. The values in *src1* are treated as unsigned packed 16-bit quantities, and the values in *src2* are treated as signed packed 16-bit quantities. The results are placed in an unsigned packed 16-bit format into *dst*. The assembler uses the **SADDUS2** *src1*, *src2*, *dst* instruction to perform this operation.

For each pair of 16-bit quantities in *src1* and *src2*, the sum between the unsigned 16-bit value from *src1* and the signed 16-bit value from *src2* is calculated and saturated to produce a signed 16-bit result. The result is placed in the corresponding position in *dst*.

Saturation is performed on each 16-bit result independently. For each sum, the following tests are applied:

- ❑ If the sum is in the range 0 to $2^{16} – 1$, inclusive, then no saturation is performed and the sum is left unchanged.
- ❑ If the sum is greater than $2^{16} – 1$, then the result is set to $2^{16} – 1$.
- ❑ If the sum is less than 0, then the result is set to 0.

Execution

```
if (cond)     {
              SAT((smsb16(src2) + umsb16(src1))) → umsb16(dst)
              SAT((slsb16(src2) + ulsb16(src1))) → ulsb16(dst)
              }
else  nop
```

Pipeline

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

Instruction Type Single-cycle

Delay Slots 0

See Also **SADD, SADD2, SADDUS2, SADDU4**

SADDUS2 — *Add With Saturation, Unsigned With Signed Packed 16-Bit*

Syntax

SADDUS2(.unit) *src1*,*src2, dst*
.unit = .S1, .S2

Opcode map field used...	For operand type...	Unit	Opfield
src1	u2	.S1 ,.S2	0001
src2	xs2		
dst	u2		

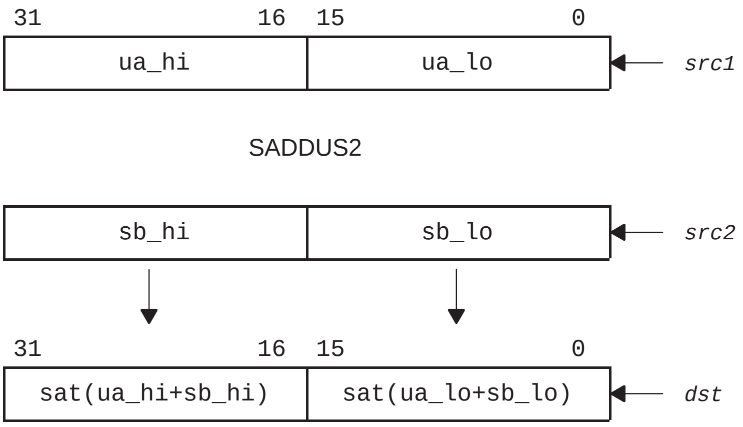
Opcode

31 29	28	27 23	22 18	17 13	12	11	10	9 6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	1	1	*op*	1	1	0	0	*s*	*p*
3	1	5	5	5	1	2		4	4				1	1

Description

The **SADDUS2** instruction performs 2s-complement addition between unsigned, and signed, packed 16-bit quantities. The values in *src1* are treated as unsigned, packed 16-bit quantities; and the values in *src2* are treated as signed, packed 16-bit quantities. The results are placed in an unsigned, packed 16-bit format into *dst*.

For each pair of 16-bit quantities in *src1* and *src2*, the sum between the unsigned 16-bit value from *src1* and the signed 16-bit value from *src2* is calculated and saturated to produce a signed 16-bit result. The result is placed in the corresponding position in *dst*.



Note:

This operation is performed on each half-word separately. This instruction does not affect the SAT bit in the CSR.

Execution

```
if (cond)     {
                  sat((umsb16(src1) + smsb16(src2))) → umsb16(dst)
                  sat((ulsb16(src1) + slsb16(src2))) → ulsb16(dst)
                  }
else  nop
```

Saturation (shown above as sat) is performed on each 16-bit result independently. For each sum, the following tests are applied:

- If the sum is in the range 0 to $2^{16} - 1$, inclusive, then no saturation is performed and the sum is left unchanged.
- If the sum is greater than $2^{16} - 1$, then the result is set to $2^{16} - 1$.
- If the sum is less than 0, then the result is set to 0.

Pipeline

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

Instruction Type Single-cycle

Delay Slots 0

See Also **SADD, SADD2, SADDU4**

Example 1 SADDUS2 .S1 A2, A8, A9

	Before instruction			1 cycle after instruction	
A2	5789 F23Ah	22409 62010 unsigned	A2	5789 F23Ah	22409 62010 unsigned
A8	74B8 4975h	29880 18805 signed	A8	74B8 4975h	9880 18805 signed
A9	XXXX XXXXh		A9	CC41 FFFF	52289 65535 unsigned

Example 2 SADDUS2 .S2 B2, B8, B12

	Before instruction			1 cycle after instruction	
B2	147C 0124h	5244 292 unsigned	B2	147C 0124h	5244 292 unsigned
B8	A051 01A6h	–24495 422 signed	B8	A051 01A6h	–24495 422 signed
B12	XXXX XXXXh		B12	0000 02ACh	0 684 unsigned

SHFL *Shuffle*

Syntax

SHFL (.unit) *src2, dst*
.unit = .M1, .M2

Opcode map field used...	For operand type...	Unit	Opfield
src2 *dst*	xuint uint	.M1, .M2	11100

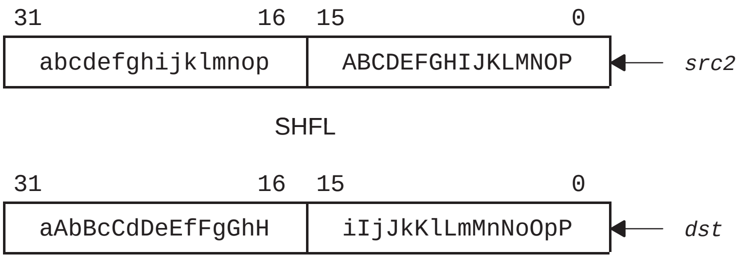
Opcode

31 29	28	27 23	22 18	17 13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src*	*op*	x	0	0	0	0	1	1	1	1	0	0	*s*	*p*
3	1	5	5	5	1	10										1	1

Description

The **SHFL** instruction performs an interleave operation on the two half-words in *src2*. The bits in the lower half-word of *src2* are placed in the even bit positions in *dst*, and the bits in the upper half-word of *src2* are placed in the odd bit positions in *dst*.

As a result, bits 0, 1, 2, ..., 14, 15 of *src2* are placed in bits 0, 2, 4, ... , 28, 30 of *dst*. Likewise, bits 16, 17, 18, .. 30, 31 of *src2* are placed in bits 1, 3, 5, ..., 29, 31 of *dst*.



Note:

The **SHFL** instruction is the exact inverse of the **DEAL** instruction.

Execution

```
if (cond)    {
                 $src2_{31,30,29...16} \rightarrow dst_{31,29,27...1}$
                 $src2_{15,14,13...0} \rightarrow dst_{30,28,26...0}$
             }
```

Pipeline

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

Instruction Type Two-cycle

Delay Slots 1

See Also **DEAL**

Example SHFL .M1 A1,A2

	Before instruction		2 cycles after instruction
A1	B174 6CA4h	A1	B174 6CA4h
A2	XXXX XXXXh	A2	9E52 6E30h

SHLMB

Shift Left and Merge Byte

Syntax

SHLMB (.unit) *src1*,*src2, dst*
.unit = .L1, .L2 or .S1, .S2

Opcode map field used...	For operand type...	Unit	Opfield
src1 *src2* *dst*	u4 xu4 u4	.L1, .L2	1100001
src1 *src2* *dst*	u4 xu4 u4	.S1 .S2	1001

Opcode

.L unit

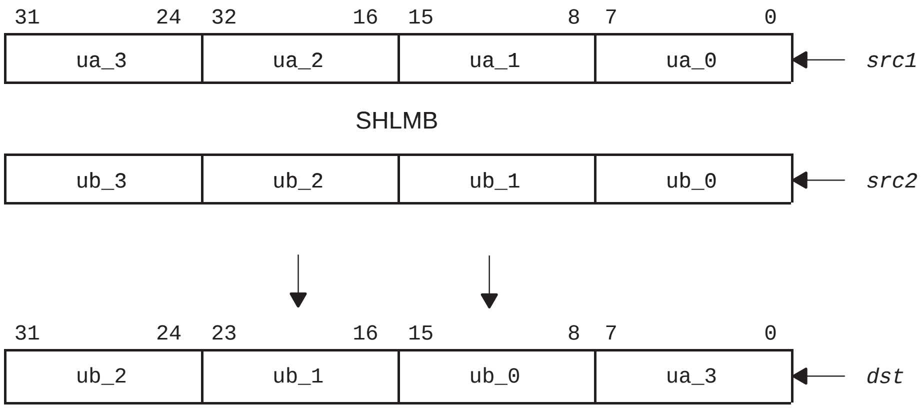
31 29	28	27 23	22 18	17 13	12	11 5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	*op*	1	1	0	*s*	*p*
3	1	5	5	5	1	7	3			1	1

.S Unit

31 29	28	27	22 18	17 13	12	11	10	9 6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	1	1	*op*	1	1	0	0	*s*	*p*
3	1	5	5	5	1	2		4	4				1	1

Description

The **SHLMB** instruction shifts the contents of *src2* left by one byte, and then the most significant byte of *src1* is merged into the least significant byte position. The result is placed in *dst*.



Execution

```
if (cond)    {
             ubyte2(src2) → ubyte3(dst);
             ubyte1(src2) → ubyte2(dst);
             ubyte0(src2) → ubyte1(dst);
             ubyte3(src1) → ubyte0(dst);
             }
else nop
```

Pipeline

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

Instruction Type Single-cycle

Delay Slots 0

See Also **ROTL, SHL, SHRMB, SHR, SHRU**

Example 1 SHLMB .L1 A2, A8, A9

	Before instruction		1 cycle after instruction
A2	`3789 F23Ah`	A2	`3789 F23Ah`
A8	`04B8 4975h`	A8	`04B8 4975h`
A9	`XXXX XXXX h`	A9	`B849 7537h`

Example 2 SHLMB .S2 B2,B8, B12

	Before instruction		1 cycle after instruction
B2	`0124 2451h`	B2	`0124 2451h`
B8	`01A6 A051h`	B8	`01A6 A051h`
B12	`XXXX XXXX h`	B12	`A6A0 5101h`

SHR2 Shift Right, Signed Packed 16-Bit

Syntax

SHR2 (.unit) *src2,src1, dst*
.unit = .S1, .S2

Opcode map field used...	For operand type...	Unit	Opfield
src1 *src2* *dst*	uint xs2 s2	.S1 .S2	0111
src1 *src2* *dst*	ucst xs2 s2	.S1 .S2	011000

Opcode

.S unit (uint form)

31 29	28	27 23	22 18	17 13	12	11	10	9 6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	1	1	*op*	1	1	0	0	*s*	*p*
3	1	5	5	5	1	2		4	4				1	1

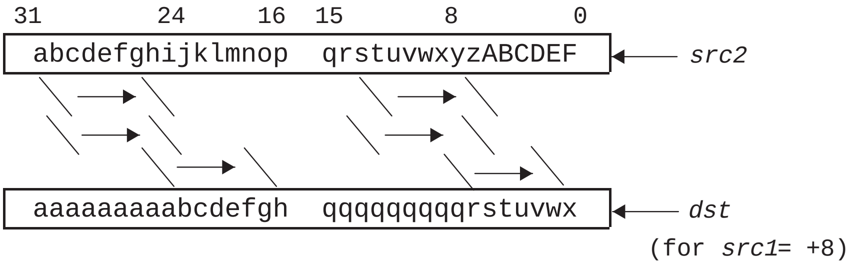
.S unit (cst form)

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

Description

The **SHR2** instruction performs an arithmetic shift right on signed, packed 16-bit quantities. The values in *src2* are treated as signed, packed 16-bit quantities. The lower five bits of *src1* or *ucst5* are treated as the shift amount. The results are placed in a signed, packed 16-bit format into *dst*.

For each signed 16-bit quantity in *src2*, the quantity is shifted right by the number of bits specified in the lower five bits of *src1* or *ucst5* . Bits 5 through 31 of *src1* are ignored and may be non-zero. The shifted quantity is sign-extended, and placed in the corresponding position in *dst*. Bits shifted out of the least-significant bit of the signed 16-bit quantity are discarded.


31
24
16
15
8
0
abcdefghijklmnop qrstuvwxyzABCDEF
src2
aaaaaaaaabcdefgh qqqqqqqqqrstuvwx
dst
(for src1= +8)

Note:

If the shift amount specified in *src1* or *ucst5* is the range 16 to 31, the behavior is identical to a shift value of 15.

Execution

```
if (cond)    {
             smsb16(src2) >> src1 → smsb16(dst)
             slsb16(src2) >> src1 → slsb16(dst);
             }
else nop
```

Pipeline

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

Instruction Type Single-cycle

Delay Slots 0

See Also **SHR**, **SHRU2**

Example 1

SHR2 .S2 B2,B4,B5

	Before instruction			1 cycle after instruction	
B2	A6E2 C179h		B2	A6E2 C179h	
B4	1458 3B69h	shift value 9	B4	1458 3B69h	shift value 9
B5	XXXX XXXXh		B5	FFD3 FFE0h	

Example 2

SHR2 .S1 A4,0fh,A5 ; shift value is 15

	Before instruction		1 cycle after instruction
A4	000A 87AFh	A4	000A 87AFh
A5	XXXX XXXXh	A5	0000 FFFFh

SHRMB — *Shift Right and Merge Byte*

Syntax

SHRMB (.unit) *src1*,*src2, dst*
.unit = .L1, .L2 or .S1, .S2

Opcode map field used...	For operand type...	Unit	Opfield
src1 *src2* *dst*	u4 xu4 u4	.L1, .L2	1100010
src1 *src2* *dst*	u4 xu4 u4	.S1, .S2	1010

Opcode

.L unit

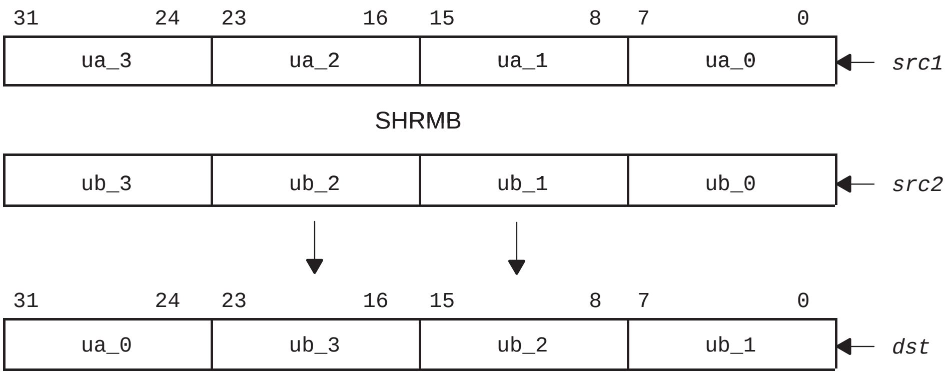
31 29	28	27 23	22 18	17 13	12	11 5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	*op*	1	1	0	*s*	*p*
3	1	5	5	5	1	7	3			1	1

.S Unit

31 29	28	27 23	22 18	17 13	12	11	10	9 6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	1	1	*op*	1	1	0	0	*s*	*p*
3	1	5	5	5	1	2		4	4				1	1

Description

The **SHRMB** instruction shifts the contents of *src2* right by one byte, and then the least significant byte of *src1* is merged into the most significant byte position. The result is placed in *dst*.



Execution

```
if (cond)    {
             ubyte0(src1) → ubyte3(dst);
             ubyte3(src2) → ubyte2(dst);
             ubyte2(src2) → ubyte1(dst);
             ubyte1(src2) → ubyte0(dst);
             }
else nop
```

Pipeline

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

Instruction Type Single-cycle

Delay Slots 0

See Also **ROTL, SHL, SHLMB, SHR, SHRU**

Example 1 SHRMB .L1 A2,A8,A9

	Before instruction		1 cycle after instruction
A2	3789 F23Ah	A2	3789 F23Ah
A8	04B8 4975h	A8	04B8 4975h
A9	XXXX XXXX h	A9	3A04 B849h

Example 2 SHRMB .S2 B2,B8,B12

	Before instruction		1 cycle after instruction
B2	0124 2451h	B2	0124 2451h
B8	01A6 A051h	B8	01A6 A051h
B12	XXXX XXXX h	B12	5101 A6A0h

SHRU2 *Shift Right, Unsigned Packed 16-Bit*

Syntax **SHRU2** (.unit) *src2, src1*, *dst*
.unit = .S1, .S2

Opcode map field used...	For operand type...	Unit	Opfield
src1 *src2* *dst*	uint xu2 u2	.S1 .S2	1000
src1 *src2* *dst*	ucst xu2 u2	.S1 .S2	011001

Opcode

.S unit (uint form)

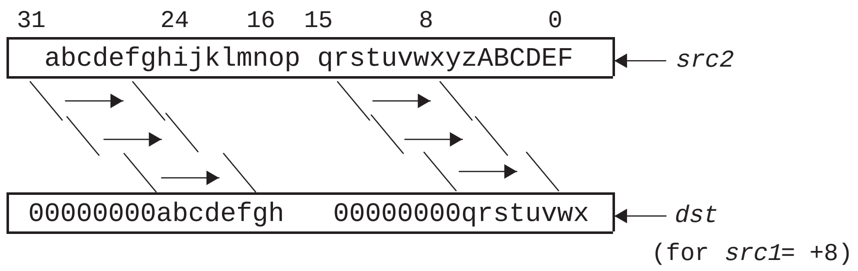
31 29	28	27 23	22 18	17 13	12	11	10	9 6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	1	1	*op*	1	1	0	0	*s*	*p*
3	1	5	5	5	1	2		4	4				1	1

.S unit (cst form)

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

Description The **SHRU2** instruction performs an arithmetic shift right on unsigned, packed 16-bit quantities. The values in *src2* are treated as unsigned, packed 16-bit quantities. The lower five bits of *src1* or *ucst5* are treated as the shift amount. The results are placed in an unsigned, packed 16-bit format into *dst*.

For each unsigned 16-bit quantity in *src2*, the quantity is shifted right by the number of bits specified in the lower five bits of *src1* or *ucst5*. Bits 5 through 31 of *src1* are ignored and may be non-zero. The shifted quantity is zero-extended, and placed in the corresponding position in *dst*. Bits shifted out of the least-significant bit of the signed 16-bit quantity are discarded.


31
24
16
15
8
0
abcdefghijklmnop qrstuvwxyzABCDEF
src2
00000000abcdefgh 00000000qrstuvwx
dst
(for src1= +8)

Note:

If the shift amount specified in *src1* or *ucst5* is in the range of 16 to 31, the *dst* will be set to all zeros.

Execution

```
if (cond)    {
             umsb16(src2) >> src1 → umsb16(dst)
             ulsb16(src2) >> src1 → ulsb16(dst);
             }
else nop
```

Pipeline

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

Instruction Type Single-cycle

Delay Slots 0

See Also **SHR2, SHRU**

Example 1 SHRU2 .S2 B2,B4,B5

	Before instruction			1 cycle after instruction	
B2	A6E2 C179h		B2	A6E2 C179h	
B4	1458 3B69h	Shift value 9	B4	1458 3B69h	Shift value 9
B5	XXXX XXXXh		B5	0053 0060h	

Example 2 SHRU2 .S1 A4,0fh,A5 ; Shift value is 15.

	Before instruction		1 cycle after instruction
A4	000A 87AFh	A4	000A 87AFh
A5	XXXX XXXXh	A5	0000 0001h

SMPY2 *Multiply Signed by Signed, with Left Shift and Saturate, Packed 16-Bit*

Syntax

SMPY2 (.unit) *src1*,*src2, dst*
.unit = .M1, .M2

Opcode map field used...	For operand type...	Unit	Opfield
src1 *src2* *dst*	s2 xs2 ullong	.M1, .M2	00001

Opcode

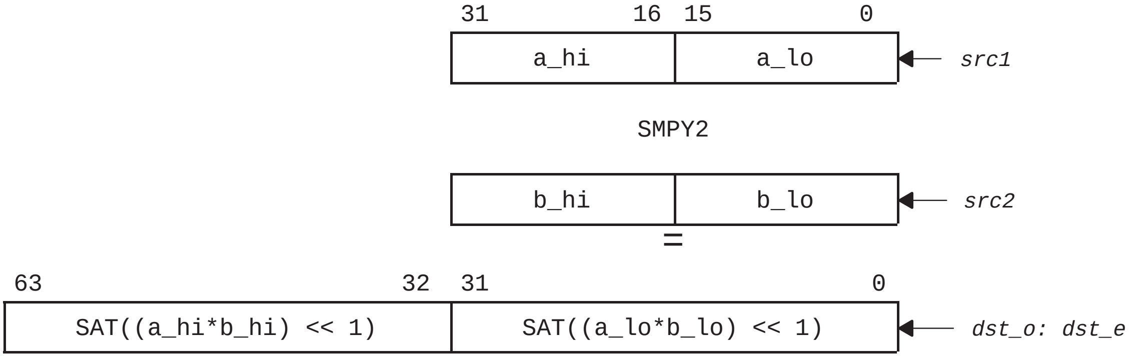
31 29	28	27 23	22 18	17 13	12	11	10 6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	*x*	0	*op*	1	1	0	0	*s*	*p*
3	1	5	5	5	1	1	5	4				1	1

Description

The **SMPY2** instruction performs two 16-bit by 16-bit multiplies between two pairs of signed packed 16-bit values, with an additional left-shift and saturate. The values in *src1* and *src2* are treated as signed, packed 16-bit quantities. The two 32-bit results are written into a 64-bit register pair.

The **SMPY2** instruction produces two 16 x 16 products. Each product is shifted left by one; and if the left-shifted result is equal to 0x80000000, the output value is saturated to 0x7FFFFFFF.

The saturated product of the lower half-words of *src1* and *src2* is written to the even destination register, *dst_e*. The saturated product of the upper half-words of *src1* and *src2* is written to the odd destination register, *dst_o*.



Note:

If either product saturates, the SAT bit is set in the CSR on the cycle that the result is written. If neither product saturates, the SAT bit in the CSR is left unaffected.

This instruction helps reduce the number of instructions required to perform two 16-bit by 16-bit saturated multiplies on both the lower and upper halves of two registers.

The following code:

```
SMPY  .M1  A0, A1, A2
SMPYH .M1  A0, A1, A3
```

may be replaced by:

```
SMPY2  .M1  A0, A1, A3:A2
```

Execution

```
if (cond)    {
             SAT((lsb16(src1) x lsb16(src2)) << 1) → dst_e ;
             SAT((msb16(src1) x msb16(src2)) <<1) → dst_o
             }
else nop
```

Pipeline

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

Instruction Type Four-cycle

Delay Slots 3

See Also **MPY2, SMPY, SMPYH**

Example 1 SMPY2 .M1 A5,A6,A9:A8

	Before instruction			**4 cycles after instruction**	
A5	6A32 1193h	27186 4499	A5	6A32 1193h	27186 4499
A6	B174 6CA4h	–20108 27812	A6	B174 6CA4h	–20108 27812
A9:A8	XXXX XXXXh	XXXX XXXXh	A9:A8	BED5 6150h	0EEA 8C58h
				–1,093,312,176	250,252,376

Example 2 SMPY2 .M2 B2, B5, B9:B8

	Before instruction			4 cycles after instruction	
B2	1234 3497h	4660 13463	B2	1234 3497h	4660 13463
B5	21FF 50A7h	8703 20647	B5	21FF 50A7h	8703 20647
			.		
B9:B8	XXXX XXXXh	XXXX XXXXh	B9:B8	04D5 AB98h	2122 FD02h
				81,111,960	555,941,122

SPACK2 Saturate and Pack into Signed Packed 16-Bit

Syntax

SPACK2 (.unit) *src1*,*src2, dst*
.unit = .S1, .S2

Opcode map field used...	For operand type...	Unit	Opfield
src1 *src2* *dst*	int xint s2	.S1 .S2	0010

Opcode

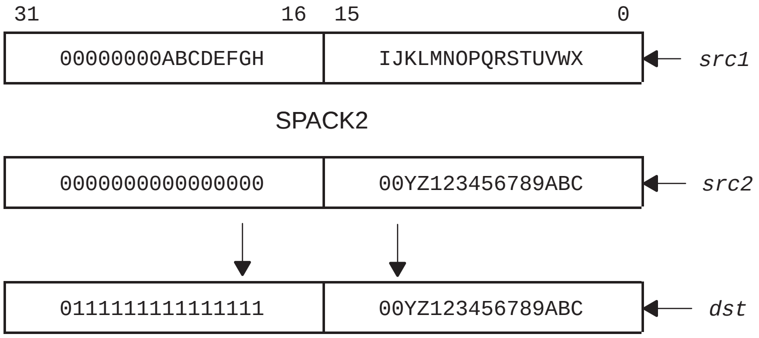
31 29	28	27 23	22 18	17 13	12	11	10	9 6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	1	1	*op*	1	1	0	0	*s*	*p*
3	1	5	5	5	1	2		4	4				1	1

Description

The **SPACK2** instruction takes two signed 32-bit quantities in *src1* and *src2* and saturates them to signed 16-bit quantities. The signed 16-bit results are then packed into a signed, packed 16-bit format and written to *dst*. Specifically, the saturated 16-bit signed value of *src1* is written to the upper half word of *dst,* and the saturated 16-bit signed value of *src2* is written to the lower half word of *dst*.

Saturation is performed on each input value independently. The input values start as signed 32-bit quantities, and are saturated to 16-bit quantities according to the following rules:

- ❑ If the value is in the range -2^{15} to $2^{15}-1$, inclusive, then no saturation is performed and the value is merely truncated to 16 bits.
- ❑ If the value is greater than $2^{15}-1$, then the result is set to $2^{15}-1$.
- ❑ If the value is less than -2^{15}, then the result is set to -2^{15}.



This instruction is useful in code which manipulates 16-bit data at 32-bit precision for its intermediate steps, but which requires the final results to be in a 16-bit representation. The saturate step ensures that any values outside the signed 16-bit range are clamped to the high or low end of the range before being truncated to 16 bits.

Note:

This operation is performed on each 16-bit value separately. This instruction does not affect the *SAT* bit in the *CSR*.

Execution

```
if (cond)     {
                if src2 > 0x00007FFF, then 0x7FFF → lsb16(dst) or
                if src2 < 0xFFFF8000, then 0x8000 → lsb16(dst)
                     else truncate(src2) → lsb16(dst);
                if src1 > 0x00007FFF, then 0x7FFFF → msb16(dst) or
                if src1 < 0xFFFF8000, then 0x8000 → msb16(dst)
                     else truncate(src1) → msb16(dst);
                }
else  nop
```

Pipeline

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

Instruction Type Single-cycle

Delay Slots 0

See Also **PACK2, PACKH2, PACKHL2, PACKLH2, SPACKU4**

Example 1 SPACK2 .S1 A2,A8,A9

	Before instruction			1 cycle after instruction	
A2	3789 F23Ah	931,787,322	A2	3789 F23Ah	931,787,322
A8	04B8 4975h	79,186,293	A8	04B8 4975h	79,186,293
A9	XXXX XXXX h		A9	7FFF 7FFFh	32767 32767

Example 2 SPACK2 .S2 B2,B8,B12

	Before instruction			1 cycle after instruction	
B2	A124 2451h	–1,591,466,927	B2	A124 2451h	–1,591,466,927
B8	01A6 A051h	27,697,233	B8	01A6 A051h	27,697,233
B12	XXXX XXXX h		B12	8000 7FFFh	–32768 32767

SPACKU4 *Saturate and Pack into Unsigned Packed 8-Bit*

Syntax

SPACKU4 (.unit) *src1*,*src2, dst*
.unit = .S1, .S2

Opcode map field used...	For operand type...	Unit	Opfield
src1 *src2* *dst*	s2 xs2 u4	.S1 .S2	0100

Opcode

31 29	28	27 23	22 18	17 13	12	11	10	9 6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	1	1	*op*	1	1	0	0	*s*	*p*
3	1	5	5	5	1	2		4	4				1	1

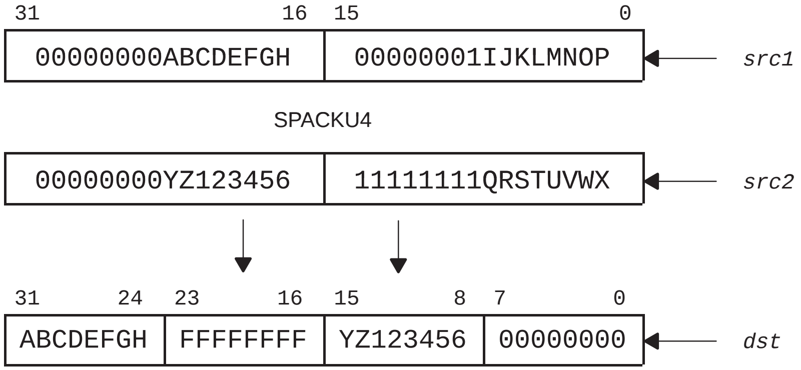
Description

The **SPACKU4** instruction takes four signed 16-bit values and saturates them to unsigned 8-bit quantities. The values in *src1* and *src2* are treated as signed, packed 16-bit quantities. The results are written into *dst* in an unsigned, packed 8-bit format.

Each signed 16-bit quantity in *src1* and *src2* is saturated to an unsigned 8-bit quantity as described below. The resulting quantities are then packed into an unsigned, packed 8-bit format. Specifically, the upper half word of *src1* is used to produce the most significant byte of *dst*. The lower half of *src1* is used to produce the second most significant byte (bits 16 to 23) of *dst*. The upper half word of *src2* is used to produce the third most significant byte (bits 8 to 15) of *dst*. The lower half word of *src2* is used to produce the least significant byte of *dst*.

Saturation is performed on each signed 16-bit input independently, producing separate unsigned 8-bit results. For each value, the following tests are applied:

- ❑ If the value is in the range *0* to $2^{8} - 1$, inclusive, then no saturation is performed and the result is truncated to 8 bits.
- ❑ If the value is greater than $2^{8} - 1$, then the result is set to $2^{8} - 1$.
- ❑ If the value is less than *0*, the result is set to *0*.



This instruction is useful in code which manipulates 8-bit data at 16-bit precision for its intermediate steps, but which requires the final results to be in an 8-bit representation. The saturate step ensures that any values outside the unsigned 8-bit range are clamped to the high or low end of the range before being truncated to 8 bits.

Note:

This operation is performed on each 8-bit quantity separately. This instruction does not affect the *SAT* bit in the *CSR*.

Execution

```
if (cond)      {
                  if msb16(src1) >> 0x00007FFF, then 0x7F → ubyte3(dst) or
                  if msb16(src1) << 0xFFFF8000, then 0 → ubyte3(dst)
                        else truncate(msb16(src1)) → ubyte3(dst);
                  if lsb16(src1) >> 0x00007FFF, then 0x7F → ubyte2(dst) or
                  if lsb16(src1) << 0xFFFF8000, then 0 → ubyte2(dst)
                        else truncate(lsb16(src1)) → ubyte2(dst);
                  if msb16(src2) >> 0x00007FFF, then 0x7F → ubyte1(dst) or
                  if msb16(src2) << 0xFFFF8000, then 0 → ubyte1(dst)
                        else truncate(msb16(src2)) → ubyte1(dst);
                  if lsb16(src2) >> 0x00007FFF, then 0x7F → ubyte0(dst) or
                  if lsb16(src2) << 0xFFFF8000, then 0 → ubyte0(dst)
                        else truncate(lsb16(src2)) → ubyte0(dst);
                  }
else  nop
```

Pipeline

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

Instruction Type Single-cycle

Delay Slots 0

See Also **PACKH4, PACKL4, SPACK2**

Example 1 SPACKU4 .S1 A2,A8,A9

	Before instruction			1 cycle after instruction	
A2	3789 F23Ah	14217 –3526	A2	3789 F23Ah	14217 –3526
A8	04B8 4975h	1208 18805	A8	04B8 4975h	1208 18805
A9	XXXX XXXXh		A9	FF 00 FF FFh	255 0 255 255

Example 2 SPACKU4 .S2 B2,B8,B12

	Before instruction			1 cycle after instruction	
B2	A124 2451h	–24284 9297	B2	A124 2451h	–24284 9297
B8	01A6 A051h	422 –24495	B8	01A6 A051h	422 –24495
B12	XXXX XXXXh		B12	00 FF FF 00h	0 255 255 0

SSHVL *Variable Shift Left, Signed*

Syntax

SSHVL(.unit) *src2*,*src1, dst*
.unit = .M1, .M2

Opcode map field used...	For operand type...	Unit	Opfield
src1 *src2* *dst*	int xint int	.M1, .M2	11100

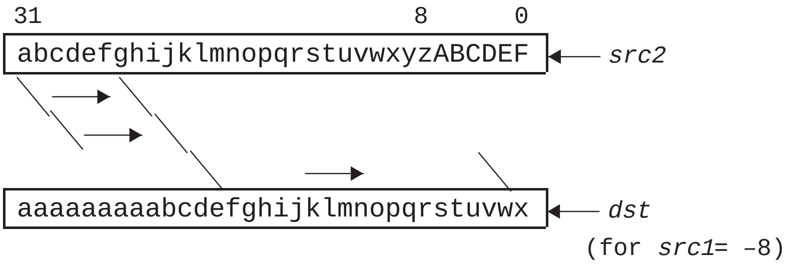
Opcode

31 29	28	27 23	22 18	17 13	12	11	10 6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	0	*op*	1	1	0	0	*s*	*p*
3	1	5	5	5	1	1	5	4				1	1

Description

The **SSHVL** instruction shifts the signed 32-bit value in *src2* to the left or right by the number of bits specified by *src1,* and places the result in *dst*.

The *src1* argument is treated as a 2s-complement shift value which is automatically limited to the range –31 to 31. If *src1* is positive, *src2* is shifted to the left. If *src1* is negative, *src2* is shifted to the right by the absolute value of the shift amount, with the sign-extended shifted value being placed in *dst*. It should also be noted that when *src1* is negative, the bits shifted right past bit 0 are lost.


31
8
0
abcdefghijklmnopqrstuvwxyzABCDEF
src2
aaaaaaaaabcdefghijklmnopqrstuvwx
dst
(for src1= –8)

Note:

If the shifted value is saturated, then the SAT bit is set in CSR in the same cycle that the result is written. If the shifted value is not saturated, then the SAT bit is unaffected.

Execution

```
if (cond)    {
                 if   0 < = src1 < = 31 then
                 SAT(src2 << src1) → dst ;
                 if  –31 < = src1 < 0 then
                 (src2 >> abs(src1)) → dst;
                 if src1 > 31 then
                 SAT(src2 << 31) → dst;
                 if src1 < –31 then
                 (src2 >> 31) → dst
                 }
else  nop
```

Saturation is performed (shown as sat above) when the value is shifted left under the following conditions:

- If the shifted value is in the range -2^{31} to $2^{31} - 1$, inclusive, then no saturation is performed, and the result is truncated to 32 bits.
- If the shifted value is greater than $2^{31} - 1$, then the result is saturated to $2^{31} - 1$.
- If the shifted value is less than -2^{31}, then the result is saturated to -2^{31}.

Pipeline

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

Instruction Type Two-cycle

Delay Slots 1

See Also **SHL, SHRU, SSHL, SSHVR**

Example 1

SSHVL .M2 B2, B4, B5

Before instruction			2 cycles after instruction		
B2	FFFF F000h		B2	FFFF F000h	
B4	FFFF FFE1h	–31	B4	FFFF FFE1h	–31
B5	XXXX XXXXh		B5	FFFF FFFFh	

Example 2

SSHVL .M1 A2,A4,A5

Before instruction			2 cycles after instruction		
A2	F14C 2108h		A2	F14C 2108h	
A4	0000 0001Fh	31	A4	0000 0001Fh	31
A5	XXXX XXXXh		A5	8000 0000h	Note: Saturated to most negative value

Example 3

SSHVL .M2 B12, B24, B25

Before instruction			2 cycles after instruction		
B12	187A 65FCh		B12	187A 65FCh	
B24	FFFF FFFFh	–1	B24	FFFF FFFFh	–1
B25	XXXX XXXXh		B25	03CD 32FEh	

SSHVR Variable Shift Right, Signed

Syntax

SSHVR (.unit) *src2*,*src1, dst*
.unit = .M1, .M2

Opcode map field used...	For operand type...	Unit	Opfield
src1 *src2* *dst*	int xint int	.M1, .M2	11101

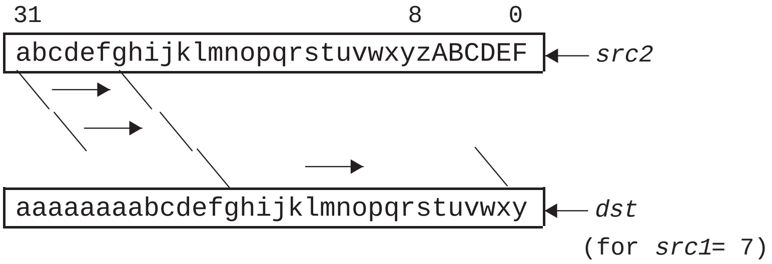
Opcode

31 29	28	27 23	22 18	17 13	12	11	10 6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	0	*op*	1	1	0	0	*s*	*p*
3	1	5	5	5	1	1	5	4				1	1

Description

The **SSHVR** instruction shifts the signed 32-bit value in *src2* to the left or right by the number of bits specified by *src1,* and places the result in *dst*.

The *src1* argument is treated as a 2s-complement shift value that is automatically limited to the range –31 to 31. If *src1* is positive, *src2* is shifted to the right by the value specified with the sign-extended shifted value being placed in *dst*. It should also be noted that when *src1* is positive, the bits shifted right past bit 0 are lost. If *src1* is negative, *src2* is shifted to the left by the absolute value of the shift amount value and the result is placed in *dst*.


31
8
0
abcdefghijklmnopqrstuvwxyzABCDEF
src2
aaaaaaaabcdefghijklmnopqrstuvwxy
dst
(for src1= 7)

Note:

If the shifted value is saturated, then the SAT bit is set in CSR in the same cycle that the result is written. If the shifted value is not saturated, then the SAT bit is unaffected.

Execution

```
if (cond)     {
                 if   0 < = src1 < = 31 then
                (src2 >> src1) → dst;
                 if –31 < = src1< 0 then
                SAT(src2 << abs(src1)) → dst;
                if src1 > 31 then
                (src2 >> 31) → dst;
                if src1 < –31 then
                SAT(src2 << 31) → dst
                }
else  nop
```

Saturation (shown as sat above) is performed when the value is shifted left under the following conditions:

- If the shifted value is in the range $–2^{31}$ to $2^{31} – 1$, inclusive, then no saturation is performed, and the result is truncated to 32 bits.
- If the shifted value is greater than $2^{31} – 1$, then the result is saturated to $2^{31} – 1$.
- If the shifted value is less than $– 2^{31}$, then the result is saturated to $– 2^{31}$.

Pipeline

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

Instruction Type Two-cycle

Delay Slots 1

See Also **SHL, SHRU, SSHL, SSHVL**

Example 1

SSHVR .M2 B2,B4,B5

	Before instruction			2 cycles after instruction	
B2	FFFF F000h		B2	FFFF F000h	
B4	FFFF FFE1h	–31	B4	FFFF FFE1h	–31
B5	XXXX XXXXh		B5	8000 0000h	Note: Saturates to most negative value

Example 2

SSHVR .M1 A2,A4,A5

	Before instruction			2 cycles after instruction	
A2	F14C 2108h		A2	F14C 2108h	
A4	0000 0001Fh	31	A4	0000 0001Fh	31
A5	XXXX XXXXh		A5	FFFF FFFFh	

Example 3 SSHVR .M2 B12, B24, B25

	Before instruction			2 cycles after instruction	
B12	187A 65FCh		B12	187A 65FCh	
B24	FFFF FFFFh	–1	B24	FFFF FFFFh	–1
B25	XXXX XXXXh		B25	30F4 CBF8h	

STDW Store Double Word

Syntax

STDW (.unit) src,*mem
.unit = .D1, .D2

Opcode map field used...	For operand type...	Unit
srcd *baseR* *offset*	ullong uint uint	.D1, .D2,
srcd *baseR* *offset*	ullong uint ucst5	.D1 .D2

Opcode

31 29	28	27 23	22 18	17 13	12 9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*baseR*	*offset*	*mode*	*r*	*y*	1	0	0	0	1	*s*	*p*
3	1	5	5	5	4	1	1		3		2		1	1

Description

The **STDW** instruction stores a 64-bit quantity to memory from a 64-bit register, *srcd*. The table below describes the addressing generator options. Alignment to a 64-bit boundary is required. The effective 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.

Both *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.

The *offsetR/ucst5* is scaled by a left shift of 3 bits. After scaling, *offsetR/ucst5* is added to, or subtracted from, *baseR*. For the pre-increment, pre-decrement, positive offset, and negative offset address generator options, the result of the calculation is the address to be accessed in memory. For post-increment or post-decrement addressing, the value of *baseR* before the addition or subtraction is the address to be accessed from 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.

The *srcd* pair can be in either register file, regardless of the .D unit or baseR or offsetR used. The s bit determines which file *srcd* will be loaded from: s = 0 indicates *srcd* will be in the A register file and s = 1 indicates *srcd* will be in the B register file. r is always zero.

Table 5–11. STDW 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

Assembler Notes

When no bracketed register or constant is specified, the assembler defaults increments and decrements to 1 and offsets to 0. Stores that do no modification to the *baseR* can use the assembler syntax *R. Square brackets, [ ], indicate that the *ucst5* offset is left-shifted by 3 for doubleword stores.

Parentheses, ( ), can be used to tell the assembler that the offset is a non-scaled, constant offset. The assember right shifts the constant by 3 bits for double word stores before using it for the *ucst5* field. After scaling by the STDW instruction, this results in the same constant offset as the assembler source if the least significant three bits are zeros.

For example, **STDW** (.unit) src, *+baseR (16) represents an offset of 16 bytes (2 double words), and the assembler writes out the instruction with ucst5 = 2. **STDW** (.unit) src, *+baseR [16] represents an offset of 16 double words, or 128 bytes, and the assembler writes out the instruction with ucst5 = 16.

Either brackets or parentheses must be typed around the specified offset if the optional offset parameter is used. The register pair syntax always places the odd-numbered register first, a colon, followed by the even-numbered register (that is, A1:A0, B1:B0, A3:A2, B3:B2, etc.).

Execution

```
if (cond)    {
               src → mem
             }
else  nop
```

Pipeline

Pipeline Stage	**E1**
Read	*baseR, offsetR, srcd*
Written	*baseR*
Unit in use	.D

Instruction Type Store

Delay Slots 0

See Also **LDDW, STW**

Example 1 STDW .D1 A3:A2,*A0++

	Before instruction			**1 cycle after instruction**	
A0	0000 1000h		A0	0000 1008h	
A3:A2	A176 3B28h	6041 AD65h	A3:A2	A176 3B28h	6041 AD65h

Byte Memory Address	1009	1008	1007	1006	1005	1004	1003	1002	1001	1000
Data Value Before Store	00	00	00	00	00	00	00	00	00	00
Data Value After Store	00	00	A1	76	3B	28	60	41	AD	65

Example 2

STDW .D1 A3:A2, *A0++

	Before instruction			1 cycle after instruction	
A0	0000 1004h		A0	0000 100Ch	
A3:A2	A176 3B28h	6041 AD65h	A3:A2	A176 3B28h	6041 AD65h

Byte Memory Address	100D	100C	100B	100A	1009	1008	1007	1006	1005	1004	1003
Data Value Before Store	00	00	00	00	00	00	00	00	00	00	00
Data Value After Store	00	00	A1	76	3B	28	60	41	AD	65	00

STNDW Store Non-Aligned Double Word

Syntax

STNDW (.unit) src,*mem
.unit = .D1, .D2

Opcode map field used...	For operand type...	Unit
srcd *baseR* *offset*	ullong uint uint	.D1, .D2
srcd *baseR* *offset*	ullong uint ucst5	.D1, .D2

Opcode

31 29	28	27 24	23	22 18	17 13	12 9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*sc*	*baseR*	*offset*	*mode*	*r*	*y*	1	1	1	0	1	*s*	*p*
3	1	4	1	5	5	4	1	1	3			2		1	1

Description

The **STNDW** instruction stores a 64-bit quantity to memory from a 64-bit register pair, *srcd*. The table below describes the addressing generator options. The **STNDW** instruction may write a 64-bit value to any byte boundary. Thus alignment to a 64-bit boundary is not required. The effective 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*).

Both *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.

The **STNDW** instruction suppports both scaled offsets and non-scaled offsets. The *sc* field is used to indicate whether the *offsetR/ucst5* is scaled or not. If sc is 1 (scaled), the *offsetR/ucst5* is shifted left 3 bits before adding or subtracting from the baseR. If *sc* is 0 (non-scaled), the *offsetR/ucst5* is not shifted before adding to or subtracting from the baseR. For the pre-increment, pre-decrement, positive offset, and negative offset address generator options, the result of the calculation is the address to be accessed in memory. For post-increment or post-decrement addressing, the value of baseR before the addition or subtraction is the address to be accessed from 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.

The *srcd* pair can be in either register file, regardless of the .D unit or baseR or offsetR used. The s bit determines which file *srcd* will be loaded from: s = 0 indicates *srcd* will be in the A register file and s = 1 indicates *srcd* will be in the B register file. r is always zero.

Table 5–12. STNDW 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

Note:

No other memory access may be issued in parallel with a non-aligned memory access. The other .D unit can be used in parallel, as long as it is not performing a memory access.

Assembler Notes

When no bracketed register or constant is specified, the assembler defaults increments and decrements to 1, and offsets to 0. Loads that do no modification to the baseR can use the assembler syntax *R. Square brackets, [ ], indicate that the ucst5 offset is left-shifted by 3 for double word stores.

Parentheses, ( ), can be used to tell the assembler that the offset is a non-scaled offset.

For example, **STNDW** (.unit) *src*, *+baseR (12) represents an offset of 12 bytes and the assembler writes out the instruction with *offsetC* = 12 and *sc* = 0.

STNDW (.unit) *src*, *+baseR [16] represents an offset of 16 double words, or 128 bytes, and the assembler writes out the instruction with *offsetC* = 16 and *sc* = 1.

Either brackets or parentheses must be typed around the specified offset if the optional offset parameter is used.

Execution

```
if (cond)    {
                 src → mem
                 }
else  nop
```

Pipeline

Pipeline Stage	**E1**	**E2**	**E3**
Read	*baseR, offsetR*, *srcd*		
Written	*baseR*		
Unit in use	.D		

Instruction Type Store

Delay Slots 0

See Also **LDNW, LDNDW, STNW**

Example 1 STNDW .D1 A3:A2, *A0++

Before instruction			**1 cycle after instruction**		
A0	0000 1001h		A0	0000 1009h	
A3	A176 3B28h	6041 AD65h	A3:A2	A176 3B28h	6041 AD65h

Byte Memory Address	1009	1008	1007	1006	1005	1004	1003	1002	1001	1000
Data Value Before Store	00	00	00	00	00	00	00	00	00	00
Data Value After Store	00	A1	76	3B	28	60	41	AD	65	00

Example 2 STNDW .D1 A3:A2, *A0++

	Before instruction			1 cycle after instruction	
A0	0000 1003h		A0	0000 100Bh	
A3:A2	A176 3B28h	6041 AD65h	A3:A2	A176 3B28h	6041 AD65h

Byte Memory Address	100B	100A	1009	1008	1007	1006	1005	1004	1003	1002	1001	1000
Data Value Before Store	00	00	00	00	00	00	00	00	00	00	00	00
Data Value After Store	00	A1	76	3B	28	60	41	AD	65	00	00	00

STNW Store Non-Aligned Word

Syntax

STNW(.unit) src,*mem
.unit = .D1, .D2

Opcode map field used...	For operand type...	Unit
src *baseR* *offset*	uint uint uint	.D1, .D2
src *baseR* *offset*	uint uint ucst5	.D1, .D2

Opcode

31 29	28	27 23	22 18	17 13	12 9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*baseR*	*offset*	*mode*	*r*	*y*	1	0	1	0	1	*s*	*p*
3	1	5	5	5	4	1	1		3		2		1	1

Description

The **STNW** instruction stores a 32-bit quantity to memory from a 32-bit register, *src*. The table below describes the addressing generator options. The **STNW** instruction may write a 32-bit value to any byte boundary. Thus alignment to a 32-bit boundary is not required. The effective 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*).

Both *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.

The *offsetR/ucst5* is scaled by a left shift of 2 bits. After scaling, *offsetR/ucst5* is added to, or subtracted from, *baseR*. For the pre-increment, pre-decrement, positive offset, and negative offset address generator options, the result of the calculation is the address to be accessed in memory. For post-increment or post-decrement addressing, the value of *baseR* before the addition or subtraction is the address to be accessed from 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.

The *src* can be in either register file, regardless of the .D unit or baseR or offsetR used. The s bit determines which file *src* will be loaded from: s = 0 indicates *src* will be in the A register file and s = 1 indicates *src* will be in the B register file. is always zero.

Table 5–13. STNW 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

Note:

No other memory access may be issued in parallel with a non-aligned memory access. The other .D unit can be used in parallel as long as it is not performing memory access.

Assembler Notes

When no bracketed register or constant is specified, the assembler defaults increments and decrements to 1 and offsets to 0. Loads that do no modification to the baseR can use the assembler syntax *R. Square brackets, [ ], indicate that the ucst5 offset is left-shifted by 2 for word stores.

Parentheses, ( ), can be used to tell the assembler that the offset is a non-scaled, constant offset. The assember right shifts the constant by 2 bits for word stores before using it for the ucst5 field. After scaling by the STNW instruction, this results in the same constant offset as the assembler source if the least significant two bits are zeros.

For example, **STNW** (.unit) *src*,*+baseR (12) represents an offset of 12 bytes (3 words), and the assembler writes out the instruction with ucst5 = 3.

STNW (.unit) *src*,*+baseR [12] represents an offset of 12 words, or 48 bytes, and the assembler writes out the instruction with ucst5 = 12.

Either brackets or parentheses must be typed around the specified offset if the optional offset parameter is used.

Execution

```
if (cond) {
            src → mem
           }
else  nop
```

Pipeline

Pipeline Stage	E1	E2	E3
Read	*baseR, offsetR*, *srcd*		
Written	*baseR*		
Unit in use	.D		

Instruction Type Store

Delay Slots 0

See Also **LDNW, LDNDW, STNDW**

Example 1

STNW .D1 A3, *A0++

	Before instruction		1 cycle after instruction
A0	0000 1001h	A0	0000 1005h
A3	A176 3B28h	A3	A176 3B28h

Byte Memory Address	1007	1006	1005	1004	1003	1002	1001	1000
Data Value Before Store	00	00	00	00	00	00	00	00
Data Value After Store	00	00	00	A1	76	3B	28	00

Example 2

STNW .D1 A3, *A0++

	Before instruction		1 cycle after instruction
A0	0000 1003h	A0	0000 1007h
A3	A176 3B28h	A3	A176 3B28h

Byte Memory Address	1007	1006	1005	1004	1003	1002	1001	1000
Data Value Before Store	00	00	00	00	00	00	00	00
Data Value After Store	00	A1	76	3B	28	00	00	00

SUB2 *Two 16-Bit Integer Subtractions on Upper and Lower Register Halves*

Syntax

SUB2(.unit) *src1*,*src2, dst*
.unit = .L1, .L2, .S1, .S2, .D1, .D2

Opcode map field used...	For operand type...	Unit	Opfield
src1 *src2* *dst*	i2 xi2 i2	.L1, .L2	0000100
src1 *src2* *dst*	i2 xi2 i2	.S1, .S2	010001
src1 *src2* *dst*	i2 xi2 i2	.D1, .D2	0101

Opcode

.L unit

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

.S unit

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

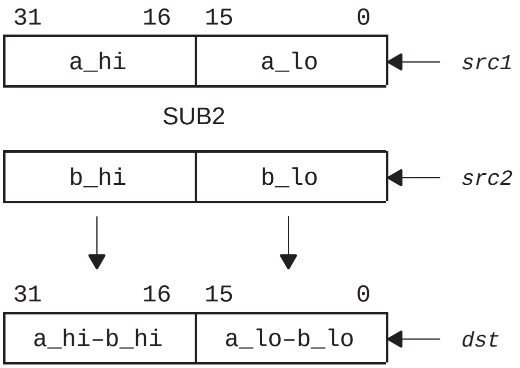
.D unit

31 29	28	27 23	22 18	17 13	12	11	10	9 6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	1	0	*op*	1	1	0	0	*s*	*p*
3	1	5	5	5	1	2		4	4				1	1

Description

In the **SUB2** instruction, the upper and lower halves of *src2* are subtracted from the upper and lower halves of *src1* and the result is placed in *dst*. Any borrow from the lower-half subtraction does not affect the upper-half subtraction. Specifically, the upper-half of *src2* is subtracted from the upper-half of *src1* and placed in the upper-half of *dst*. The lower-half of *src2* is subtracted from the lower-half of *src1* and placed in the lower-half of *dst*.

This is the same **SUB2** instruction found on the C62, but with the added flexibility of being able to perform this instruction on the .L and .D units as well as the .S unit.



Note:

Unlike the **SUB** instruction, the argument ordering on the .D unit form of .S2 is consistent with the argument ordering for the .L and .S unit forms.

Execution

```
if (cond)    {
              (lsb16(src1) – lsb16(src2)) → lsb16(dst);
              (msb16(src1) – msb16(src2)) → msb16(dst);
              }
else  nop
```

Pipeline

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

Instruction Type Single-cycle

Delay Slots 0

See Also **ADD2, SUB, SUB4**

Example 1 SUB2 .S1 A3, A4, A5

	Before instruction			1 cycle after instruction	
A3	1105 6E30h	4357 28208	A3	1105 6E30h	4357 28208
A4	1105 6980h	4357 27008	A4	1105 6980h	4357 27008
A5	XXXX XXXXh		A5	0000 04B0h	0 1200

Example 2 SUB2 .D2 B2, B8, B15

	Before instruction			1 cycle after instruction	
B2	F23A 3789h	–3526 14217	B2	F23A 3789h	–3526 14217
B8	04B8 6732h	1208 26418	B8	04B8 6732h	1208 26418
B15	XXXX XXXXh		B15	ED82 D057h	–4734 –12201

SUB4 *Subtract Without Saturation, Signed Packed 8-Bit*

Syntax

SUB4 (.unit) *src1*,*src2, dst*
.unit = .L1, .L2

Opcode map field used...	For operand type...	Unit	Opfield
src1 *src2* *dst*	i4 xi4 i4	.L1, .L2	1100110

Opcode

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

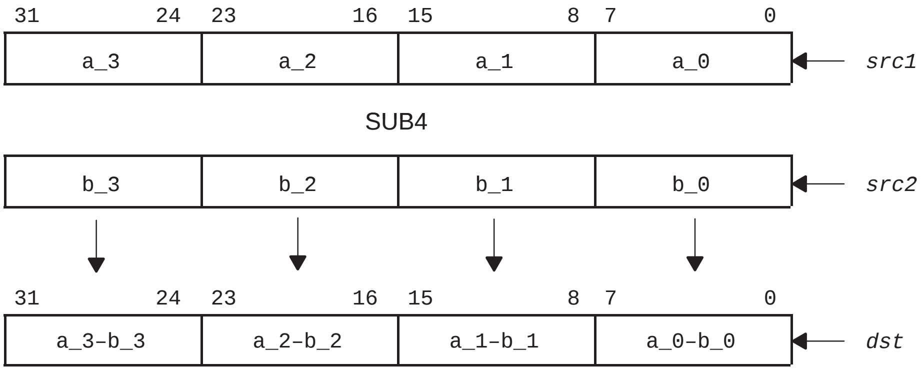
Description

The **SUB4** instruction performs 2s-complement subtraction between packed 8-bit quantities. The values in *src1* and *src2* are treated as packed 8-bit quantities, and the results are written into *dst* in a packed 8-bit format.

For each pair of 8-bit quantities in *src1* and *src2*, the difference between the 8-bit value from *src1* and the 8-bit value from *src2* is calculated to produce an 8-bit result. The result is placed in the corresponding position in *dst*.

Specifically, the difference between *src1* byte0 and *src2* byte0 is placed in byte0 of *dst*. The difference between *src1* byte1 and *src2* byte1 is placed in byte1 of *dst*. The difference between *src1* byte2 and *src2* byte2 is placed in byte2 of *dst*. The difference between *src1* byte3 and *src2* byte3 is placed in byte3 of *dst*.

No saturation is performed.



Execution

```
if (cond)    {
                 (byte0(src1) – byte0(src2)) → byte0(dst);
                 (byte1(src1) – byte1(src2)) → byte1(dst);
                 (byte2(src1) – byte2(src2)) → byte2(dst);
                 (byte3(src1) – byte3(src2)) → byte3(dst);
             }
else  nop
```

Pipeline

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

Instruction Type Single-cycle

Delay Slots 0

See Also **ADD4, SUB, SUB2**

Example SUB4 .L1 A2, A8, A9

	Before instruction		1 cycle after instruction
A2	37 89 F2 3Ah	A2	37 89 F2 3Ah
A8	04 B8 49 75h	A8	04 B8 49 75h
A9	XXXX XXXXh	A9	33 D1 A9 C5h

SUBABS4 *Subtract With Absolute Value, Unsigned Packed 8-Bit*

Syntax

SUBABS4 (.unit) *src1*,*src2, dst*
.unit = .L1, .L2

Opcode map field used...	For operand type...	Unit	Opfield
src1 *src2* *dst*	u4 xu4 u4	.L1, .L2	1011010

Opcode

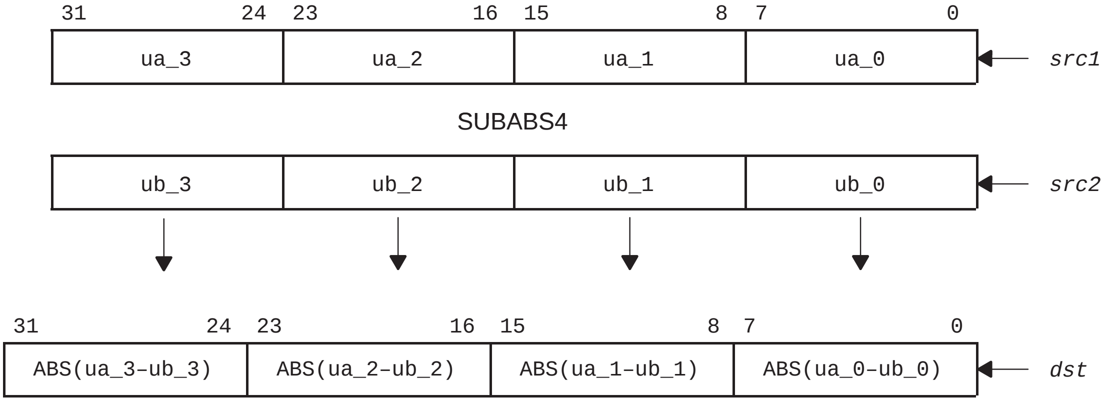
31 29	28	27 23	22 18	17 13	12	11 5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	*op*	1	1	0	*s*	*p*
3	1	5	5	5	1	7	3			1	1

Description

The **SUBABS4** instruction calculates the absolute value of the differences between the packed 8-bit data contained in the source registers. The values in *src1* and *src2* are treated as unsigned, packed 8-bit quantities. The result is written into *dst* in an unsigned, packed 8-bit format.

For each pair of unsigned 8-bit values in *src1* and *src2*, the absolute value of the difference is calculated. This result is then placed in the corresponding position in *dst*.

Specifically, the absolute value of the difference between *src1* byte0 and *src2* byte0 is placed in byte0 of *dst*. The absolute value of the difference between *src1* byte1 and *src2* byte1 is placed in byte1 of *dst*. The absolute value of the difference between *src1* byte2 and *src2* byte2 is placed in byte2 of *dst*. And the absolute value of the difference between *src1* byte3 and *src2* byte3 is placed in byte3 of *dst*.



The **SUBABS4** instruction aids in motion-estimation algorithms, and other algorithms, that compute the “best match” between two sets of 8-bit quantities.

Execution

```
if (cond)    {
             ABS(ubyte0(src1) – ubyte0(src2)) → ubyte0(dst);
             ABS(ubyte1(src1) – ubyte1(src2)) → ubyte1(dst);
             ABS (ubyte2(src1) – ubyte2(src2)) → ubyte2(dst);
             ABS (ubyte3(src1) – ubyte3(src2)) → ubyte3(dst);
              }
else  nop
```

Pipeline

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

Instruction Type Single-cycle

Delay Slots 0

See Also **ABS, SUB, SUB4**

Example SUBABS4 .L1 A2, A8, A9

	Before instruction			1 cycle after instruction	
A2	37 89 F2 3Ah	55 137 242 58 unsigned	A2	37 89 F2 3Ah	55 137 242 58 unsigned
A8	04 B8 49 75h	4 184 73 117 unsigned	A8	04 B8 49 75h	4 184 73 117 unsigned
A9	XXXX XXXXh		A9	33 2F A9 3Bh	51 47 169 59 unsigned

SWAP2 *Swap Bytes in Each Half-Word (Pseudo-Operation)*

Syntax

SWAP2 (.unit) *src2, dst*
.unit = .L1, .L2 or .S1, .S2

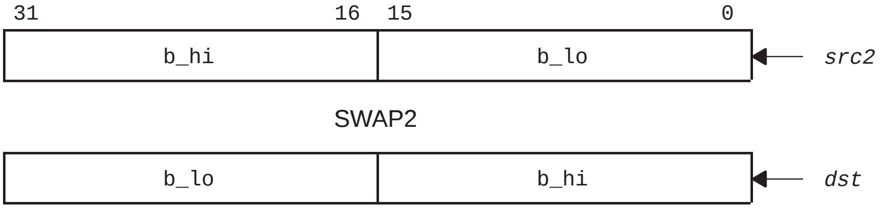
Opcode map field used...	For operand type...	Unit	Opfield
src2 *dst*	s2 s2	.L1, .L2	0011011
src2 *dst*	s2 s2	.S1, .S2	010000

Opcode

See **PACKLH2** instruction.

Description

The **SWAP2** is a pseudo-operation that takes the lower half-word from *src2* and places it in the upper half-word of *dst*, while the upper-half word from *src2* is placed in the lower half-word of *dst*.



The **SWAP2** instruction can be used in conjunction with the **SWAP4** instruction to change the byte ordering (and therefore, the endianess) of 32-bit data.

Execution

```
if (cond)    {
                 msb16(src2) → lsb16(dst);
                 lsb16(src2) → msb16(dst);
              }
else  nop
```

Pipeline

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

Instruction Type Single-cycle

Delay Slots 0

Example 1 SWAP2 .L1 A2,A9

```
      Before instruction                         1 cycle after instruction
A2   │3789 F23Ah   │ 14217 –3526          A2   │3789 F23Ah  │ 14217 –3526

A9   │XXXX XXXXh   │                      A9   │F23A 3789h  │ –3526 14217
```

Example 2 SWAP2 .S2 B2,B12

```
       Before instruction                        1 cycle after instruction
B2    │0124 2451h  │ 292 9297             B2    │0124 2451h  │ 292 9297

B12   │XXXX XXXXh  │                      B12   │2451 0124h  │ 9297 292
```

SWAP4 Swap Bytes in Each Half-Word

Syntax

SWAP4 (.unit) *src2, dst*
.unit = .L1, .L2

Opcode map field used...	For operand type...	Unit	Opfield
src2 *dst*	xu4 u4	.L1, .L2	00001

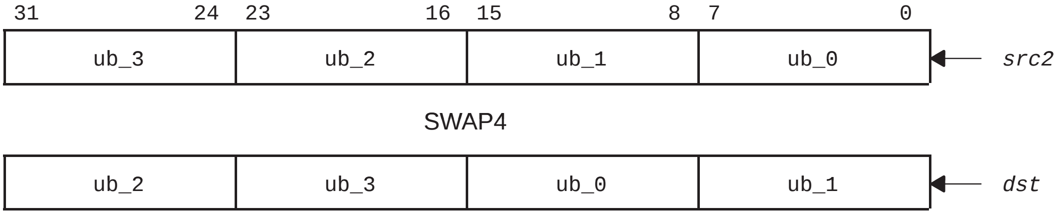
Opcode

31 29	28	27 23	22 18	17 13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src*	*op*	x	0	0	1	1	0	1	0	1	1	0	*s*	*p*
3	1	5	5	5	1	10										1	1

Description

The **SWAP4** instruction exchanges pairs of bytes within each half-word of *src2*, placing the result in *dst*. The values in *src2* are treated as unsigned, packed 8-bit values.

Specifically the upper byte in the upper half-word is placed in the lower byte in the upper halfword, while the lower byte of the upper half-word is placed in the upper byte of the upper half-word. Also the upper byte in the lower half-word is placed in the lower byte of the lower half-word, while the lower byte in the lower half-word is placed in the upper byte of the lower half word.



By itself, this instruction changes the ordering of bytes within half words. This effectively changes the endianess of 16-bit data packed in 32-bit words. The endianess of full 32-bit quantities can be changes by using the **SWAP4** instruction in conjunction with the **SWAP2** instruction.

Execution

```
if (cond)   {
            ubyte0(src2) → ubyte1(dst);
            ubyte1(src2) → ubyte0(dst);
            ubyte2(src2) → ubyte3(dst);
            ubyte3(src2) → ubyte2(dst);
            }
else  nop
```

Pipeline

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

Instruction Type Single-cycle

Delay Slots 0

See Also **SWAP2**

Example

```
SWAP4  .L1   A1,A2
```

	Before instruction		**1 cycle after instruction**
A1	9E 52 6E 30h	A1	9E 52 6E 30h
A2	XXXX XXXXh	A2	52 9E 30 6Eh

UNPKHU4 *Unpack High Unsigned Packed 8-Bit to Unsigned Packed 16-Bit*

Syntax **UNPKHU4** (.unit) *src2, dst*
.unit = .L1, .L2, .S1, .S2

Opcode map field used...	For operand type...	Unit	Opfield
src2 *dst*	xu4 u2	.L1, .L2	00011
src2 *dst*	xu4 u2	.S1, .S2	00011

Opcode

.L unit

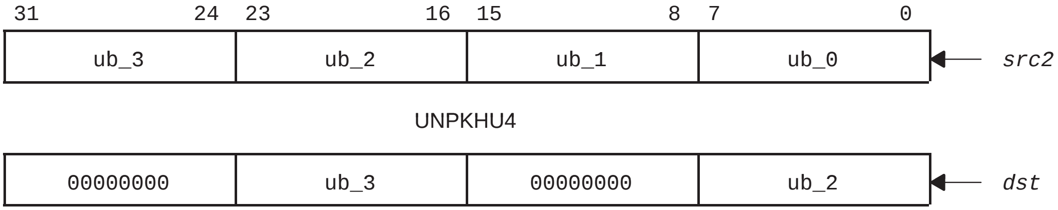
31 29	28	27 23	22 18	17 13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src*	*op*	x	0	0	1	1	0	1	0	1	1	0	*s*	*p*
3	1	5	5	5	1	10										1	1

.S unit

31 29	28	27 23	22 18	17 13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src*	*op*	x	1	1	1	1	0	0	1	0	0	0	*s*	*p*
3	1	5	5	5	1	10										1	1

Description The **UNPKHU4** instruction moves the two most significant bytes of *src2* into the two low bytes of the two half-words of *dst*.

Specifically the upper byte in the upper half-word is placed in the lower byte in the upper halfword, while the lower byte of the upper half-word is placed in the lower byte of the lower half-word. The *src2* bytes are zero-extended when unpacked, filling the two high bytes of the two half-words of *dst* with zeros.



Execution

```
if (cond)    {
             ubyte3(src2) → ubyte2(dst);
                        0 → ubyte3(dst);
             ubyte2(src2) → ubyte0(dst);
                        0 → ubyte1(dst);
             }
else  nop
```

Pipeline

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

Instruction Type Single cycle

Delay Slots 0

See Also **UNPKLU4**

Example 1 UNPKHU4 .L1 A1,A2

	Before instruction		**1 cycle after instruction**
A1	9E 52 6E 30h	A1	9E 52 6E 30h
A2	XXXX XXXXh	A2	00 9E 00 52h

Example 2 UNPKHU4 .L2 B17,B18

	Before instruction		**1 cycle after instruction**
B17	11 05 69 34h	B17	11 05 69 34h
B18	XXXX XXXXh	B18	00 11 00 05h

UNPKLU4 *Unpack Low Unsigned Packed 8-Bit to Unsigned Packed 16-Bit*

Syntax

UNPKLU4 (.unit) *src2, dst*
.unit = .L1, .L2, .S1, .S2

Opcode map field used...	For operand type...	Unit	Opfield
src2 *dst*	xu4 u2	.L1, .L2	00010
src2 *dst*	xu4 u2	.S1, .S2	00010

Opcode

.L unit

31 29	28	27 23	22 18	17 13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src*	*op*	x	0	0	1	1	0	1	0	1	1	0	*s*	*p*
3	1	5	5	5	1	10										1	1

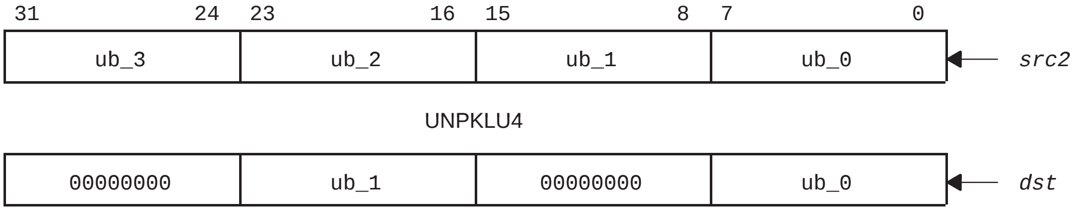
.S unit

31 29	28	27 23	22 18	17 13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src*	*op*	x	1	1	1	1	0	0	1	0	0	0	*s*	*p*
3	1	5	5	5	1	10										1	1

Description

The **UNPKLU4** instruction moves the two least significant bytes of *src2* into the two low bytes of the two half-words of *dst*.

Specifically the upper byte in the lower half-word is placed in the lower byte in the upper halfword, while the lower byte of the lower half-word is kept in the lower byte of the lower half-word. The *src2* bytes are zero-extended when unpacked, filling the two high bytes of the two half-words of *dst* with zeros.



Execution

```
if (cond)    {
                 ubyte0(src2) → ubyte0(dst);
                            0 → ubyte1(dst);
                 ubyte1(src2) → ubyte2(dst);
                            0 → ubyte3(dst);
                 }
else  nop
```

Pipeline

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

Instruction Type Single cycle

Delay Slots 0

See Also **UNPKHU4**

Example 1 UNPKLU4 .L1 A1,A2

	Before instruction		**1 cycle after instruction**
A1	9E 52 6E 30h	A1	9E 52 6E 30h
A2	XXXX XXXXh	A2	00 6E 00 30h

Example 2 UNPKLU4 .L2 B17,B18

	Before instruction		1 cycle after instruction
B17	11 05 69 34h	B17	11 05 69 34h
B18	XXXX XXXXh	B18	00 69 00 34h

XOR

Bitwise XOR

Syntax

XOR (.unit) src1, src2, dst
.unit = .L1, .L2, .S1, .S2, .D1 or .D2

Opcode map field used...	For operand type...	Unit	Opfield
src1 *src2* *dst*	uint xint uint	.L1, .L2	1101111
src1 *src2* *dst*	scst5 xuint uint	.L1, .L2	1101110
src1 *src2* *dst*	uint xunit uint	.S1, .S2	001011
src1 *src2* *dst*	scst5 xuint uint	.S1, .S2	001010
src *src2* *dst*	uint xuint uint	.D1, .D2	1110
src1 *src2* *dst*	scst5 xuint uint	.D1, .D2	1111

Opcode

.L unit

31 29	28	27 23	22 18	17 13	12	11 5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1/cst5*	x	*op*	1	1	0	*s*	*p*
3	1	5	5	5	1	7		3		1	1

.S unit

31 29	28	27 23	22 18	17 13	12	11 6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1/cst5*	x	*op*	1	0	0	0	*s*	*p*
3	1	5	5	5	1	6	4				1	1

.D unit

31 29	28	27 23	22 18	17 13	12	11	10	9 8 7 6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1/cst5*	x	1	0	*op*	1	1	0	0	*s*	*p*
3	1	5	5	5	1	2		4	4				1	1

Description In the **XOR** instruction, 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. This is the same **XOR** instruction that is found on the C62x, but with the added flexibility of being able to perform this operation on the .D unit as well as the .L and .S units.

Execution if (cond) *src1* XOR *src2* → *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

See Also **AND, ANDN, OR**

Example 1 XOR .S1 A3, A4, A5

	Before instruction		**1 cycle after instruction**
A3	0721 325Ah	A3	0721 325Ah
A4	0019 0F12h	A4	0019 0F12h
A5	XXXX XXXXh	A5	0738 3D48h

Example 2 XOR .D2 B1,0dh,B8

	Before instruction		1 cycle after instruction
B1	0000 1023h	B1	0000 1023h
B8	XXXX XXXXh	B8	0000 102Eh

XPND2 *Expand Bits to Packed 16-Bit Masks*

Syntax

XPND2 (.unit) *src2, dst*
.unit = .M1, .M2

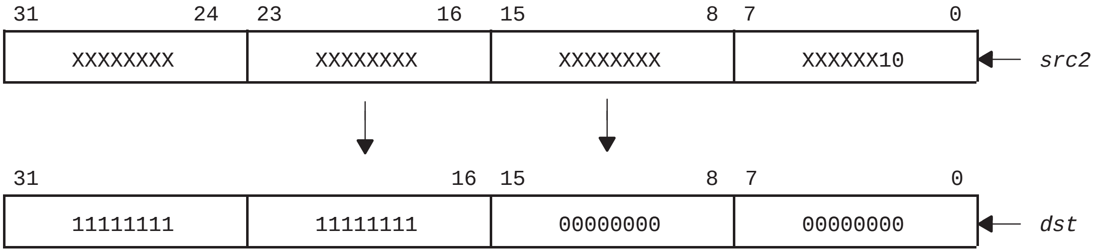
Opcode map field used...	For operand type...	Unit	Opfield
src2 *dst*	xuint uint	.M1, .M2	11001

Opcode

31 29	28	27 23	22 18	17 13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src*	*op*	x	0	0	0	0	1	1	1	1	0	0	*s*	*p*
3	1	5	5	5	1	10										1	1

Description

The **XPND2** instruction reads the two least-significant bits of *src2* and expands them into two half-word masks written to *dst*. Bit 1 of *src2* is replicated and placed in the upper half-word of *dst*. Bit 0 of *src2* is replicated and placed in the lower half-word of *dst*. Bits 2 through 31 of *src2* are ignored.



The **XPND2** instruction is useful, when combined with the output of **CMPGT2** or **CMPEQ2**, for generating a mask that corresponds to the individual half-word positions that were compared. That mask may then be used with **ANDN, AND** or **OR** instructions to perform other operations like compositing. This is an example:

```
CMPGT2    .S1   A3, A4, A5 ; Compare two registers, both upper and
                           ; lower halves.

XPND2     .M1  A5, A2      ; Expand the compare results into two 16-bit
                           ;masks.

NOP

AND       .D1   A2, A7, A8 ; Apply the mask to a value to create result.
```

Because **XPND2** only examines the two least-significant bits of *src2*, it is possible to store a large bit mask in a single 32-bit word and expand it using multiple SHR and XPND2 pairs. This can be useful for expanding a packed 1-bit per pixel bitmap into full 16-bit pixels in imaging applications.

Execution

```
if (cond)   {
                XPND2(src2 & 1) → lsb16(dst);
                XPND2(src2 & 2) → msb16(dst);
                }
else  nop
```

Pipeline

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

Instruction Type Two-cycle

Delay Slots 1

See Also **CMPEQ2, CMPGT2, XPND4**

Example 1 XPND2 .M1 A1,A2

```
      Before instruction                         2 cycles after instruction
A1  | B174 6CA1h |  2 LSBs are 01          A1  | B174 6CA1h |  2 LSBs are 01

A2  | XXXX XXXXh |                         A2  | 0000 FFFFh |
```

Example 2 XPND2 .M2 B1,B2

	Before instruction			2 cycles after instruction	
B1	0000 0003h	2 LSBs are 11	B1	0000 0003h	2 LSBs are 11
B2	XXXX XXXXh		B2	FFFF FFFFh	

XPND4 *Expand Bits to Packed 8-Bit Masks*

Syntax

XPND4 (.unit) *src2, dst*
.unit = .M1, .M2

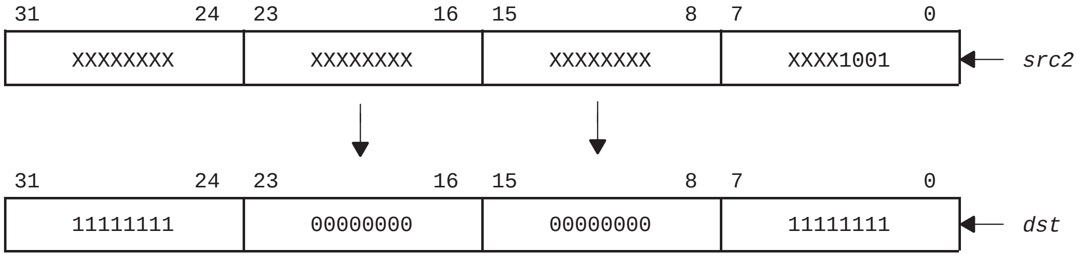
Opcode map field used...	For operand type...	Unit	Opfield
src2 *dst*	xuint uint	.M1, .M2	11000

Opcode

31 29	28	27 23	22 18	17 13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src*	*op*	x	0	0	0	0	1	1	1	1	0	0	*s*	*p*
3	1	5	5	5	1	10										1	1

Description

The **XPND4** instruction reads the four least-significant bits of *src2* and expands them into four-byte masks written to *dst*. Bit 0 of *src2* is replicated and placed in the least significant byte of *dst*. Bit 1 of *src2* is replicated and placed in second least significant byte of *dst*. Bit 2 of *src2* is replicated and placed in second most significant byte of *dst*. Bit 3 of *src2* is replicated and placed in most significant byte of *dst*. Bits 4 through 31 of *src2* are ignored.



The **XPND4** instruction is useful, when combined with the output of **CMPGT4** or **CMPEQ4**, for generating a mask that corresponds to the individual byte positions that were compared. That mask may then be used with **ANDN, AND** or **OR** instructions to perform other operations like compositing. This is an example:

```
CMPEQ4   .S1  A3, A4, A5 ; Compare two 32-bit registers all four bytes.
XPND4    .M1  A5, A2     ; Expand the compare results into four 8-bit
                         ; masks.
NOP
AND      .D1  A2, A7, A8 ; Apply the mask to a value to create result.
```

Because **XPND4** only examines the four least-significant bits of *src2*, it is possible to store a large bit mask in a single 32-bit word and expand it using multiple **SHR** and **XPND4** pairs. This can be useful for expanding a packed, 1-bit per pixel bitmap into full 8-bit pixels in imaging applications.

Execution

```
if (cond)   {
              XPND4(src2 & 1) → byte0(dst);
              XPND4(src2 & 2) → byte1(dst);
              XPND4(src2 & 4) →byte2(dst);
              XPND4(src2 & 8) →byte3(dst);
            }
else  nop
```

Pipeline

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

Instruction Type Two-cycle

Delay Slots 1

See Also **CMPEQ4, CMPGTU4, XPND2**

Example 1

XPND4 .M1 A1,A2

	Before instruction			2 cycles after instruction	
A1	B174 6CA4h	4 LSBs are 0100	A1	B174 6CA4h	4 LSBs are 0100
A2	XXXX XXXXh		A2	00 FF 00 00h	

Example 2

XPND4 .M2 B1,B2

	Before instruction			2 cycles after instruction	
B1	0000 000Ah	4 LSBs are 1010	B1	00 00 00 0Ah	4 LSBs are 1010
B12	XXXX XXXXh		B2	FF 00 FF 00h	

Chapter 6

TMS320C62x/C64x Pipeline

The TMS320C62x™/TMS320C64x™ DSP pipeline provides flexibility to simplify programming and improve performance. These two factors provide this flexibility:

1) Control of the pipeline is simplified by eliminating pipeline interlocks.
2) 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™/C64x™ generally requires fewer execution phases than the C67x™ because the C62x/C64x 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 *TMS320C6000 Programmer's Guide* (SPRU198).

Topic **Page**

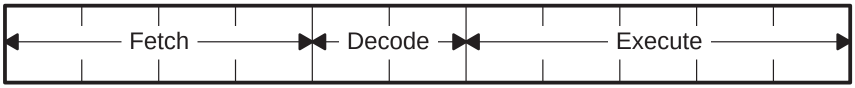
6.1 Pipeline Operation Overview

The pipeline phases are divided into three stages:

- ❑ Fetch
- ❑ Decode
- ❑ Execute

All instructions in the C62x/C64x 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/C64x pipeline are shown in Figure 6–1.

Figure 6–1. Fixed-Point Pipeline Stages



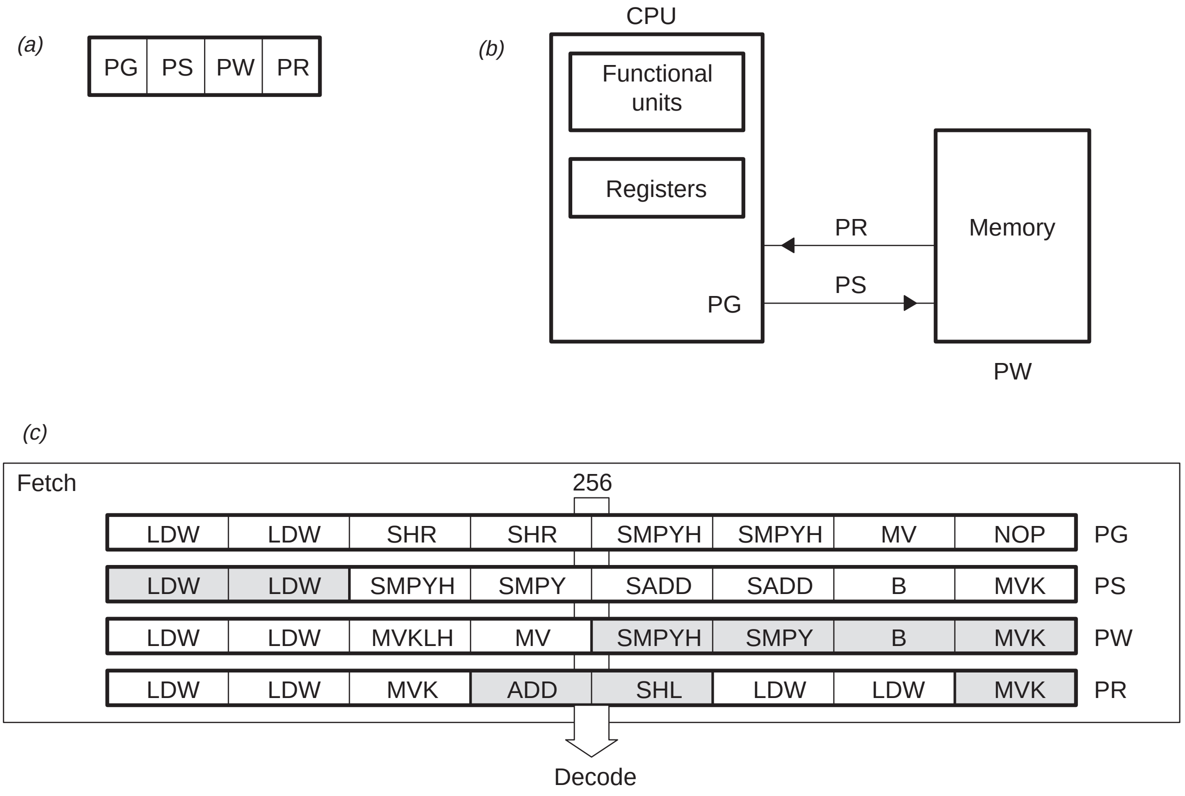
6.1.1 Fetch

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 C62x/C64x 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) 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 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 instructions.

Figure 6–2. Fetch Phases of the Pipeline



LDW	LDW	SHR	SHR	SMPYH	SMPYH	MV	NOP	PG
LDW	LDW	SMPYH	SMPY	SADD	SADD	B	MVK	PS
LDW	LDW	MVKLH	MV	SMPYH	SMPY	B	MVK	PW
LDW	LDW	MVK	ADD	SHL	LDW	LDW	MVK	PR

6.1.2 Decode

The decode phases of the pipeline are:

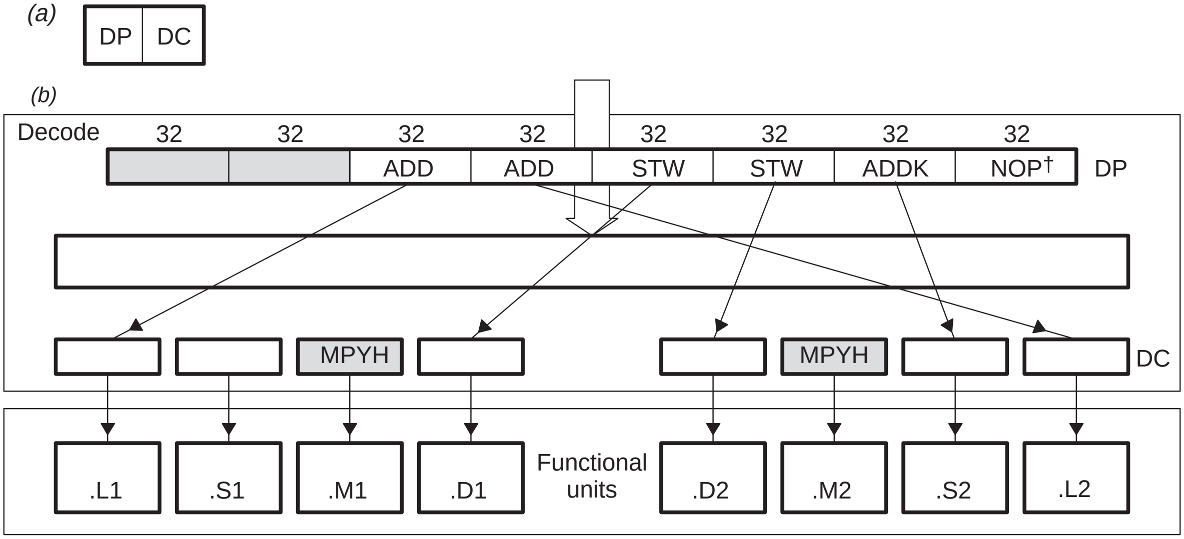
- ❑ **DP:** Instruction dispatch
- ❑ **DC:** Instruction decode

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

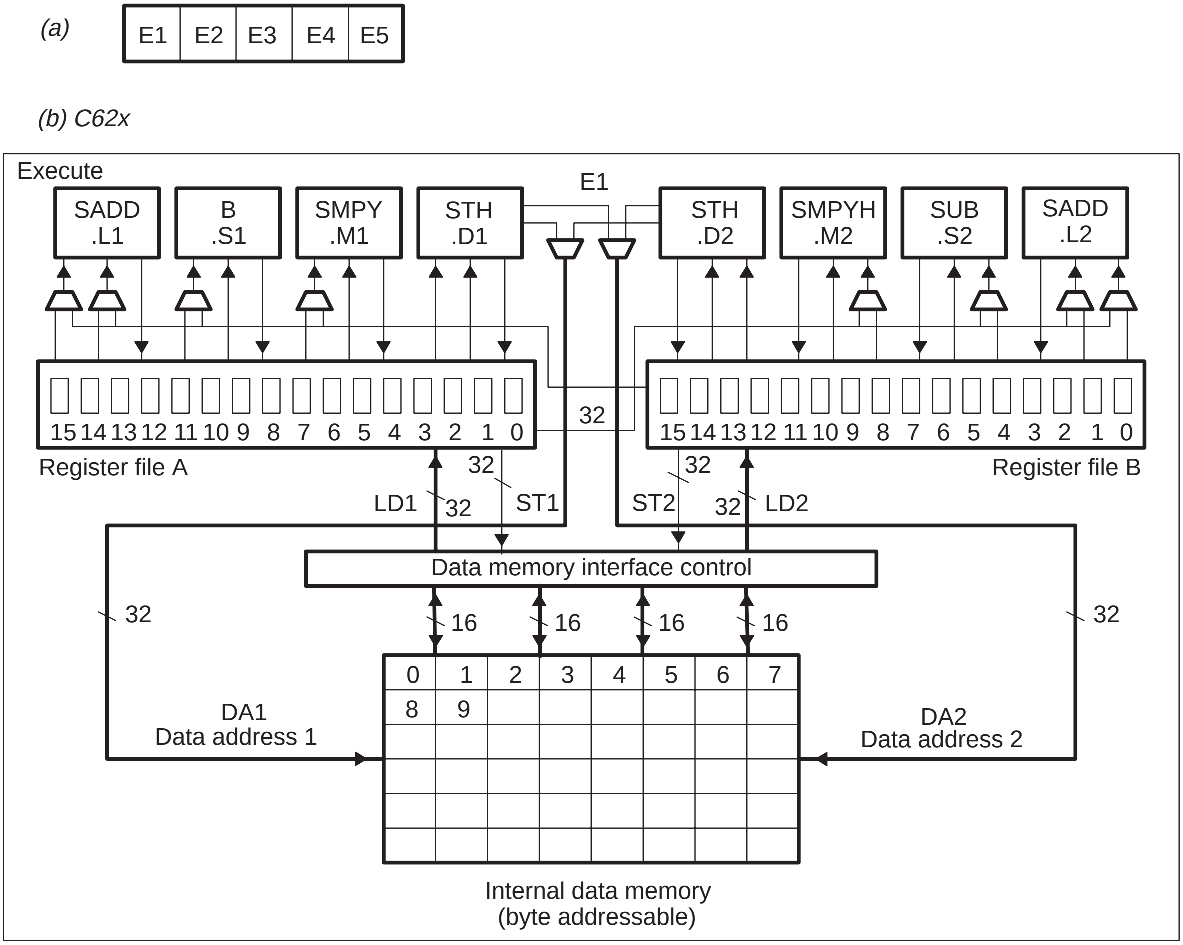


† NOP is not dispatched to a functional unit.

6.1.3 Execute

The execute portion of the 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 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) and (c) show the portion of the functional block diagram in which execution occurs on the C62x and C64x, respectively.

Figure 6–4. Execute Phases of the Pipeline and Functional Block Diagram of the TMS320C62x/C64x

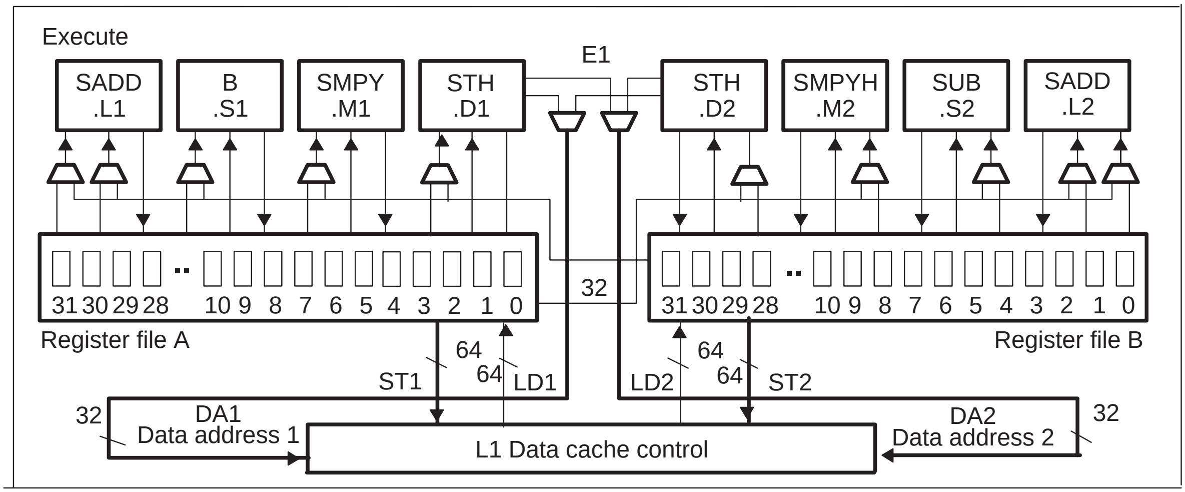


(a)

(b) C62x

Figure 6–4. Execute Phases of the Pipeline and Functional Block Diagram of the TMS320C62x/C64x (Continued)

(c) C64x



6.1.4 Summary of Pipeline Operation

Figure 6–5 shows all the phases in each stage of the C62x/C64x pipeline in sequential order, from left to right.

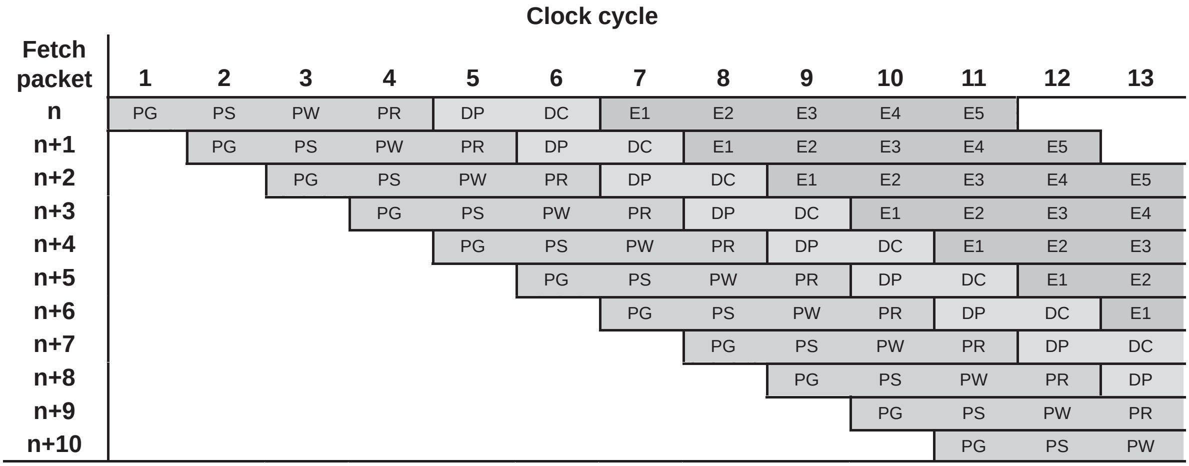
Figure 6–5. Fixed-Point Pipeline Phases

Fetch				Decode		Execute				
PG	PS	PW	PR	DP	DC	E1	E2	E3	E4	E5

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 6.3, *Performance Considerations*, on page 6-21 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
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 6–1 summarizes the pipeline phases and what happens in each.

Table 6–1. 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

Table 6–1. Operations Occurring During Fixed-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 control status register (CSR) if saturation occurs.† For single 16 x 16 multiply instructions, results are written to a register file.† For C64x multiply unit non-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.† For C64x multiply extensions, results are written to a register file.§	Multiply Extensions
	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.

‡ Multiply unit, non-multiply instructions are **AVG2, AVG4, BITC4, BITR, DEAL, ROT, SHFL, SSHVL,** and **SSHVR.**

§ Multiply extensions include **MPY2, MPY4, DOTPx2, DOTPU4, MPYHIx, MPYLIx,** and **MVD.**

Figure 6–7 shows a C62x functional block diagram laid out vertically by stages of the pipeline.

Figure 6–7. Functional Block Diagram of TMS320C62x Based on Pipeline Phases

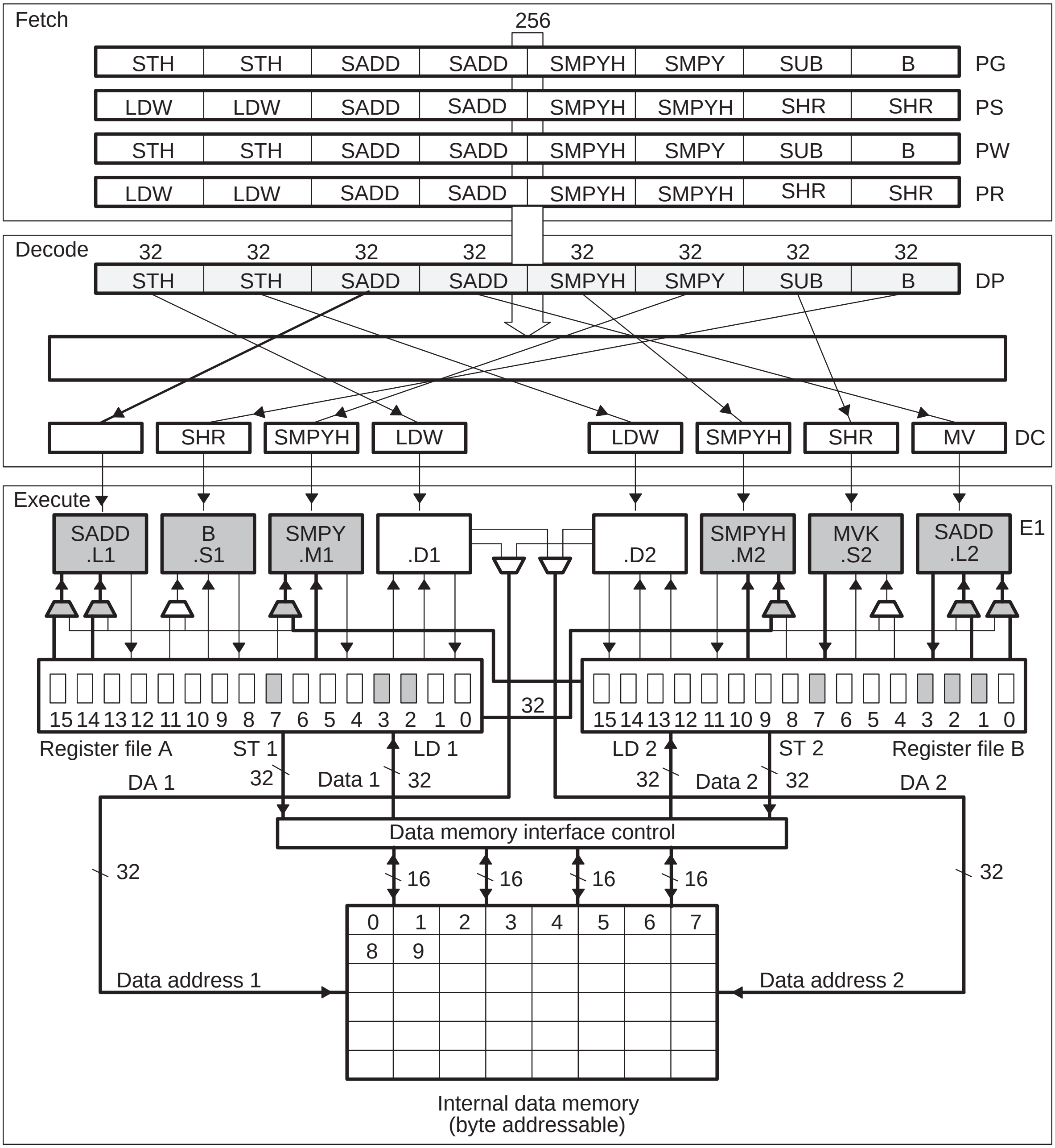
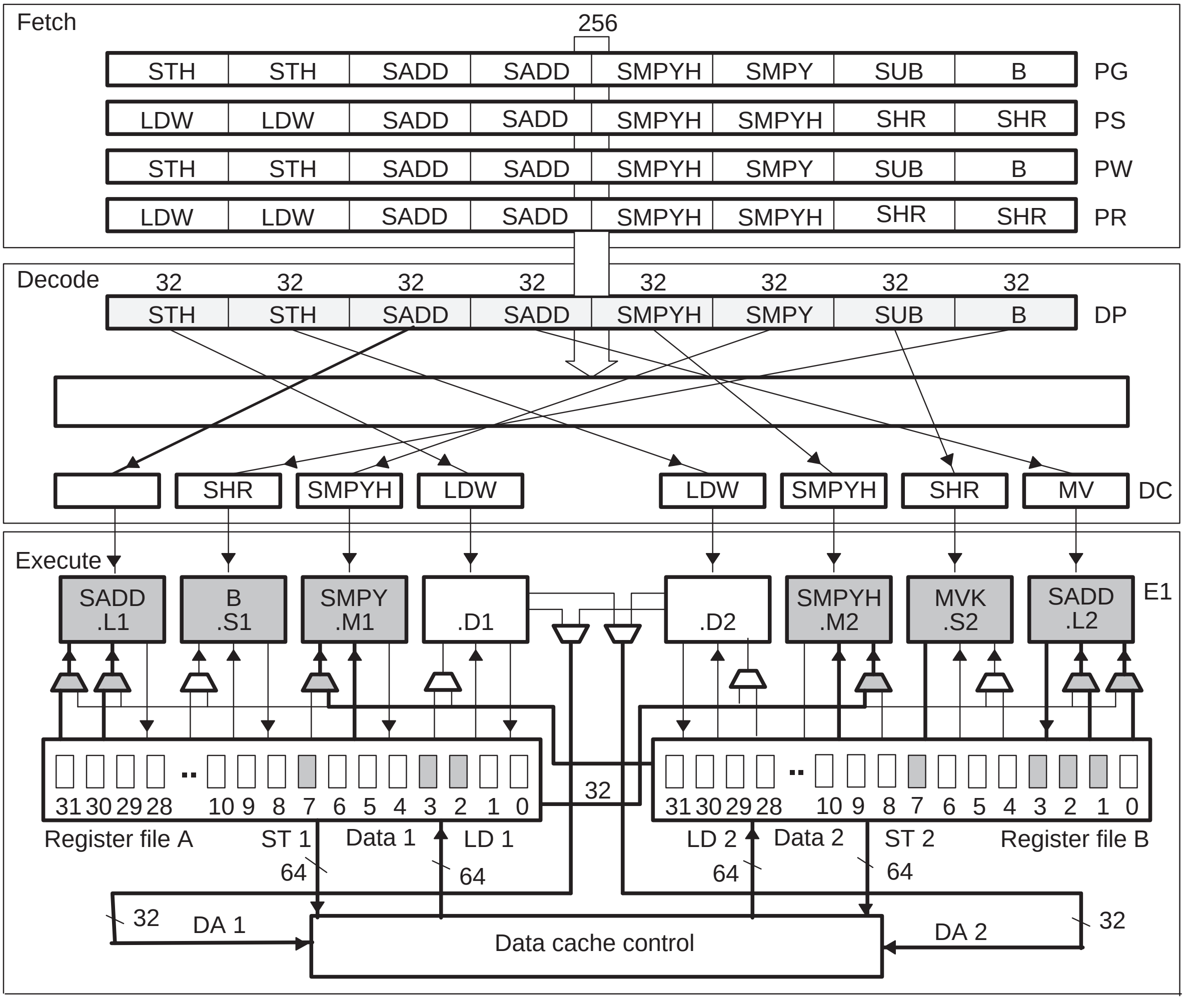


Figure 6–8 shows a C64x functional block diagram laid out vertically by stages of the pipeline. It is identical to Figure 6–7 except for the additional registers and functional unit hardware that are not being used in the code example being shown.

Figure 6–8. Functional Block Diagram of TMS320C64x 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/C64x. Figure 6–7 and Figure 6–8 show 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.

Example 6–1. Execute Packet in Figure 6–7 and Figure 6–8

```
        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 6–7 and Figure 6–8, 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 and Figure 6–8. 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/C64x 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 6.2, *Pipeline Execution of Instruction Types*.

6.2 Pipeline Execution of Instruction Types

The pipeline operation of the C62x/C64x instructions can be categorized into seven instruction types. Six 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 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 X 16 Single Multiply/ C64x .M Unit Non-Multiply**	**Store**	**C64x Multiply Extensions**	**Load**	**Branch**
Execution phases	E1	Compute result and write to register	Read operands and start computations	Compute address	Reads operands and start computations	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				Write results to register	Send data back to CPU	
	E5					Write data into register	
Delay slots		0	1	0†	3	4†	5‡

† See sections 6.2.3 and 6.2.5 for more information on execution and delay slots for stores and loads.

‡ See section 6.2.6 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.

6.2.1 Single-Cycle Instructions

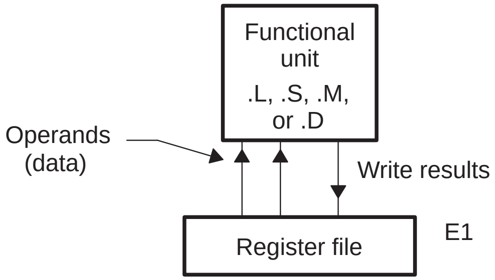
Single-cycle instructions complete execution during the E1 phase of the pipeline. Figure 6–9 shows the fetch, decode, and execute phases of the pipeline that single-cycle instructions use.

Figure 6–9. Single-Cycle Instruction Phases

PG	PS	PW	PR	DP	DC	E1

Figure 6–10 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 6–10. Single-Cycle Execution Block Diagram



6.2.2 Two-Cycle Instructions and C64x Non-multiply .M Unit Operations

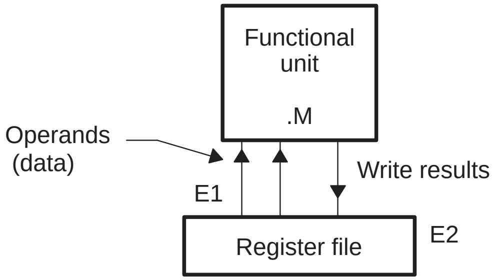
Multiply instructions use both the E1 and E2 phases of the pipeline to complete their operations. Figure 6–11 shows the pipeline phases two-cycle instructions use.

Figure 6–11. Instruction Phases

PG	PS	PW	PR	DP	DC	E1	E2	1 delay slot

Figure 6–12 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. This execution block diagram also applies to the other C64x non-multiply .M unit operations.

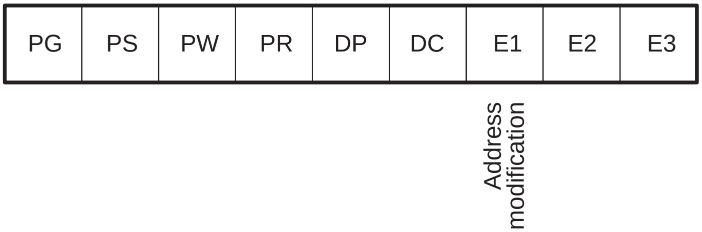
Figure 6–12. Single 16 x 16 Multiply Execution Block Diagram



6.2.3 Store Instructions

Store instructions require phases E1 through E3 to complete their operations. Figure 6–13 shows the pipeline phases the store instructions use.

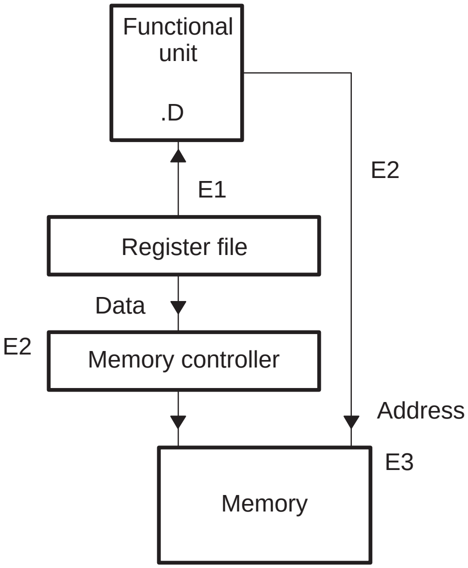
Figure 6–13. Store Instruction Phases



PG	PS	PW	PR	DP	DC	E1	E2	E3

Figure 6–14 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 6–14. 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 6.2.5.

6.2.4 Extended Multiply Instructions

The extended multiply instructions use phases E1 – E4 to complete their operations. Figure 6–15 shows the pipeline phases used by the extended multiply instructions.

Figure 6–15. Extended Multiply Instruction Phases

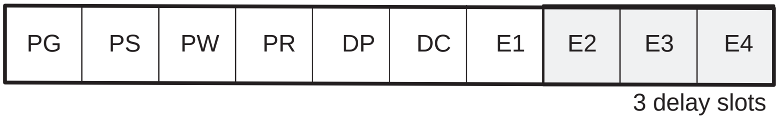
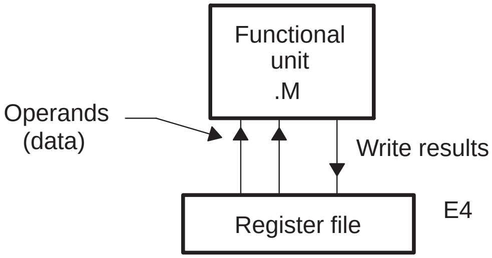


Figure 6–16 shows the operations occurring in the pipeline for the multiply extensions. In the E1 phase, the operands are read and the multiplies begin. In the E4 phase, the multiplies finish, and the results are written to the destination register. Extended multiply instructions have three delay slots.

Figure 6–16. Multiply Extensions Execution Block Diagram



6.2.5 Load Instructions

Data loads require all five of the pipeline execute phases to complete their operations. Figure 6–17 shows the pipeline phases the load instructions use.

Figure 6–17. Load Instruction Phases

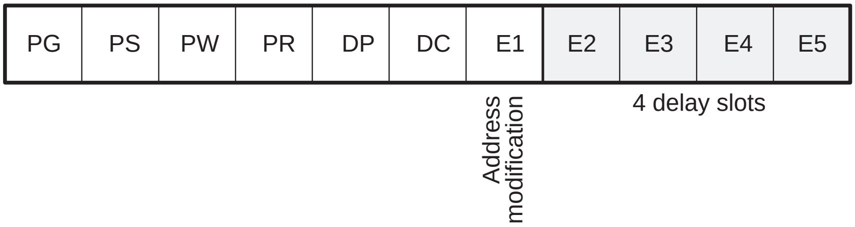
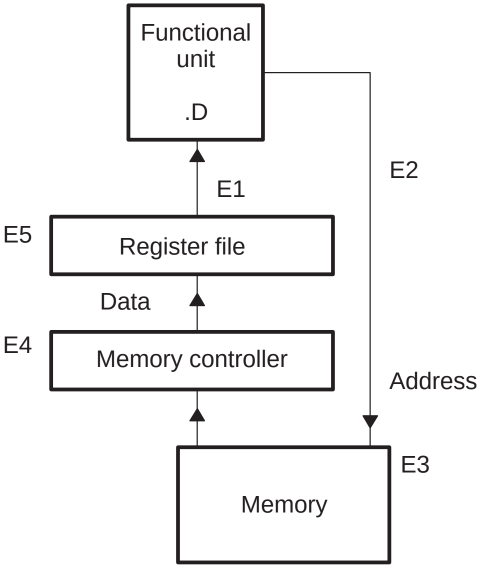


Figure 6–18 shows the operations occurring in the pipeline phases for a load. In the E1 phase, the data address pointer is modified in its register. In the E2 phase, the data address is sent to data memory. In the E3 phase, a memory read at that address is performed.

Figure 6–18. 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.

6.2.6 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 6–19 shows the pipeline phases used by the branch instruction and branch target code. The delay slots are shaded.

Figure 6–19. Branch Instruction Phases

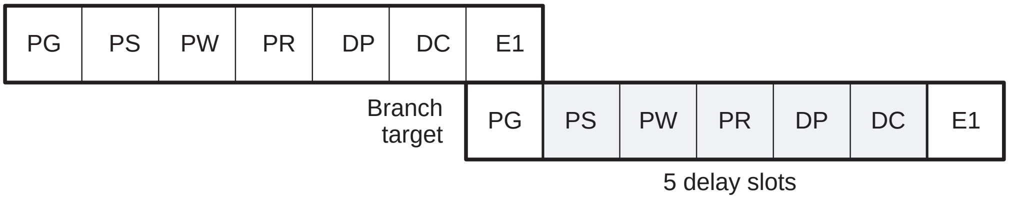
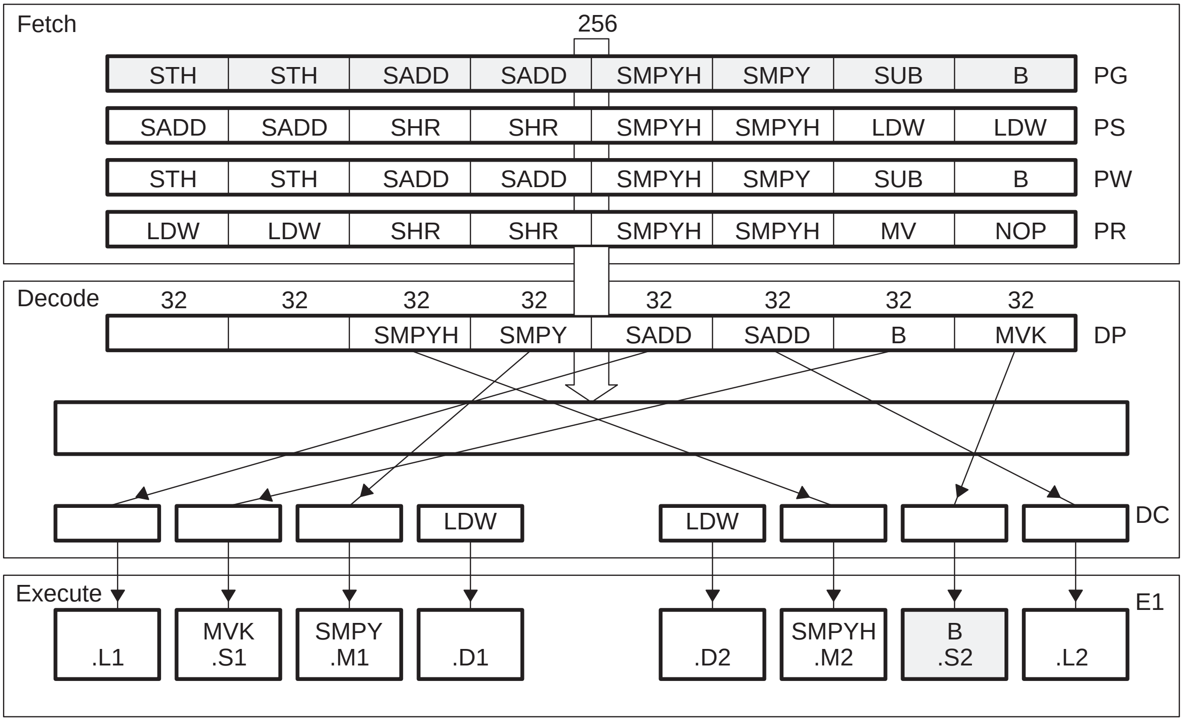


Figure 6–20 shows a branch execution block diagram. If a branch is in the E1 phase of the pipeline (in the .S2 unit in the figure), its branch target is in the fetch packet that is in PG during that same cycle (shaded in the figure). Because the branch target has to wait until it reaches the E1 phase to begin execution, the branch takes five delay slots before the branch target code executes.

Figure 6–20. Branch Execution Block Diagram


Fetch
256
STH STH SADD SADD SMPYH SMPY SUB B
PG
SADD SADD SHR SHR SMPYH SMPYH LDW LDW
PS
STH STH SADD SADD SMPYH SMPY SUB B
PW
LDW LDW SHR SHR SMPYH SMPYH MV NOP
PR
Decode
32 32 32 32 32 32 32 32
SMPYH SMPY SADD SADD B MVK
DP
LDW
LDW
DC
Execute
E1
.L1
MVK .S1
SMPY .M1
.D1
.D2
SMPYH .M2
B .S2
.L2

6.3 Performance Considerations

The C62x/C64x 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 6.3.2 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.

6.3.1 Pipeline Operation With Multiple Execute Packets in a Fetch Packet

Again referring to Figure 6–6 on page 6-8, pipeline operation is shown with eight instructions in every fetch packet. Figure 6–21, however, shows the pipeline operation with a fetch packet that contains multiple execute packets. Code for Figure 6–21 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 6–21. Pipeline Operation: Fetch Packets With Different Numbers of Execute Packets

Fetch packet (FP)	Execute packet (EP)	Clock cycle												
		1	2	3	4	5	6	7	8	9	10	11	12	13
n	k	PG	PS	PW	PR	DP	DC	E1	E2	E3	E4	E5		
n	k+1						DP	DC	E1	E2	E3	E4	E5	
n	k+2							DP	DC	E1	E2	E3	E4	E5
n+1	k+3		PG	PS	PW	PR			DP	DC	E1	E2	E3	E4
n+2	k+4			PG	PS	PW	Pipeline		PR	DP	DC	E1	E2	E3
n+3	k+5				PG	PS	stall		PW	PR	DP	DC	E1	E2
n+4	k+6					PG			PS	PW	PR	DP	DC	E1
n+5	k+7								PG	PS	PW	PR	DP	DC
n+6	k+8									PG	PS	PW	PR	DP

In Figure 6–21, 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.

6.3.2 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 6–22 shows how a multicycle **NOP** can drive the execution of other instructions in the same execute packet. Figure 6–22(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 6–22(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 6–22. Multicycle NOP in an Execute Packet

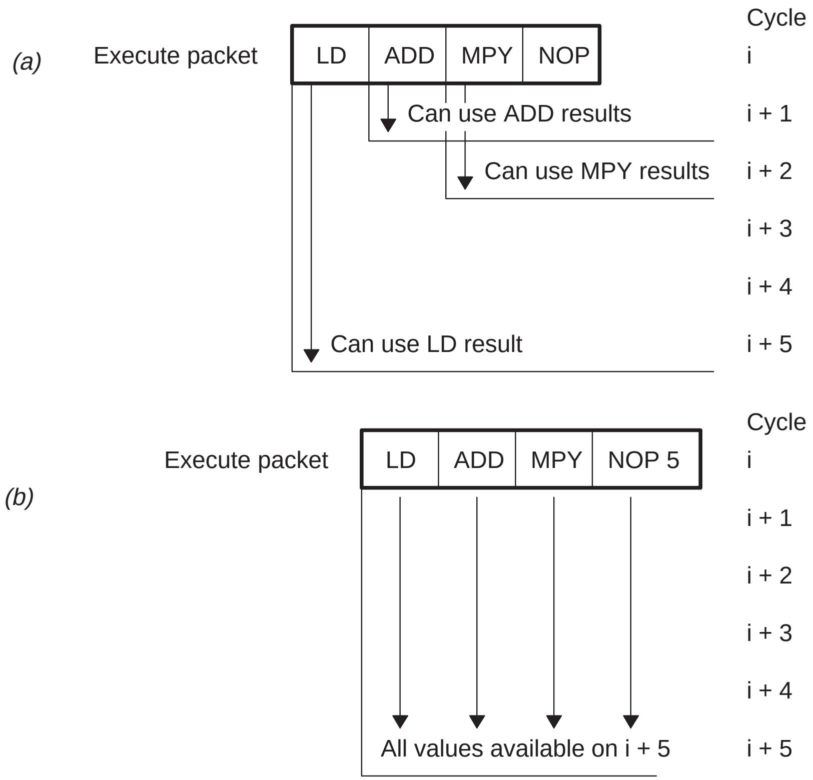
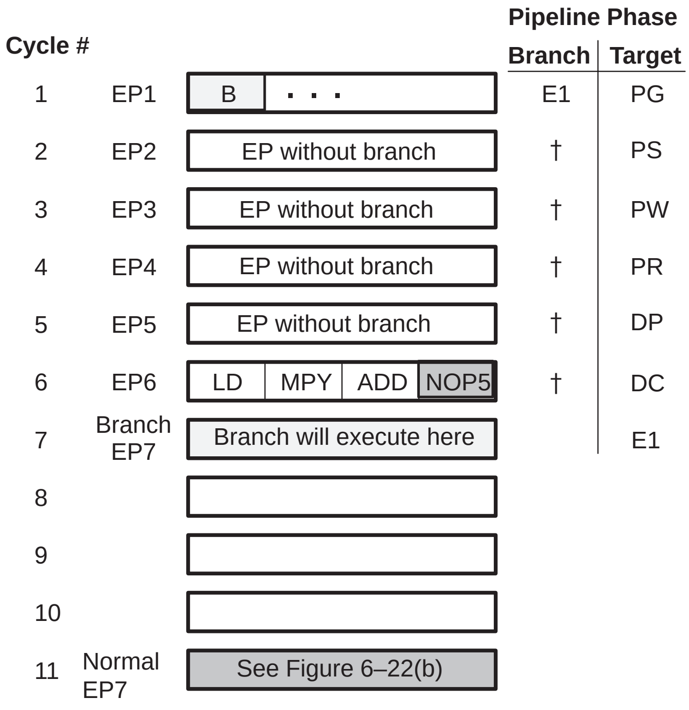


Figure 6–23 shows how a multicycle **NOP** can be affected by a branch. If the delay slots of a branch finish while a multicycle **NOP** is still dispatching **NOP**s into the pipeline, the branch overrides the multicycle **NOP** and the branch target begins execution five delay slots after the branch was issued.

Figure 6–23. Branching and Multicycle ***NOP****s*



Cycle #			Pipeline Phase: Branch	Pipeline Phase: Target
1	EP1	B . . .	E1	PG
2	EP2	EP without branch	†	PS
3	EP3	EP without branch	†	PW
4	EP4	EP without branch	†	PR
5	EP5	EP without branch	†	DP
6	EP6	LD | MPY | ADD | NOP5	†	DC
7	Branch EP7	Branch will execute here		E1
8				
9				
10				
11	Normal EP7	See Figure 6–22(b)		

† Delay slots of the branch

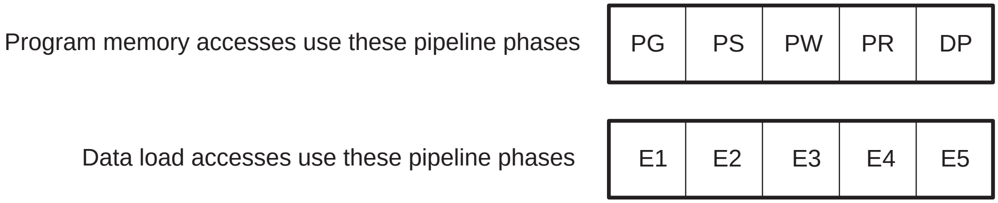
In one case, execute packet 1 (EP1) does not have a branch. The **NOP** 5 in EP6 will force the CPU to wait until cycle 11 to execute EP7.

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.

6.3.3 Memory Considerations

The C62x/C64x 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/C64x to access memory at a high speed. These phases are shown in Figure 6–24.

Figure 6–24. Pipeline Phases Used During Memory Accesses


Program memory accesses use these pipeline phases
PG
PS
PW
PR
DP
Data load accesses use these pipeline phases
E1
E2
E3
E4
E5

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/C64x and perform the same types of operations (listed in Table 6–3) to accommodate those memories. Table 6–3 shows the operation of program fetches pipeline versus the operation of a data load.

Table 6–3. 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

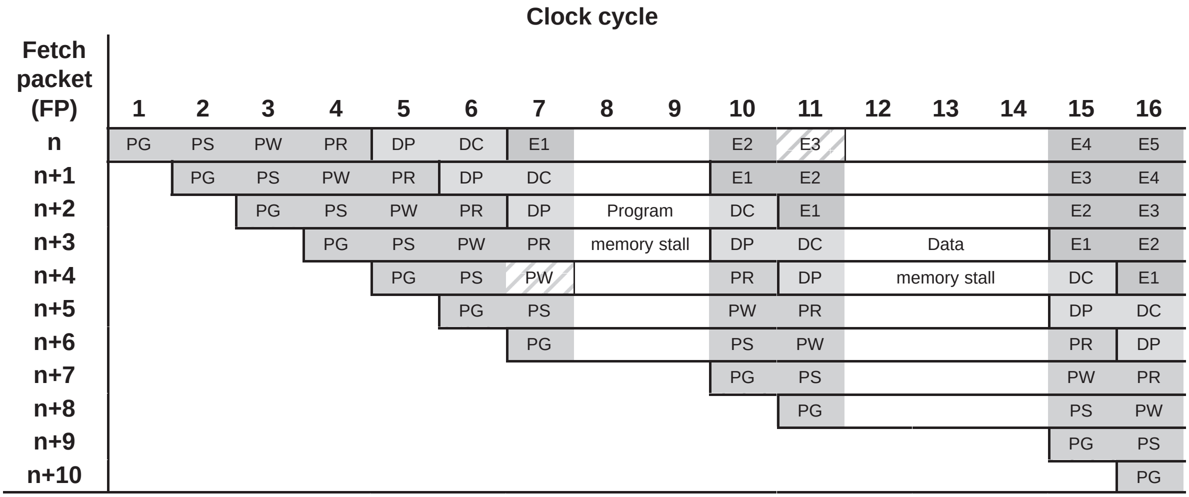
Depending on the type of memory and the time required to complete an access, the pipeline may stall to ensure proper coordination of data and instructions. This is discussed in section 6.3.3.1, *Memory Stalls.*

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 6.3.3.2, *Memory Bank Hits*.

6.3.3.1 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 6–25 illustrates this point.

Figure 6–25. Program and Data Memory Stalls



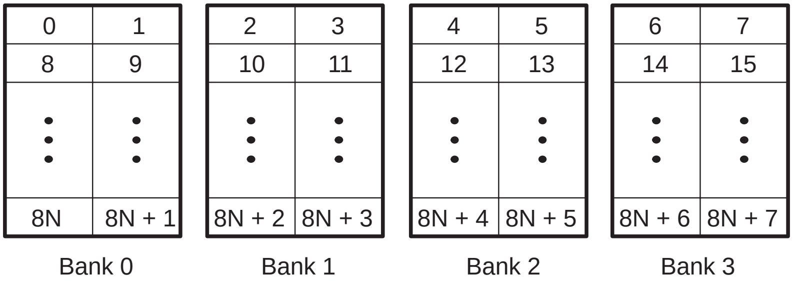
Clock cycle

Fetch packet (FP)	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
n	PG	PS	PW	PR	DP	DC	E1			E2	E3				E4	E5
n+1		PG	PS	PW	PR	DP	DC			E1	E2				E3	E4
n+2			PG	PS	PW	PR	DP	Program		DC	E1				E2	E3
n+3				PG	PS	PW	PR	memory stall		DP	DC	Data			E1	E2
n+4					PG	PS	PW			PR	DP	memory stall			DC	E1
n+5						PG	PS			PW	PR				DP	DC
n+6							PG			PS	PW				PR	DP
n+7										PG	PS				PW	PR
n+8											PG				PS	PW
n+9															PG	PS
n+10																PG

6.3.3.2 Memory Bank Hits

Most C62x devices use an interleaved memory bank scheme, as shown in Figure 6–26. The C6211, C6711, and C64x devices use a two-level cache memory scheme. Thus the example below does not pertain to these devices. 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 6–26. 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 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 6–4 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/C64x/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
```

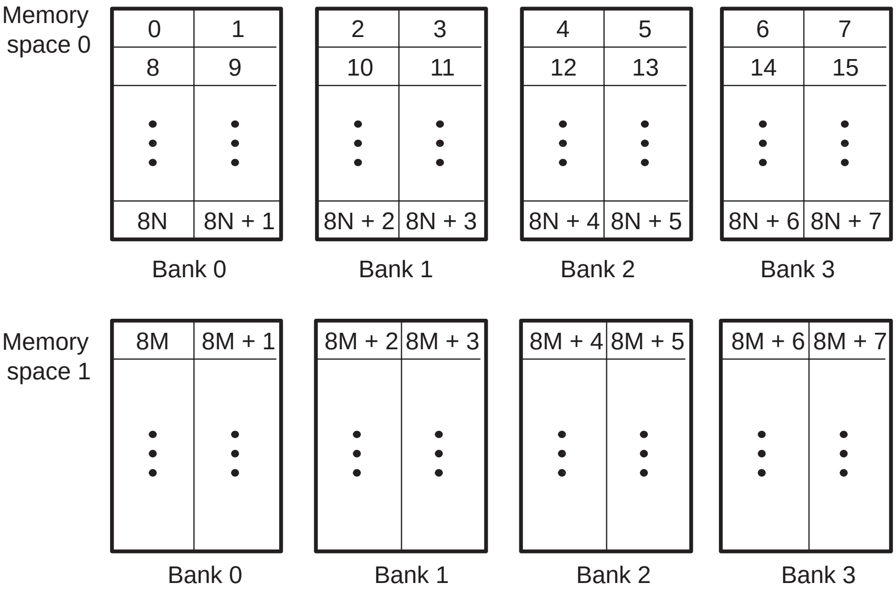
Table 6–4. Loads in Pipeline From Example 6–2

	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

† Stall due to memory bank hit

For devices that have more than one memory space (see Figure 6–27), 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 6–27. 4-Bank Interleaved Memory With Two Memory Spaces



The internal memory of the C62x/C64x family varies from device to device. See the *TMS320C6000 Peripherals Reference Guide* to determine the memory spaces in your particular device.

Chapter 7

TMS320C67x Pipeline

The TMS320C67x™ DSP 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 TMS320C62x™ DSP 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 *TMS320C6000 Programmer's Guide* (SPRU198).

Topic **Page**

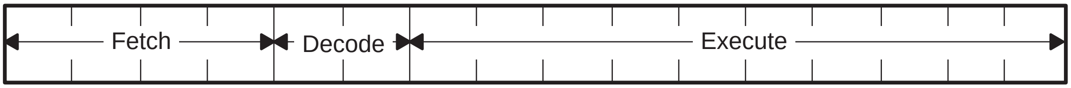
7.1 Pipeline Operation Overview

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 7–1.

Figure 7–1. Floating-Point Pipeline Stages



7.1.1 Fetch

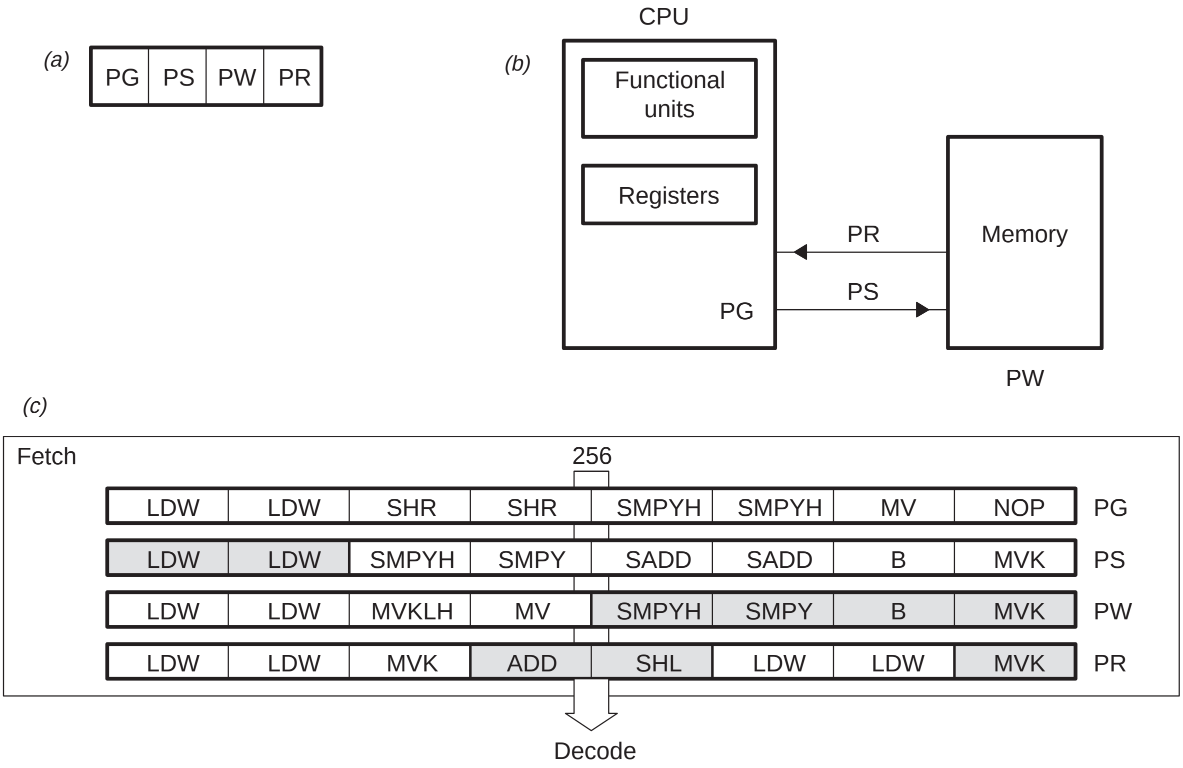
The fetch phases of the pipeline are:

- ❑ **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 7–2(a) shows the fetch phases in sequential order from left to right. Figure 7–2(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 7–2(c) shows fetch packets flowing through the phases of the fetch stage of the pipeline. In Figure 7–2(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 7–2. Fetch Phases of the Pipeline


(a)
PG
PS
PW
PR
(b)
CPU
Functional units
Registers
PG
PR
PS
Memory
PW
(c)
Fetch
256
LDW LDW SHR SHR SMPYH SMPYH MV NOP PG
LDW LDW SMPYH SMPY SADD SADD B MVK PS
LDW LDW MVKLH MV SMPYH SMPY B MVK PW
LDW LDW MVK ADD SHL LDW LDW MVK PR
Decode

7.1.2 Decode

The decode phases of the pipeline are:

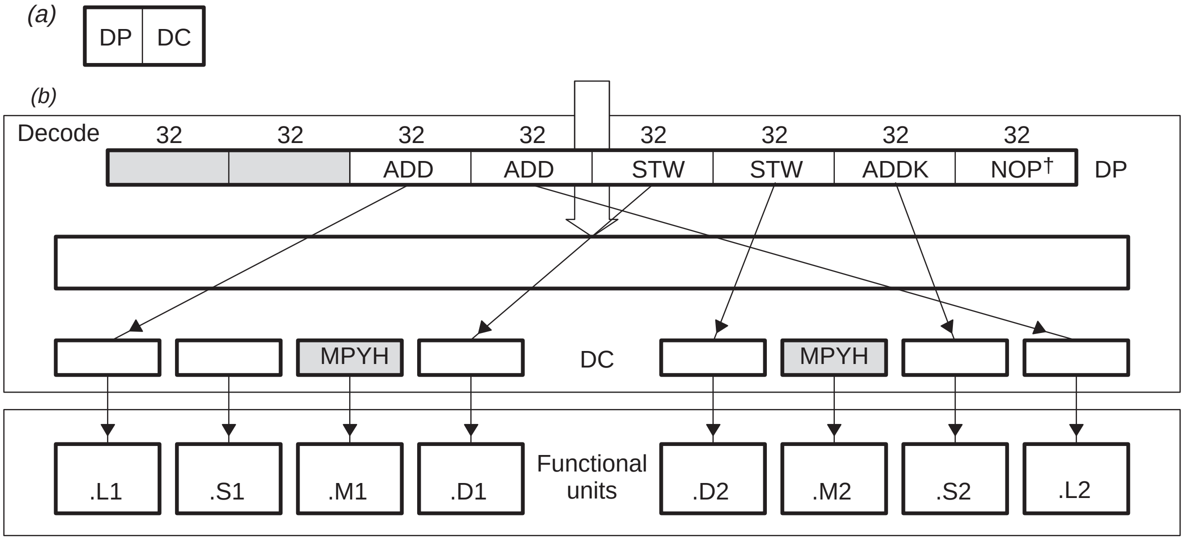
- ❑ **DP:** Instruction dispatch
- ❑ **DC:** Instruction decode

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 7–3(a) shows the decode phases in sequential order from left to right. Figure 7–3(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 7–3. Decode Phases of the Pipeline



† NOP is not dispatched to a functional unit.

7.1.3 Execute

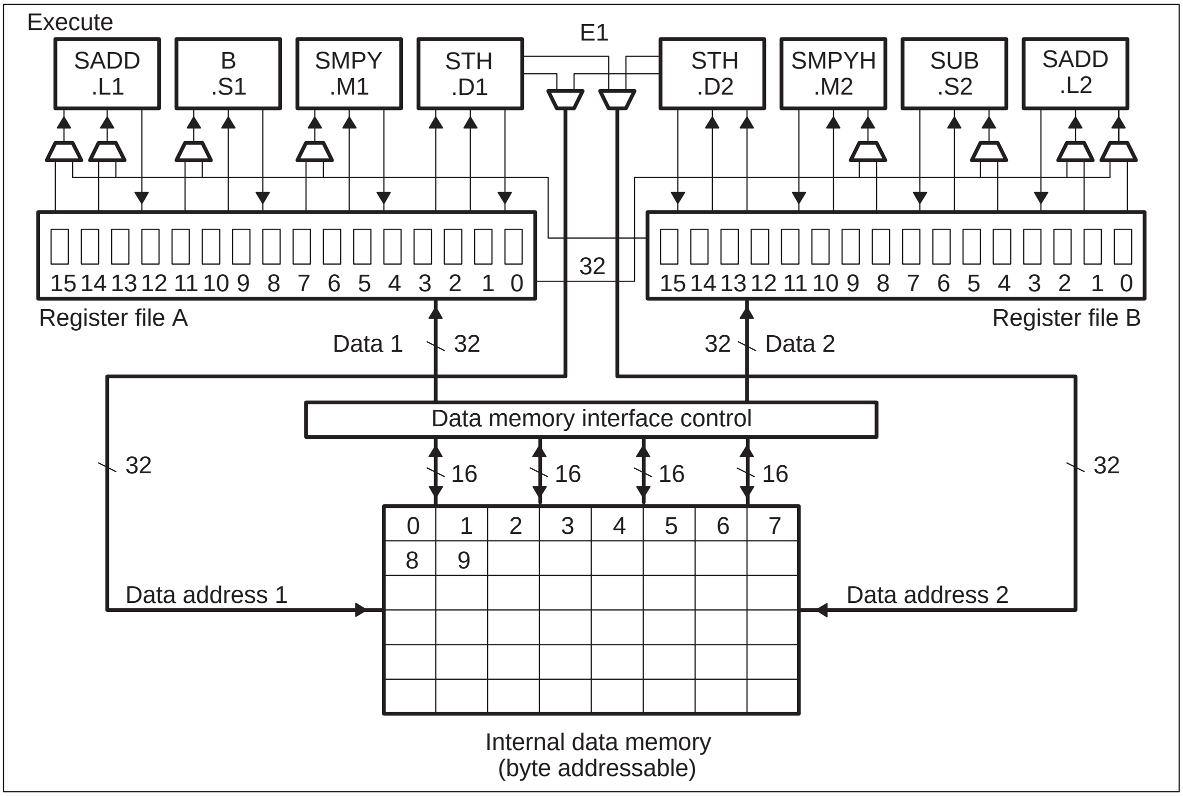
The execute portion of the floating-point pipeline is subdivided into ten phases (E1–E10), as compared to the fixed-point pipeline's five phases. Different types of instructions require different numbers of these phases to complete their execution. These phases of the pipeline play an important role in your understanding the device state at CPU cycle boundaries. The execution of different types of instructions in the pipeline is described in section 7.2, *Pipeline Execution of Instruction Types*. Figure 7–4(a) shows the execute phases of the pipeline in sequential order from left to right. Figure 7–4(b) shows the portion of the functional block diagram in which execution occurs.

Figure 7–4. Execute Phases of the Pipeline and Functional Block Diagram of the TMS320C67x

(a)

E1	E2	E3	E4	E5	E6	E7	E8	E9	E10

(b)



7.1.4 Summary of Pipeline Operation

Figure 7–5 shows all the phases in each stage of the C67x pipeline in sequential order, from left to right.

Figure 7–5. Floating-Point Pipeline Phases

Fetch				Decode		Execute									
PG	PS	PW	PR	DP	DC	E1	E2	E3	E4	E5	E6	E7	E8	E9	E10

Figure 7–6 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 7–6. 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 7.4, *Performance Considerations*, on page 7-52 for additional detail on code flowing through the pipeline.

Figure 7–6. 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**	**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 7–1 summarizes the pipeline phases and what happens in each.

Table 7–1. 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 7–1. 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.

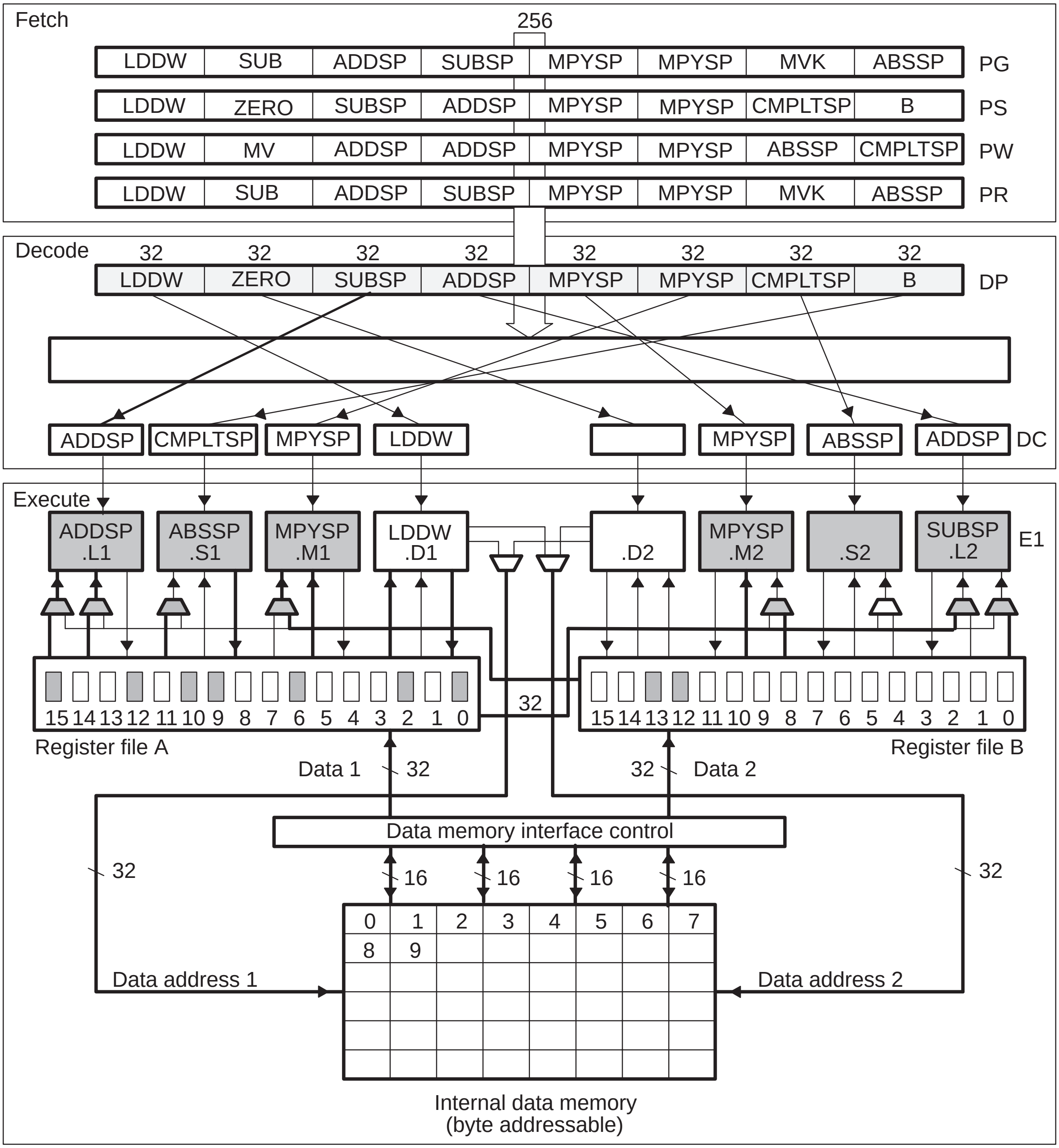
Table 7–1. Operations Occurring During Floating-Point Pipeline Phases (Continued)

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 7–7 shows a C67x functional block diagram laid out vertically by stages of the pipeline.

Figure 7–7. 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 7–7 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 7–1.

In the DC phase portion of Figure 7–7, 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 7–7. 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 7.2, *Pipeline Execution of Instruction Types*.

Example 7–1. Execute Packet in Figure 7–7

```
        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
```

7.2 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 7–2 (**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 7–2. 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 7.3.7 (page 7-40) and 7.3.8 (page 7-42) for more information on execution and delay slots for stores and loads.
‡ See section 7.3.9 (page 7-44) 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 7–2. 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 7–2. 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 constraints 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 constraint 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 constraint 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 constraint 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 constraint on cycle i + 3.

A multiply (16 × 16-bit) instruction cannot be scheduled on the same functional unit on cycle i + 2 due to a write constraint 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 constraint 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 constraint on cycle i + 1.

A 4-cycle instruction cannot be scheduled on the same functional unit on cycle i + 1 due to a write constraint 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 constraint 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 constraint 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 constraint 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 constraint 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 constraint 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 constraint 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.

7.3 Functional Unit Constraints

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.

7.3.1 .S-Unit Constraints

Table 7–3 shows the instruction constraints for single-cycle instructions executing on the .S unit.

Table 7–3. Single-Cycle .S-Unit Instruction Constraints

	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:	▨	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 7–4 shows the instruction constraints for DP compare instructions executing on the .S unit.

Table 7–4. DP Compare .S-Unit Instruction Constraints

	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 constraint
Xrw Next instruction cannot enter E1 during cycle–read/decode/write constraint
† The branch on register instruction is the only branch instruction that reads a general-purpose register

Table 7–5 shows the instruction constraints for 2-cycle DP instructions executing on the .S unit.

Table 7–5. 2-Cycle DP .S-Unit Instruction Constraints

	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 constraint

Table 7–6 shows the instruction constraints for branch instructions executing on the .S unit.

Table 7–6. Branch .S-Unit Instruction Constraints

	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:
▨ 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

7.3.2 .M-Unit Constraints

Table 7–7 shows the instruction constraints for 16 X 16 multiply instructions executing on the .M unit.

Table 7–7. 16 X 16 Multiply .M-Unit Instruction Constraints

	Instruction Execution		
Cycle	1	2	3
16 X 16 multiply	R	W	
Instruction Type	**Subsequent Same-Unit Instruction Executable**		
16 X 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:		
	▨	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 7–8 shows the instruction constraints for 4-cycle instructions executing on the .M unit.

Table 7–8. 4-Cycle .M-Unit Instruction Constraints

	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:		
	▨	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 constraint

Table 7–9 shows the instruction constraints for **MPYI** instructions executing on the .M unit.

Table 7–9. MPYI .M-Unit Instruction Constraints

	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:	▨	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 constraint
	Xw	Next instruction cannot enter E1 during cycle–write constraint
	Xu	Next instruction cannot enter E1 during cycle–other resource conflict

Table 7–10 shows the instruction constraints for **MPYID** instructions executing on the .M unit.

Table 7–10. MPYID .M-Unit Instruction Constraints

	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: ▨ 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 constraint
Xw Next instruction cannot enter E1 during cycle–write constraint
Xu Next instruction cannot enter E1 during cycle–other resource conflict

Table 7–11 shows the instruction constraints for **MPYDP** instructions executing on the .M unit.

Table 7–11. MPYDP .M-Unit Instruction Constraints

	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:	▨	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 constraint
	Xw	Next instruction cannot enter E1 during cycle–write constraint
	Xu	Next instruction cannot enter E1 during cycle–other resource conflict

7.3.3 .L-Unit Constraints

Table 7–12 shows the instruction constraints for single-cycle instructions executing on the .L unit.

Table 7–12. Single-Cycle .L-Unit Instruction Constraints

	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: ▨ 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 7–13 shows the instruction constraints for 4-cycle instructions executing on the .L unit.

Table 7–13. 4-Cycle .L-Unit Instruction Constraints

	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: ▨ 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 constraint

Table 7–14 shows the instruction constraints for **INTDP** instructions executing on the .L unit.

Table 7–14. INTDP .L-Unit Instruction Constraints

	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:

▨	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 constraint

Table 7–15 shows the instruction constraints for **ADDDP/SUBDP** instructions executing on the .L unit.

Table 7–15. ADDDP/SUBDP .L-Unit Instruction Constraints

	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 × 16 multiply	▨	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 constraint
	Xw	Next instruction cannot enter E1 during cycle–write constraint
	Xrw	Next instruction cannot enter E1 during cycle–read/decode/write constraint

7.3.4 D-Unit Instruction Constraints

Table 7–16 shows the instruction constraints for load instructions executing on the .D unit.

Table 7–16. Load .D-Unit Instruction Constraints

	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: ▨ 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 7–17 shows the instruction constraints for store instructions executing on the .D unit.

Table 7–17. Store .D-Unit Instruction Constraints

	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: ▨ 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 7–18 shows the instruction constraints for single-cycle instructions executing on the .D unit.

Table 7–18. Single-Cycle .D-Unit Instruction Constraints

	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 7–19 shows the instruction constraints for **LDDW** instructions executing on the .D unit.

Table 7–19. LDDW Instruction With Long Write Instruction Constraints

	Instruction Execution					
Cycle	1	2	3	4	5	6
LDDW	RW				W	

Instruction Type	Subsequent Same-Unit Instruction Executable					
Instruction with long result	▨	✓	✓	✓	Xw	✓

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 constraint

7.3.5 Single-Cycle Instructions

Single-cycle instructions complete execution during the E1 phase of the pipeline (see Table 7–20). Figure 7–8 shows the fetch, decode, and execute phases of the pipeline that single-cycle instructions use. Figure 7–9 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.

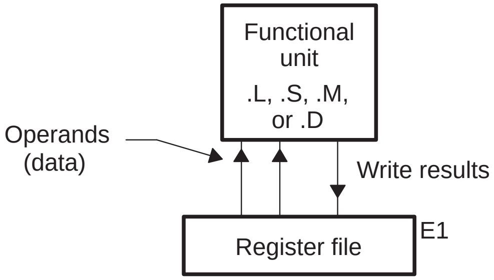
Table 7–20. Single-Cycle Execution

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

Figure 7–8. Single-Cycle Instruction Phases

PG	PS	PW	PR	DP	DC	E1

Figure 7–9. Single-Cycle Execution Block Diagram



7.3.6 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 7–21). Figure 7–10 shows the pipeline phases the multiply instructions use. Figure 7–11 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 7–21. 16 × 16-Bit Multiply Execution

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

Figure 7–10. Multiply Instruction Phases

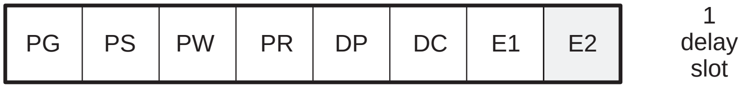
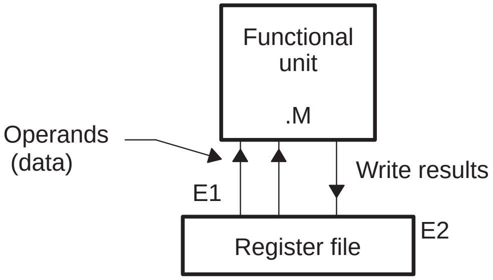


Figure 7–11. Multiply Execution Block Diagram



7.3.7 Store Instructions

Store instructions require phases E1 through E3 to complete their operations (see Table 7–22). Figure 7–12 shows the pipeline phases the store instructions use. Figure 7–13 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 7–22. Store Execution

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

Figure 7–12. Store Instruction Phases

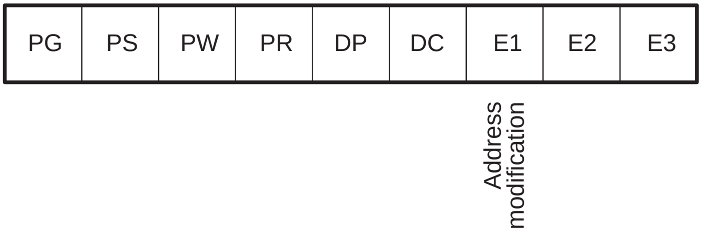
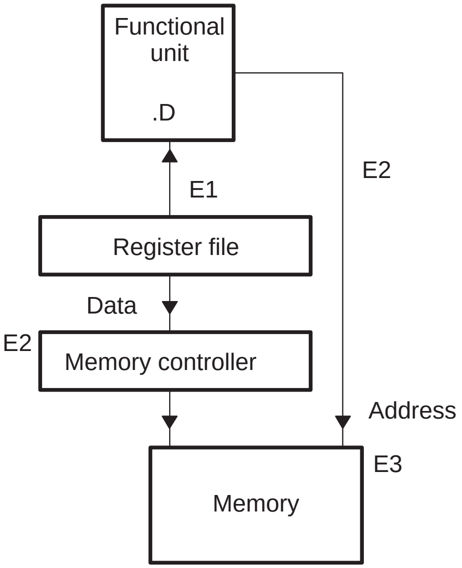


Figure 7–13. Store Execution Block Diagram



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 7.3.8.

7.3.8 Load Instructions

Data loads require five of the pipeline execute phases to complete their operations (see Table 7–23). Figure 7–14 shows the pipeline phases the load instructions use.

Table 7–23. Load Execution

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

Figure 7–14. Load Instruction Phases

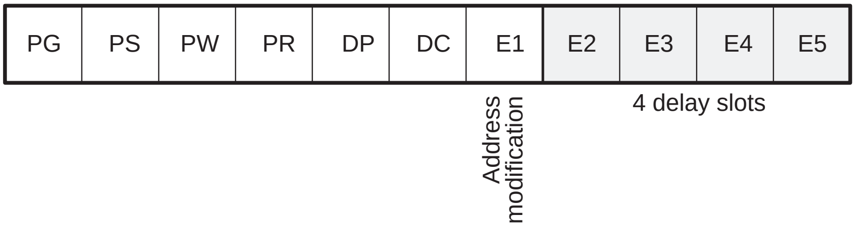
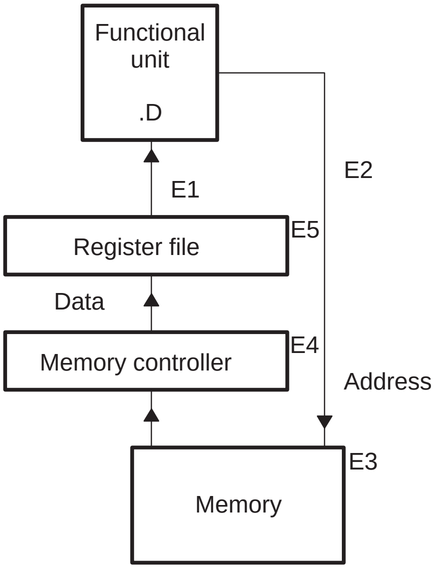


Figure 7–15 shows the operations occurring in the pipeline phases for a load. In the E1 phase, the data address pointer is modified in its register. In the E2 phase, the data address is sent to data memory. In the E3 phase, a memory read at that address is performed.

Figure 7–15. 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.

7.3.9 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 7–24). Figure 7–16 shows the pipeline phases used by the branch instruction and branch target code. The delay slots are shaded.

Table 7–24. Branch Execution

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

Figure 7–16. Branch Instruction Phases

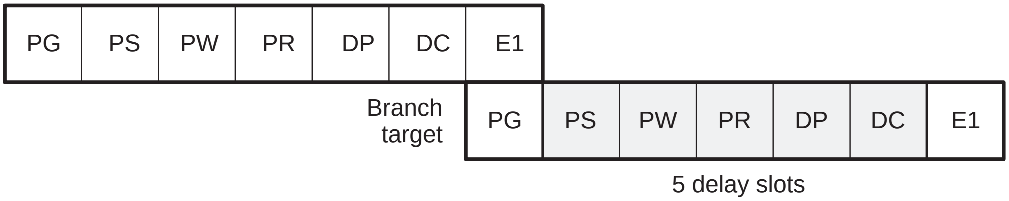
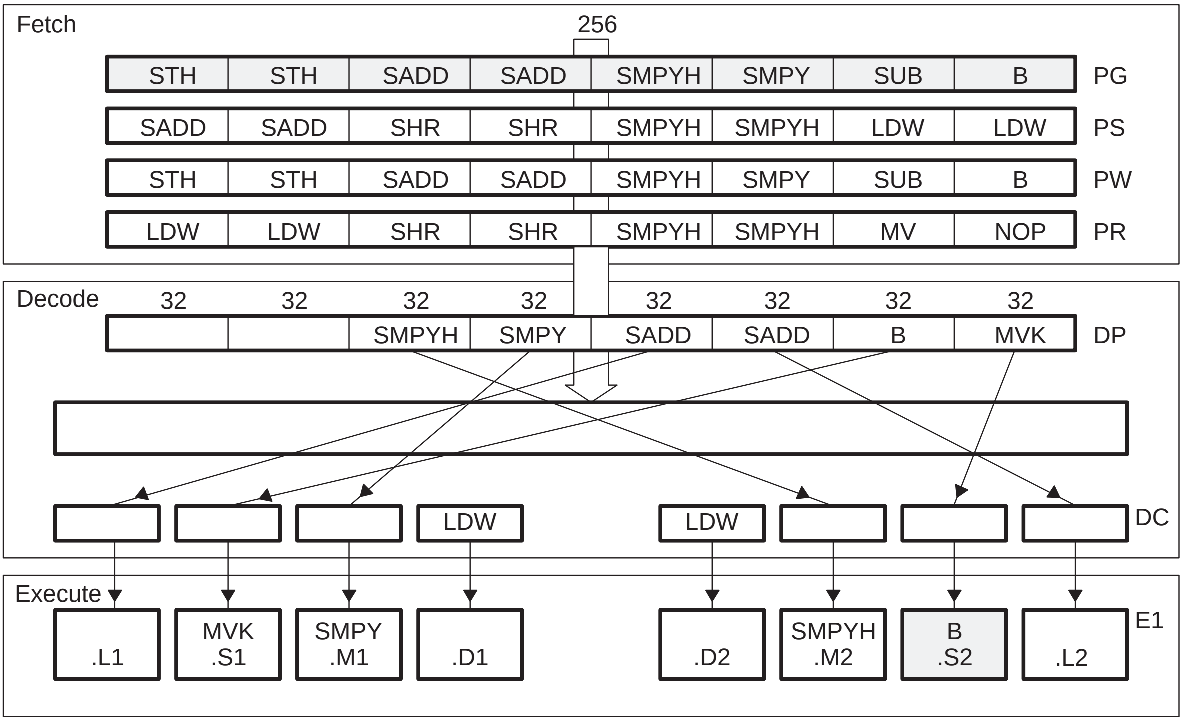


Figure 7–17 shows a branch execution block diagram. If a branch is in the E1 phase of the pipeline (in the .S2 unit in the figure), its branch target is in the fetch packet that is in PG during that same cycle (shaded in the figure). Because the branch target has to wait until it reaches the E1 phase to begin execution, the branch takes five delay slots before the branch target code executes.

Figure 7–17. Branch Execution Block Diagram


Fetch
256
STH STH SADD SADD SMPYH SMPY SUB B
PG
SADD SADD SHR SHR SMPYH SMPYH LDW LDW
PS
STH STH SADD SADD SMPYH SMPY SUB B
PW
LDW LDW SHR SHR SMPYH SMPYH MV NOP
PR
Decode
32 32 32 32 32 32 32 32
SMPYH SMPY SADD SADD B MVK
DP
LDW
LDW
DC
Execute
.L1
MVK .S1
SMPY .M1
.D1
.D2
SMPYH .M2
B .S2
.L2
E1

7.3.10 2-Cycle DP Instructions

Two-cycle DP instructions use the E1 and E2 phases of the pipeline to complete their operations (see Table 7–25). The following instructions are two-cycle DP instructions:

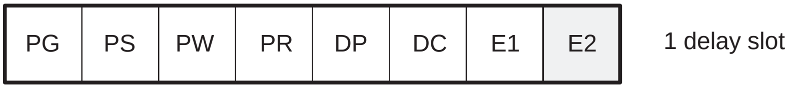
- ABSDP
- RCPDP
- RSQDP
- SPDP

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 7–18 shows the pipeline phases the 2-cycle DP instructions use.

Table 7–25. 2-Cycle DP Execution

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

Figure 7–18. 2-Cycle DP Instruction Phases



7.3.11 4-Cycle Instructions

Four-cycle instructions use the E1 through E4 phases of the pipeline to complete their operations (see Table 7–26). The following instructions are 4-cycle instructions:

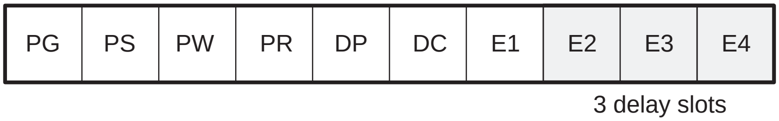
- ADDSP
- DPINT
- DPSP
- DPTRUNC
- INTSP
- MPYSP
- SPINT
- SPTRUNC
- SUBSP

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 7–19 shows the pipeline phases the 4-cycle instructions use.

Table 7–26. 4-Cycle Execution

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

Figure 7–19. 4-Cycle Instruction Phases



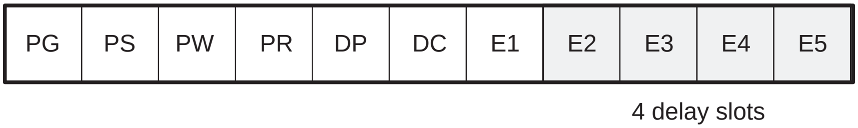
7.3.12 INTDP Instruction

The INTDP instruction uses the E1 through E5 phases of the pipeline to complete its operations (see Table 7–27). *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 7–20 shows the pipeline phases the INTDP instructions use.

Table 7–27. INTDP Execution

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

Figure 7–20. INTDP Instruction Phases



PG	PS	PW	PR	DP	DC	E1	E2	E3	E4	E5

4 delay slots

7.3.13 DP Compare Instructions

The DP compare instructions use the E1 and E2 phases of the pipeline to complete their operations (see Table 7–28). 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:

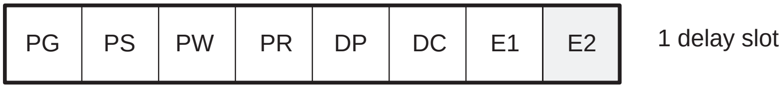
❑ CMPEQDP
❑ CMPLTDP
❑ CMPGTDP

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 7–21 shows the pipeline phases the DP compare instructions use.

Table 7–28. DP Compare Execution

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

Figure 7–21. DP Compare Instruction Phases



PG	PS	PW	PR	DP	DC	E1	E2

1 delay slot

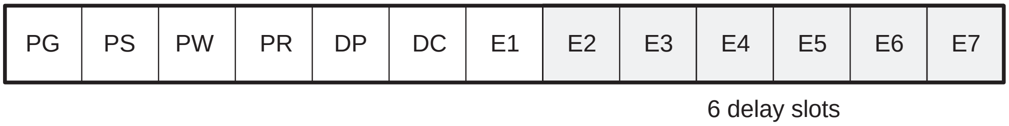
7.3.14 ADDDP/SUBDP Instructions

The ADDDP/SUBDP instructions use the E1 through E7 phases of the pipeline to complete their operations (see Table 7–29). 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 7–22 shows the pipeline phases the ADDDP/SUBDP instructions use.

Table 7–29. 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 7–22. ADDDP/SUBDP Instruction Phases



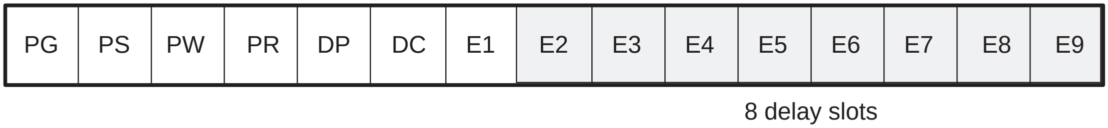
7.3.15 MPYI Instructions

The MPYI instruction uses the E1 through E9 phases of the pipeline to complete its operations (see Table 7–30). 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 7–23 shows the pipeline phases the MPYI instructions use.

Table 7–30. 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 7–23. MPYI Instruction Phases



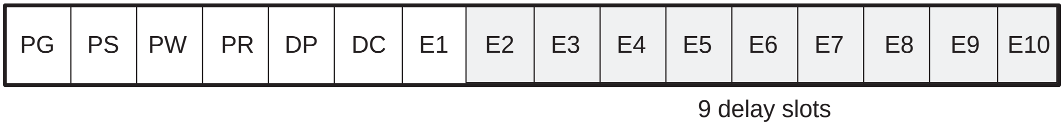
7.3.16 MPYID Instructions

The MPYID instruction uses the E1 through E10 phases of the pipeline to complete its operations (see Table 7–31). 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 7–24 shows the pipeline phases the MPYID instructions use.

Table 7–31. 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 7–24. MPYID Instruction Phases



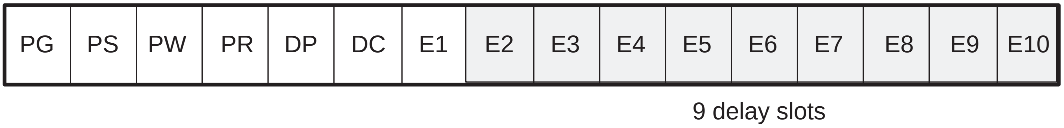
7.3.17 MPYDP Instructions

The MPYDP instruction uses the E1 through E10 phases of the pipeline to complete its operations (see Table 7–32). 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 7–25 shows the pipeline phases the MPYDP instructions use.

Table 7–32. 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 7–25. MPYDP Instruction Phases



7.4 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 7.4.2 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.

7.4.1 Pipeline Operation With Multiple Execute Packets in a Fetch Packet

Again referring to Figure 7–6 on page 7-6, pipeline operation is shown with eight instructions in every fetch packet. Figure 7–26, however, shows the pipeline operation with a fetch packet that contains multiple execute packets. Code for Figure 7–26 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 7–26. Pipeline Operation: Fetch Packets With Different Numbers of Execute Packets

Fetch packet (FP)	Execute packet (EP)	Clock cycle																
		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

In Figure 7–26, 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.

7.4.2 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 7–27 shows how a multicycle **NOP** can drive the execution of other instructions in the same execute packet. Figure 7–27(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 7–27(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 7–27. Multicycle NOP in an Execute Packet

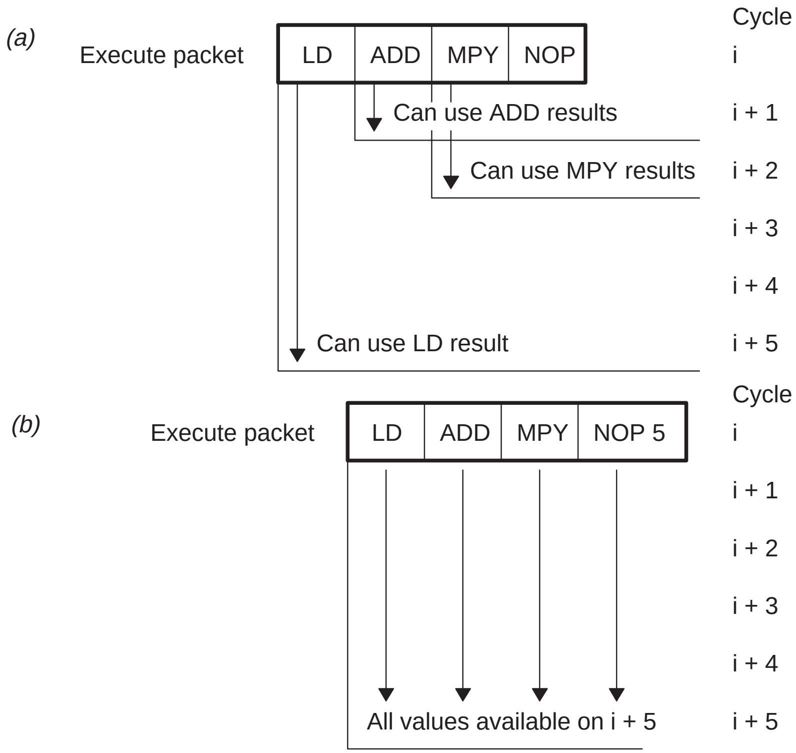
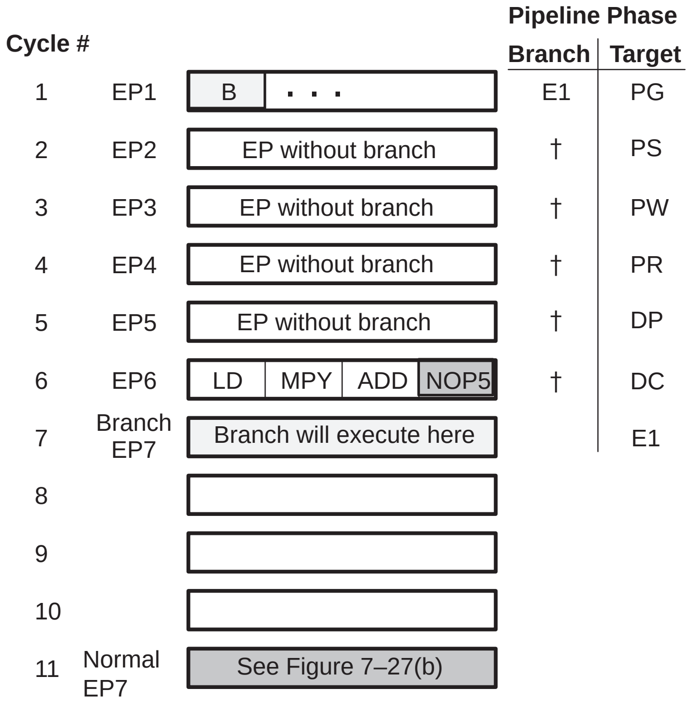


Figure 7–28 shows how a multicycle **NOP** can be affected by a branch. If the delay slots of a branch finish while a multicycle **NOP** is still dispatching **NOP**s into the pipeline, the branch overrides the multicycle **NOP** and the branch target begins execution five delay slots after the branch was issued.

*Figure 7–28. Branching and Multicycle **NOP**s*



Cycle #			Pipeline Phase Branch	Pipeline Phase Target
1	EP1	B . . .	E1	PG
2	EP2	EP without branch	†	PS
3	EP3	EP without branch	†	PW
4	EP4	EP without branch	†	PR
5	EP5	EP without branch	†	DP
6	EP6	LD | MPY | ADD | NOP5	†	DC
7	Branch EP7	Branch will execute here		E1
8				
9				
10				
11	Normal EP7	See Figure 7–27(b)		

† Delay slots of the branch

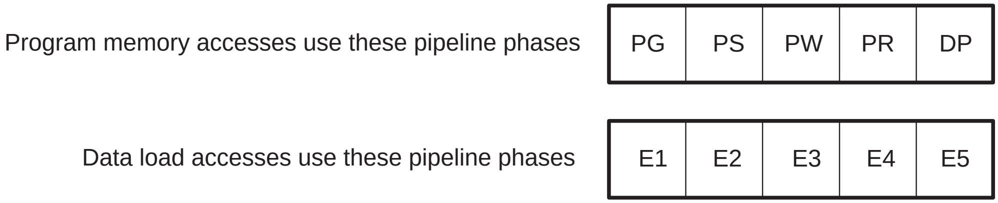
In one case, execute packet 1 (EP1) does not have a branch. The **NOP** 5 in EP6 will force the CPU to wait until cycle 11 to execute EP7.

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.

7.4.3 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 7–29.

Figure 7–29. Pipeline Phases Used During Memory Accesses


Program memory accesses use these pipeline phases
PG
PS
PW
PR
DP
Data load accesses use these pipeline phases
E1
E2
E3
E4
E5

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 7–33) to accommodate those memories. Table 7–33 shows the operation of program fetches pipeline versus the operation of a data load.

Table 7–33. 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

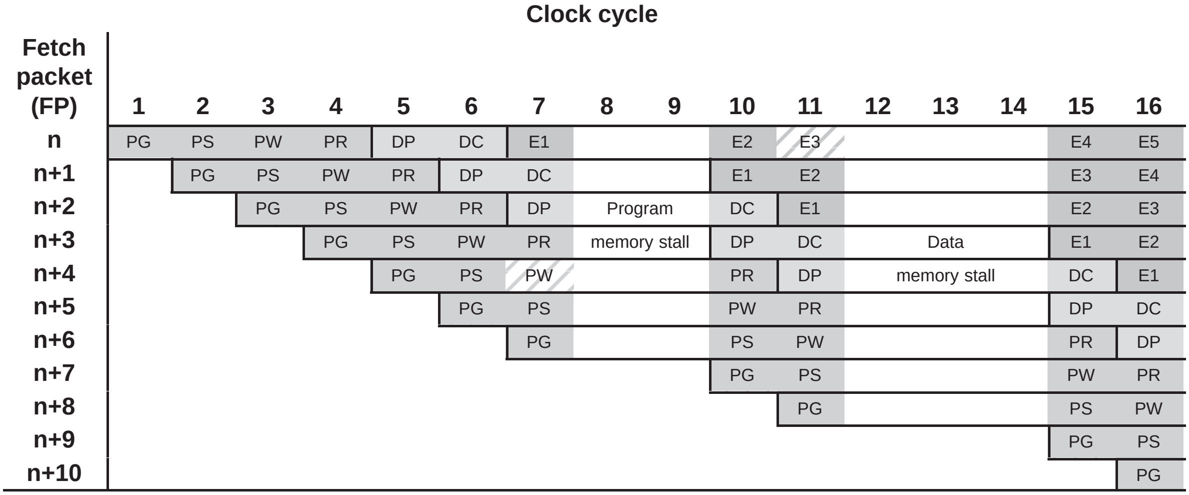
Depending on the type of memory and the time required to complete an access, the pipeline may stall to ensure proper coordination of data and instructions. This is discussed in section 7.4.3.1, *Memory Stalls.*

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 7.4.3.2, *Memory Bank Hits*.

7.4.3.1 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 7–30 illustrates this point.

Figure 7–30. Program and Data Memory Stalls

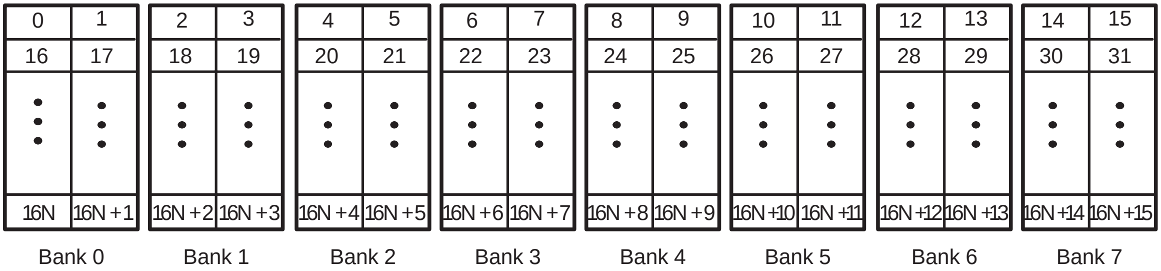


Fetch packet (FP)	Clock cycle 1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
n	PG	PS	PW	PR	DP	DC	E1			E2	E3				E4	E5
n+1		PG	PS	PW	PR	DP	DC			E1	E2				E3	E4
n+2			PG	PS	PW	PR	DP	Program		DC	E1				E2	E3
n+3				PG	PS	PW	PR	memory stall		DP	DC		Data		E1	E2
n+4					PG	PS	PW			PR	DP		memory stall		DC	E1
n+5						PG	PS			PW	PR				DP	DC
n+6							PG			PS	PW				PR	DP
n+7										PG	PS				PW	PR
n+8											PG				PS	PW
n+9															PG	PS
n+10																PG

7.4.3.2 Memory Bank Hits

Most C67x devices use an interleaved memory bank scheme, as shown in Figure 7–31. 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 7–31. 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 7–2. 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 7–34 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 7–2. 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
```

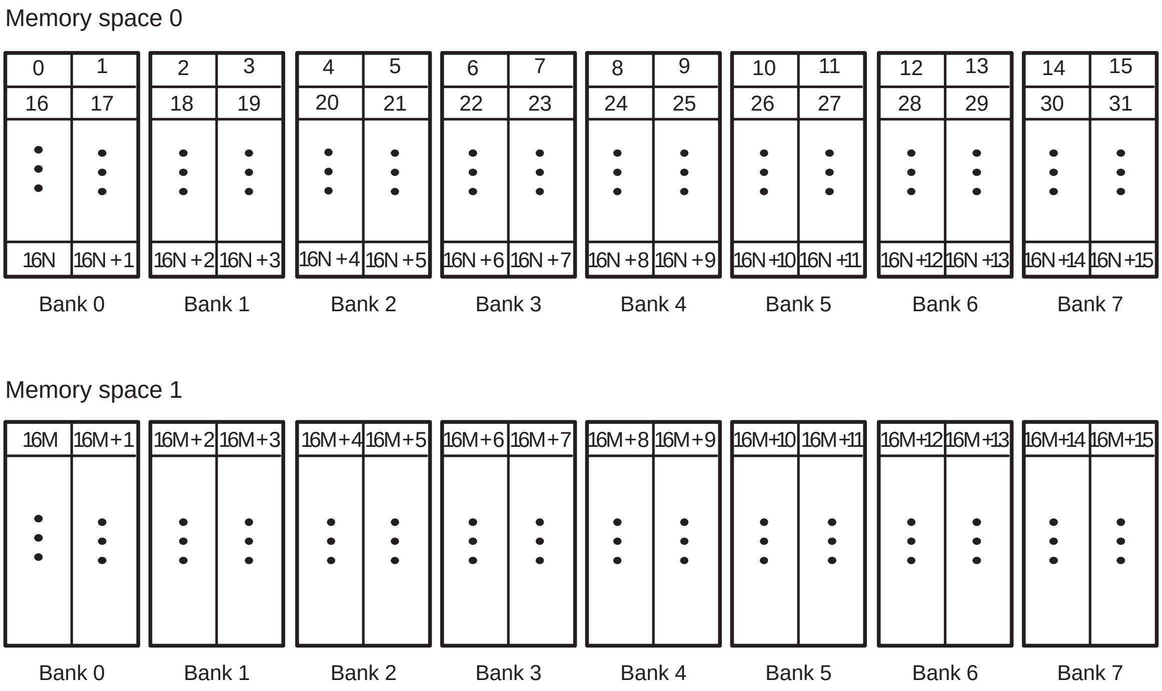
Table 7–34. Loads in Pipeline From Example 7–2

	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 7–32), 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 7–32. 8-Bank Interleaved Memory With Two Memory Spaces



Chapter 8

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**

8.1 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.

8.1.1 Types of Interrupts and Signals Used

There are three types of interrupts on the CPUs of the TMS320C6000™ DSPs. These three types are differentiated by their priorities, as shown in Table 8–1. 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 C6000 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 8–1. Interrupt Priorities

Priority	Interrupt Name
Highest	Reset
	NMI
	INT4
	INT5
	INT6
	INT7
	INT8
	INT9
	INT10
	INT11
	INT12
	INT13
	INT14
Lowest	INT15

8.1.1.1 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.

8.1.1.2 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.

8.1.1.3 Maskable Interrupts (INT4–INT15)

The CPUs of the C6000™ DSPs have 12 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.

8.1.1.4 Interrupt Acknowledgment (IACK and INUMx)

The IACK and INUMx signals alert hardware external to the C6000 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.

8.1.2 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 8–1. 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 8–1. Interrupt Service Table

Interrupt service table
(IST)

000h	$\overline{\text{RESET}}$ ISFP
020h	NMI ISFP
040h	Reserved
060h	Reserved
080h	INT4 ISFP
0A0h	INT5 ISFP
0C0h	INT6 ISFP
0E0h	INT7 ISFP
100h	INT8 ISFP
120h	INT9 ISFP
140h	INT10 ISFP
160h	INT11 ISFP
180h	INT12 ISFP
1A0h	INT13 ISFP
1C0h	INT14 ISFP
1E0h	INT15 ISFP

Program memory

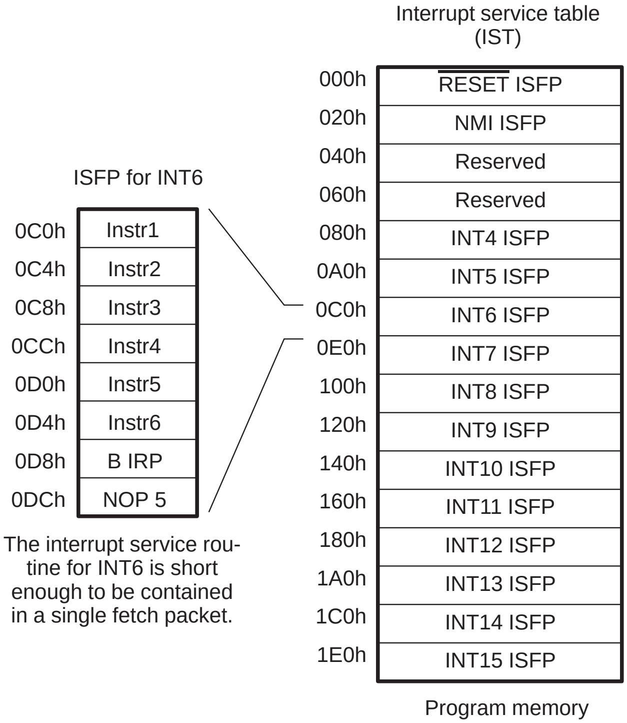
8.1.2.1 Interrupt Service Fetch Packet (ISFP)

An ISFP is a fetch packet used to service an interrupt. Figure 8–2 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:

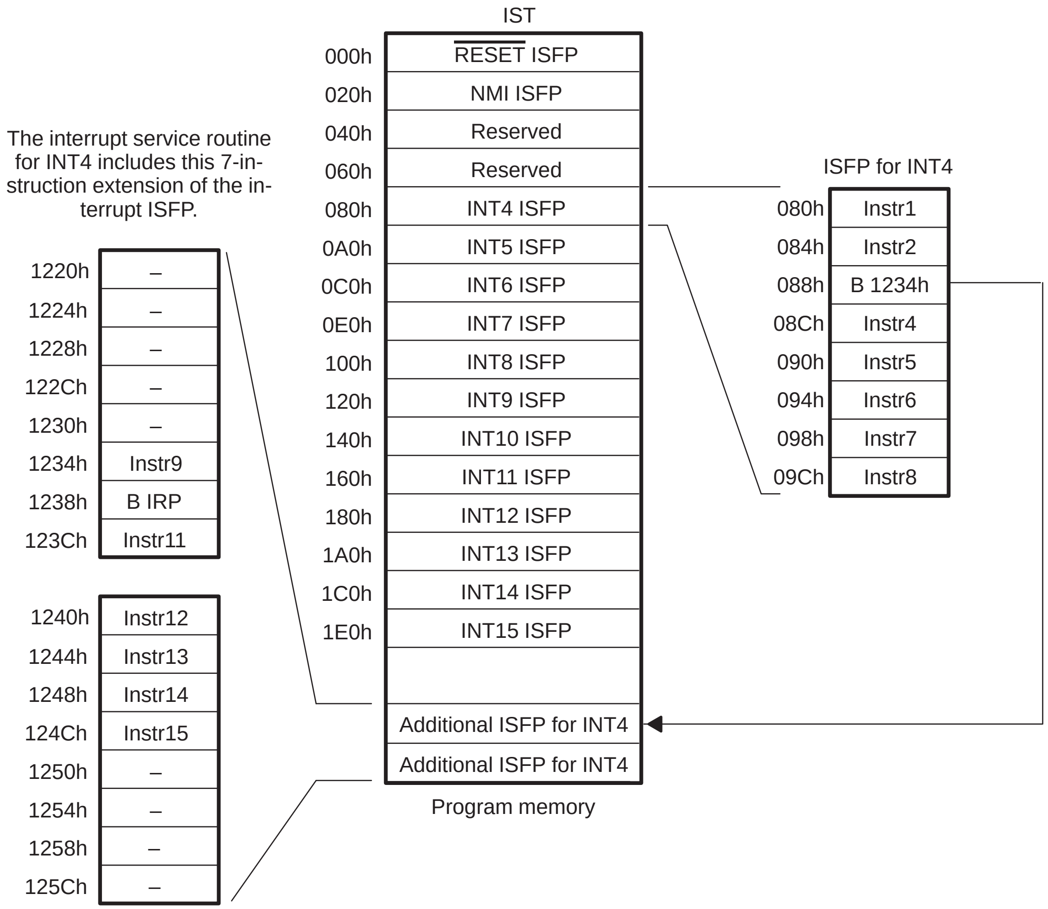
If the **NOP** 5 was not in the routine, the CPU would execute the next five execute packets that are associated with the next ISFP.

Figure 8–2. 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 8–3 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 8–3. IST With Branch to Additional Interrupt Service Code Located Outside the IST



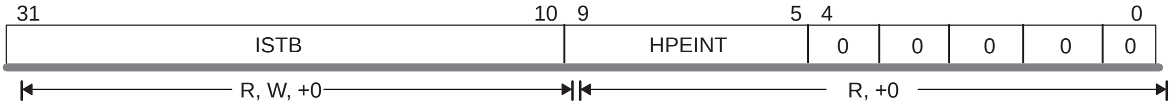
Note:

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.

8.1.2.2 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 8–4 shows the fields of the ISTP. Table 8–2 describes the fields and how they are used.

Figure 8–4. 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 8–2. 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 8–1) 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 8–1.

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 8–1 shows the relationship of the ISTB to the table location.

Example 8–1. 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

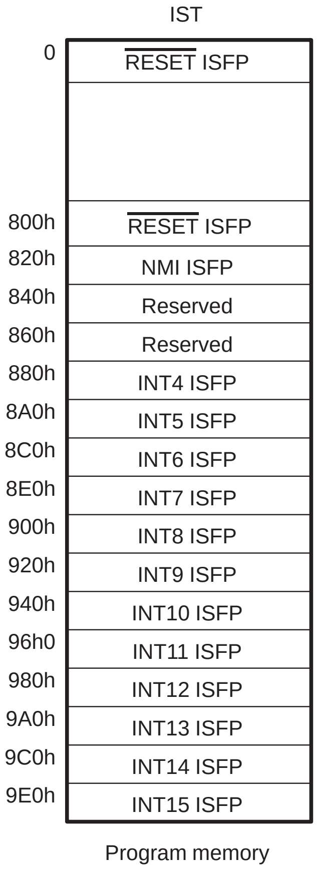
(b) How the ISTP directs the CPU to the appropriate ISFP in the relocated IST

Assume: IFR = BBC0h = 101$\underline{1}$ 10$\underline{1}$1 1100 0000b
IER = 1230h = 000$\underline{1}$ 00$\underline{1}$0 0011 0001b

2 enabled interrupts pending: INT9 and INT12

The 1s in the IFR indicate pending interrupts; the 1s in the IER indicate the interrupts that are enabled. INT9 has a higher priority than INT12, so HPEINT is encoded with the value for INT9, 01001b.

HPEINT corresponds to bits 9–5 of the ISTP:
ISTP = 1001 0010 0000b = 920h = address of INT9



	IST
0	$\overline{\text{RESET}}$ ISFP
800h	$\overline{\text{RESET}}$ ISFP
820h	NMI ISFP
840h	Reserved
860h	Reserved
880h	INT4 ISFP
8A0h	INT5 ISFP
8C0h	INT6 ISFP
8E0h	INT7 ISFP
900h	INT8 ISFP
920h	INT9 ISFP
940h	INT10 ISFP
96h0	INT11 ISFP
980h	INT12 ISFP
9A0h	INT13 ISFP
9C0h	INT14 ISFP
9E0h	INT15 ISFP

Program memory

8.1.3 Summary of Interrupt Control Registers

Table 8–3 lists the eight interrupt control registers on the C6000 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.

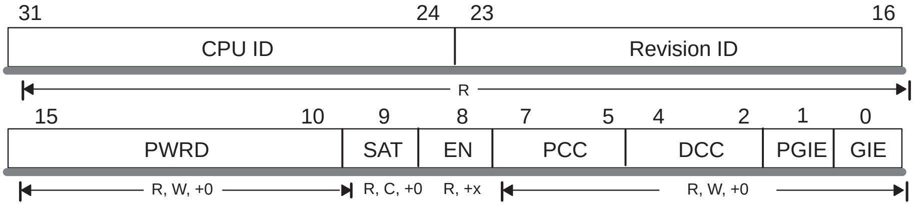
Table 8–3. Interrupt Control Registers

Abbreviation	Name	Description	Page Number
CSR	Control status register	Allows you to globally set or disable interrupts	8-11
IER	Interrupt enable register	Allows you to enable interrupts	8-13
IFR	Interrupt flag register	Shows the status of interrupts	8-14
ISR	Interrupt set register	Allows you to set flags in the IFR manually	8-14
ICR	Interrupt clear register	Allows you to clear flags in the IFR manually	8-14
ISTP	Interrupt service table pointer	Pointer to the beginning of the interrupt service table	8-8
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.	8-16
IRP	Interrupt return pointer	Contains the return address used on return from a maskable interrupt. This return is accomplished via the B IRP instruction.	8-17

8.2 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 8–5 and Table 8–4. The other fields of the registers serve other purposes and are discussed in section 2.6.3 on page 2-17.

Figure 8–5. 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 8–4. 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 8–2 shows how to disable maskable interrupts globally and Example 8–3 shows how to enable maskable interrupts globally.

Example 8–2. 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 8–3. 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
```

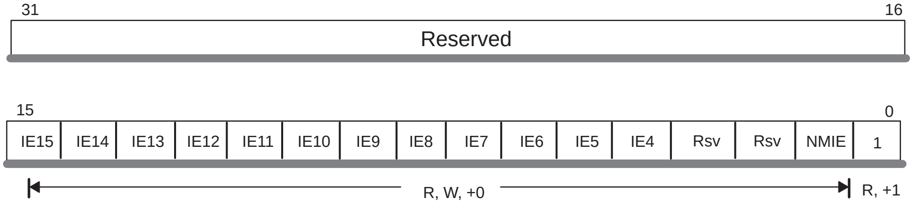
8.3 Individual Interrupt Control

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.

8.3.1 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 8–6.

Figure 8–6. 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 8–4 and Example 8–5 show code for enabling and disabling individual interrupts, respectively.

Example 8–4. 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
```

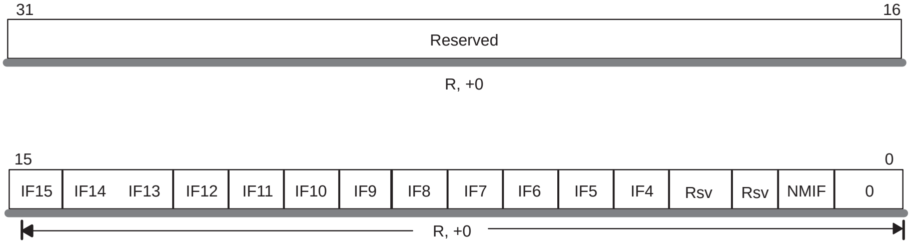
Example 8–5. Code Sequence to Disable an Individual Interrupt (INT9)

```
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
```

8.3.2 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 8–7 shows the IFR.

Figure 8–7. 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 8–8, and the interrupt clear register (ICR), shown in Figure 8–9, 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.

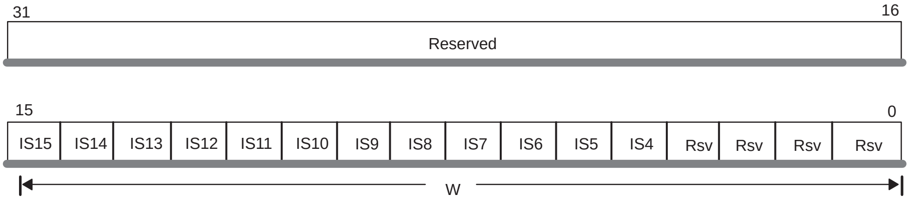
Note:

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.

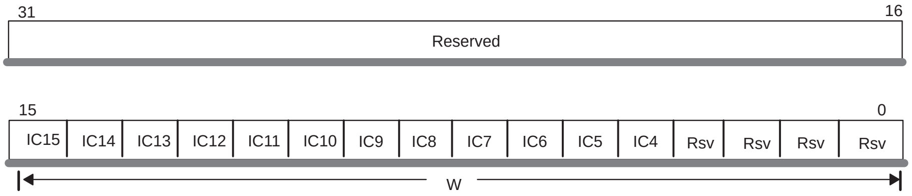
Example 8–6 and Example 8–7 show code examples to set and clear individual interrupts.

Figure 8–8. Interrupt Set Register (ISR)



Legend: W = Writeable by the **MVC** instruction
Rsv = Reserved

Figure 8–9. Interrupt Clear Register (ICR)



Legend: W = Writeable by the **MVC** instruction
Rsv = Reserved

Example 8–6. Code to Set an Individual Interrupt (INT6) and Read the Flag Register

```
MVK      40h,B3
MVC      B3,ISR
NOP
MVC      IFR,B4
```

Example 8–7. Code to Clear an Individual Interrupt (INT6) and Read the Flag Register

```
MVK      40h,B3
MVC      B3,ICR
NOP
MVC      IFR,B4
```

8.3.3 Returning From Interrupt Servicing

After $\overline{\text{RESET}}$ 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.

8.3.3.1 CPU State After $\overline{\text{RESET}}$

After $\overline{\text{RESET}}$, 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

8.3.3.2 Returning From Nonmaskable Interrupts (NMI Return Pointer Register–NRP)

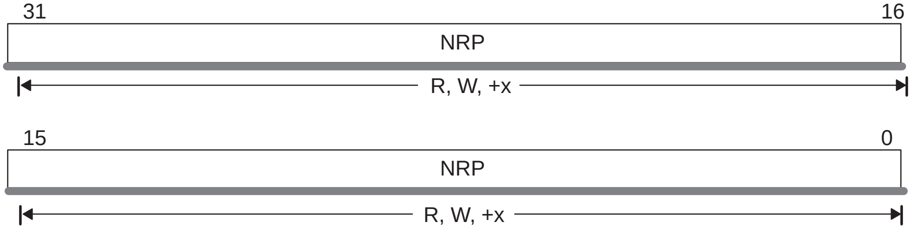
The NMI return pointer register (NRP) contains the return pointer that directs the CPU to the proper location to continue program execution after NMI processing. A branch using the address in the NRP (**B NRP**) in your interrupt service routine returns to the program flow when NMI servicing is complete. Example 8–8 shows how to return from an NMI.

Example 8–8. 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 8–10 shows the NRP register.

Figure 8–10. NMI Return Pointer (NRP)



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

8.3.3.3 Returning From Maskable Interrupts (Interrupt Return Pointer Register–IRP)

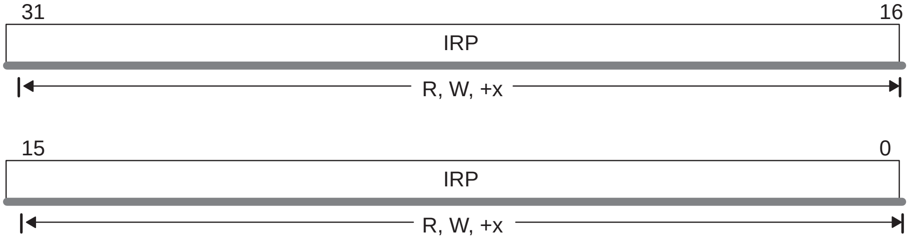
The interrupt return pointer register (IRP) contains the return pointer that directs the CPU to the proper location to continue program execution after processing a maskable interrupt. A branch using the address in the IRP (**B IRP**) in your interrupt service routine returns to the program flow when interrupt servicing is complete. Example 8–9 shows how to return from a maskable interrupt.

Example 8–9. 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 8–11 shows the IRP register.

Figure 8–11. Interrupt Return Pointer (IRP)



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

8.4 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.

8.4.1 Setting the Nonreset Interrupt Flag

Figure 8–12 and Figure 8–13 show the processing of a nonreset interrupt (INTm) for the TMS320C62x™ /TMS320C64x™ and TMS320C67x™ DSPs, 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 8–12 and Figure 8–13, 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 8–12 and Figure 8–13 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 a 1 to bit m of the ICR, or the processing of INTm occurs.

8.4.2 Conditions for Processing a Nonreset Interrupt

In clock cycle 4 of Figure 8–12 and Figure 8–13, 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).

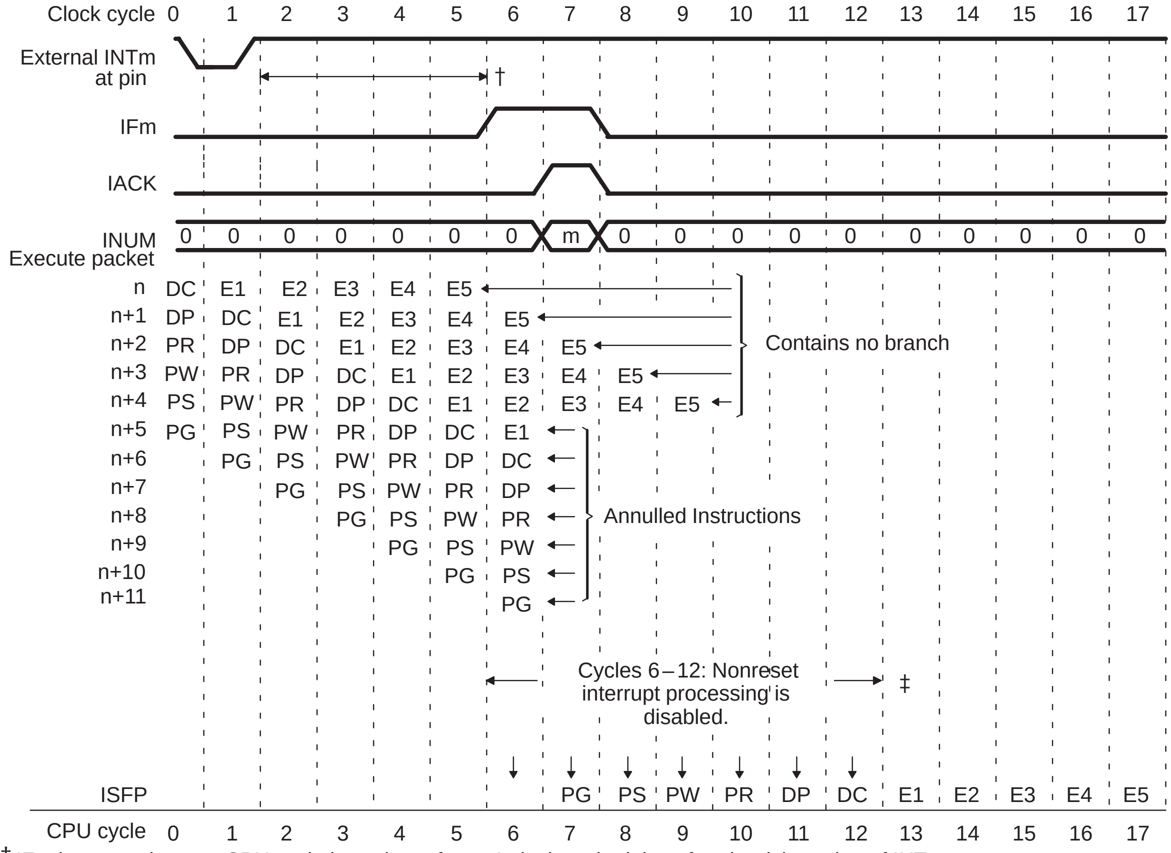
❑ GIE = 1

❑ 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.

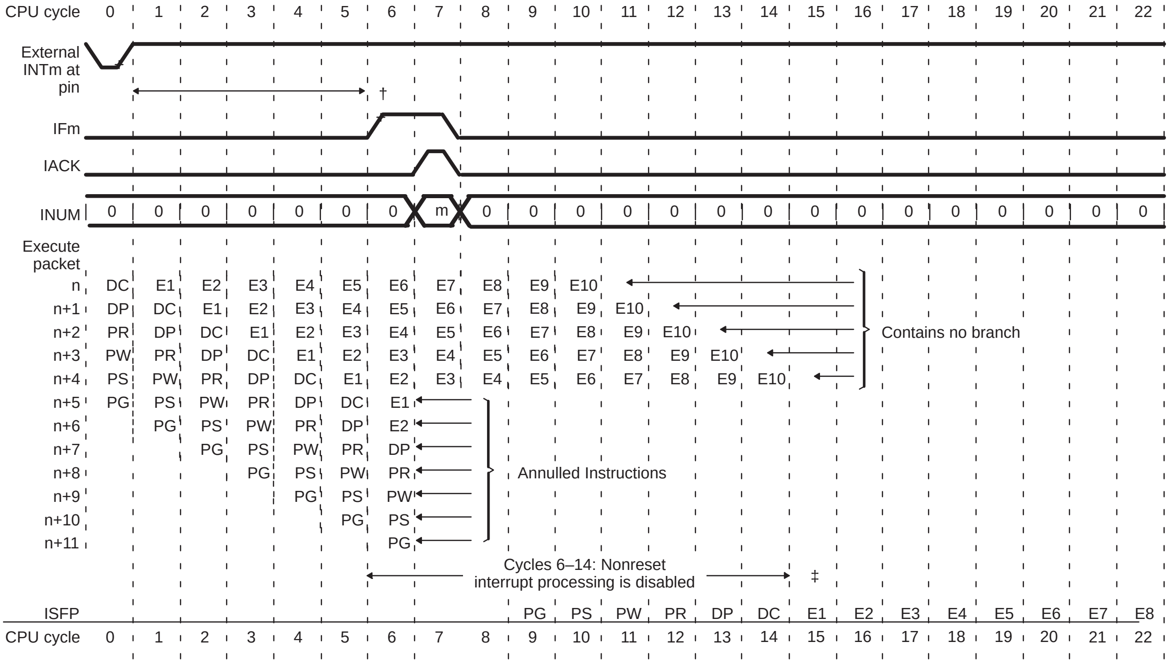
Figure 8–12. TMS320C62x/C64x Nonreset Interrupt Detection and Processing: Pipeline Operation


Clock cycle 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
External INTm at pin
†
IFm
IACK
INUM 0 0 0 0 0 0 0 m 0 0 0 0 0 0 0 0 0 0
Execute packet
n DC E1 E2 E3 E4 E5
n+1 DP DC E1 E2 E3 E4 E5
n+2 PR DP DC E1 E2 E3 E4 E5
n+3 PW PR DP DC E1 E2 E3 E4 E5
n+4 PS PW PR DP DC E1 E2 E3 E4 E5
n+5 PG PS PW PR DP DC E1
n+6 PG PS PW PR DP DC
n+7 PG PS PW PR DP
n+8 PG PS PW PR
n+9 PG PS PW
n+10 PG PS
n+11 PG
Contains no branch
Annulled Instructions
Cycles 6–12: Nonreset interrupt processing is disabled.
‡
ISFP PG PS PW PR DP DC E1 E2 E3 E4 E5
CPU cycle 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17

† 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 8–13. TMS320C67x Nonreset Interrupt Detection and Processing: Pipeline Operation


CPU cycle 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22
External INTm at pin
†
IFm
IACK
INUM 0 0 0 0 0 0 0 m 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Execute packet
n DC E1 E2 E3 E4 E5 E6 E7 E8 E9 E10
n+1 DP DC E1 E2 E3 E4 E5 E6 E7 E8 E9 E10
n+2 PR DP DC E1 E2 E3 E4 E5 E6 E7 E8 E9 E10
n+3 PW PR DP DC E1 E2 E3 E4 E5 E6 E7 E8 E9 E10
n+4 PS PW PR DP DC E1 E2 E3 E4 E5 E6 E7 E8 E9 E10
n+5 PG PS PW PR DP DC E1
n+6 PG PS PW PR DP E2
n+7 PG PS PW PR DP
n+8 PG PS PW PR
n+9 PG PS PW
n+10 PG PS
n+11 PG
Contains no branch
Annulled Instructions
Cycles 6–14: Nonreset interrupt processing is disabled
‡
ISFP PG PS PW PR DP DC E1 E2 E3 E4 E5 E6 E7 E8
CPU cycle 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22

† 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.

8.4.3 Actions Taken During Nonreset Interrupt Processing

During CPU cycles 6–12 of Figure 8–12 and cycles 6–14 of Figure 8–13, 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™ /C64x™ and cycle 9 for the C67x™ .

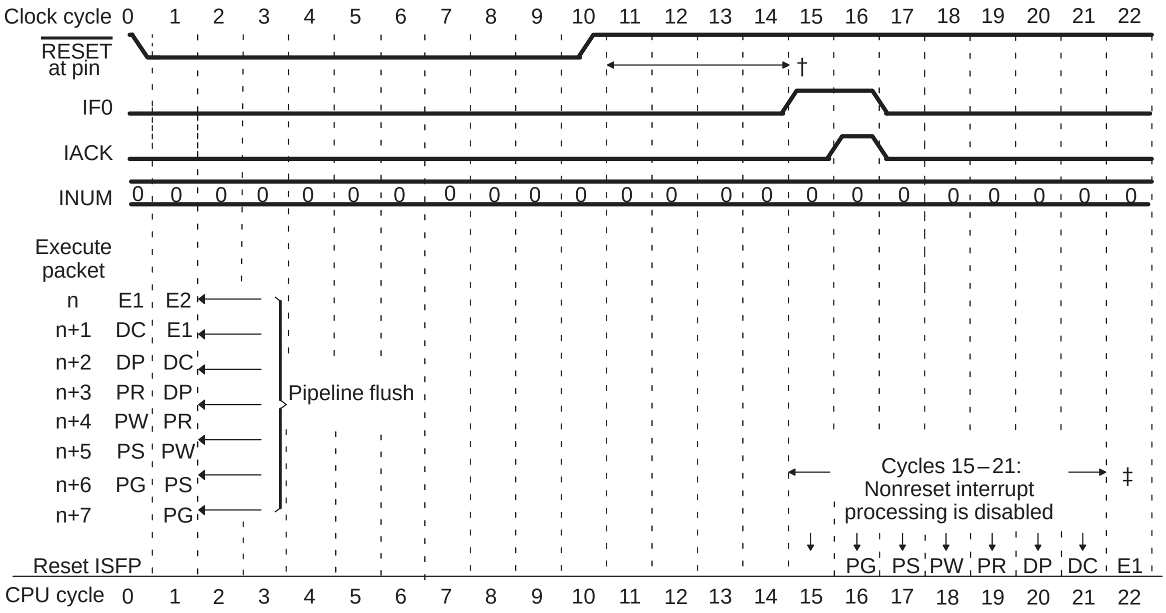
❑ During cycle 7, IACK is asserted and the proper INUMx signals are asserted to indicate which interrupt is being processed. The timings for these signals in Figure 8–12 and Figure 8–13 represent only the signals' characteristics inside the CPU. The external signals may be delayed and be longer in duration to handle external devices. Check the data sheet for your specific device for particular timing values.

❑ IFm is cleared during cycle 8.

8.4.4 Setting the $\overline{\text{RESET}}$ Interrupt Flag for the TMS320C6000

$\overline{\text{RESET}}$ 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 8–14, 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 8–14. $\overline{\text{RESET}}$ Interrupt Detection and Processing: Pipeline Operation



Clock cycle 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22
RESET at pin
†
IF0
IACK
INUM
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Execute packet
n E1 E2
n+1 DC E1
n+2 DP DC
n+3 PR DP
n+4 PW PR
n+5 PS PW
n+6 PG PS
n+7 PG
Pipeline flush
Cycles 15 – 21: Nonreset interrupt processing is disabled
‡
Reset ISFP
PG PS PW PR DP DC E1
CPU cycle 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22

† 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.

8.4.5 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 8.3.3.1 on page 8-16.

During CPU cycles 15–21 of Figure 8–14, 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.

8.5 Performance Considerations

The interaction of the C6000 CPU and sources of interrupts present performance issues for you to consider when you are developing your code.

8.5.1 General Performance

- ❑ **Overhead**. Overhead for all CPU interrupts is seven cycles for the C62x/C64x and nine cycles for the C67x. You can see this in Figure 8–12 and Figure 8–13, where no new instructions are entering the E1 pipeline phase during CPU cycles 6 through 12 for the C62x/C64x and CPU cycles 6 through 14 for the C67x.
- ❑ **Latency.** Interrupt latency is 11 cycles for the C62x/C64x and 13 cycles for the C67x (21 cycles for $\overline{\text{RESET}}$). In Figure 8–13, although the interrupt is active in cycle 2, execution of interrupt service code does not begin until cycle 13 for the C62x//C64x 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.

8.5.2 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 8–12 or Figure 8–13 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.

8.6 Programming Considerations

The interaction of the C6000 CPUs and sources of interrupts present programming issues for you to consider when you are developing your code.

8.6.1 Single Assignment Programming

Example 8–10 shows code without single assignment and Example 8–11 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 8–10. 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 8–11, 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 8–10. 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 8–11. Code Using Single Assignment

```
LDW    .D1    *A0,A6
ADD    .L1    A1,A2,A3
NOP           3
MPY    .M1    A6,A4,A5 ; uses A6
```

8.6.2 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.

8.6.3 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 8–12.

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 8–12. 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)
```

8.6.4 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 8–13 and Example 8–14 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 8–14 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 8–13. 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 8–14. 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