TMS320C67x/C67x+ DSP CPU and Instruction Set Reference Guide

Literature Number: SPRU733A
November 2006

TEXAS INSTRUMENTS

IMPORTANT NOTICE

Following are URLs where you can obtain information on other Texas Instruments products and application solutions:

Products		**Applications**	
Amplifiers	amplifier.ti.com	Audio	www.ti.com/audio
Data Converters	dataconverter.ti.com	Automotive	www.ti.com/automotive
DSP	dsp.ti.com	Broadband	www.ti.com/broadband
Interface	interface.ti.com	Digital Control	www.ti.com/digitalcontrol
Logic	logic.ti.com	Military	www.ti.com/military
Power Mgmt	power.ti.com	Optical Networking	www.ti.com/opticalnetwork
Microcontrollers	microcontroller.ti.com	Security	www.ti.com/security
Low Power Wireless	www.ti.com/lpw	Telephony	www.ti.com/telephony
		Video & Imaging	www.ti.com/video
		Wireless	www.ti.com/wireless

Mailing Address: Texas Instruments
Post Office Box 655303 Dallas, Texas 75265

Preface

Read This First

About This Manual

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 TMS320C67x+™ DSP is an enhancement of the C67x™ DSP with added functionality and an expanded instruction set. This document describes the CPU architecture, pipeline, instruction set, and interrupts of the C67x and C67x+™ DSPs.

Notational Conventions

This document uses the following conventions.

- Any reference to the C67x DSP or C67x CPU also applies, unless otherwise noted, to the C67x+ DSP and C67x+ CPU, respectively.
- Hexadecimal numbers are shown with the suffix h. For example, the following number is 40 hexadecimal (decimal 64): 40h.

Related Documentation From Texas Instruments

The following documents describe the C6000™ devices and related support tools. Copies of these documents are available on the Internet at www.ti.com. *Tip:* Enter the literature number in the search box provided at www.ti.com.

The current documentation that describes the C6000 devices, related peripherals, and other technical collateral, is available in the C6000 DSP product folder at: www.ti.com/c6000.

TMS320C6000 DSP Peripherals Overview Reference Guide (literature number SPRU190) describes the peripherals available on the TMS320C6000 DSPs.

TMS320C672x DSP Peripherals Overview Reference Guide (literature number SPRU723) describes the peripherals available on the TMS320C672x DSPs.

TMS320C6000 Technical Brief (literature number SPRU197) gives an introduction to the TMS320C62x and TMS320C67x DSPs, development tools, and third-party support.

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

Code Composer Studio Application Programming Interface Reference Guide (literature number SPRU321) describes the Code Composer Studio application programming interface (API), which allows you to program custom plug-ins for Code Composer.

TMS320C6x Peripheral Support Library Programmer's Reference (literature number SPRU273) describes the contents of the TMS320C6000 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.

TMS320C6000 Chip Support Library API Reference Guide (literature number SPRU401) describes a set of application programming interfaces (APIs) used to configure and control the on-chip peripherals.

Trademarks

Code Composer Studio, C6000, C64x, C67x, C67x+, TMS320C2000, TMS320C5000, TMS320C6000, TMS320C62x, TMS320C64x, TMS320C67x, TMS320C67x+, TMS320C672x, and VelociTI are trademarks of Texas Instruments.

Trademarks are the property of their respective owners.

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. All three DSP generations use the VelociTI™ architecture, a high-performance, advanced very long instruction word (VLIW) architecture, making these DSPs excellent choices for multichannel and multifunction applications.

The TMS320C67x+ DSP is an enhancement of the C67x DSP with added functionality and an expanded instruction set.

Any reference to the C67x DSP or C67x CPU also applies, unless otherwise noted, to the C67x+ DSP and C67x+ CPU, respectively.

Topic **Page**

1.1 TMS320 DSP 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.

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.

1.2 TMS320C6000 DSP Family Overview

With a performance of up to 6000 million instructions per second (MIPS) and an efficient C compiler, the TMS320C6000 DSPs give system architects unlimited possibilities to differentiate their products. High performance, ease of use, and affordable pricing make the C6000 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 C6000 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)

Table 1−1. Typical Applications for the TMS320 DSPs

Automotive	Consumer	Control
Adaptive ride control Antiskid brakes Cellular telephones Digital radios Engine control Global positioning Navigation Vibration analysis Voice commands	Digital radios/TVs Educational toys Music synthesizers Pagers Power tools Radar detectors Solid-state answering machines	Disk drive control Engine control Laser printer control Motor control Robotics control Servo control
General-Purpose	**Graphics/Imaging**	**Industrial**
Adaptive filtering Convolution Correlation Digital filtering Fast Fourier transforms Hilbert transforms Waveform generation Windowing	3-D transformations Animation/digital maps Homomorphic processing Image compression/transmission Image enhancement Pattern recognition Robot vision Workstations	Numeric control Power-line monitoring Robotics Security access
Instrumentation	**Medical**	**Military**
Digital filtering Function generation Pattern matching Phase-locked loops Seismic processing Spectrum analysis Transient analysis	Diagnostic equipment Fetal monitoring Hearing aids Patient monitoring Prosthetics Ultrasound equipment	Image processing Missile guidance Navigation Radar processing Radio frequency modems Secure communications Sonar processing
Telecommunications		**Voice/Speech**
1200- to 56 600-bps modems Adaptive equalizers ADPCM transcoders Base stations Cellular telephones Channel multiplexing Data encryption Digital PBXs Digital speech interpolation (DSI) DTMF encoding/decoding Echo cancellation	Faxing Future terminals Line repeaters Personal communications systems (PCS) Personal digital assistants (PDA) Speaker phones Spread spectrum communications Digital subscriber loop (xDSL) Video conferencing X.25 packet switching	Speaker verification Speech enhancement Speech recognition Speech synthesis Speech vocoding Text-to-speech Voice mail

1.3 TMS320C67x DSP Features and Options

The C6000 devices execute up to eight 32-bit instructions per cycle. The C67x CPU consists of 32 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™ and XDS560™ 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 devices include these additional features:

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

In addition to the features of the C67x device, the C67x+ device is enhanced for code size improvement and floating-point performance. These additional features include:

- Execute packets can span fetch packets.
- Register file size is increased to 64 registers (32 in each datapath).
- Floating-point addition and subtraction capability in the .S unit.
- Mixed-precision multiply instructions.
- 32-KByte instruction cache that supports execution from both on-chip RAM and ROM as well as from external memory through a VBUSP-based external memory interface (EMIF).
- Unified memory controller features support for flat on-chip data RAM and ROM organizations for zero wait-state accesses from both load store units of the CPU. The memory controller supports different banking organizations for RAM and ROM arrays. The memory controller also supports VBUSP interfaces (two master and one slave) for transfer of data from the system peripherals to and from the CPU and internal memory. A VBUSP-based DMA controller can interface to the CPU for programmable bulk transfers through the VBUSP slave port.

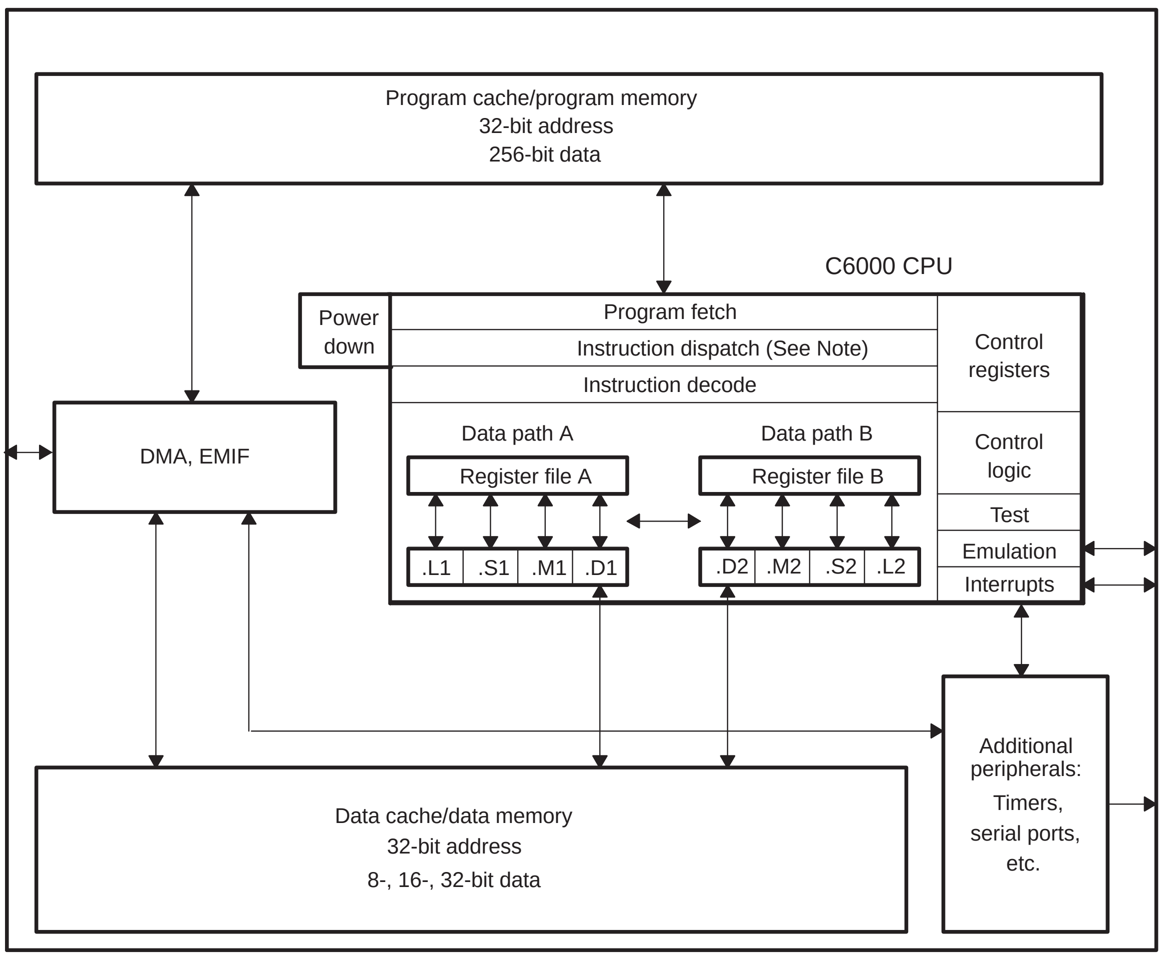
The VelociTI architecture of the C6000 platform of devices make them the first off-the-shelf DSPs to use advanced VLIW to achieve high performance through increased instruction-level parallelism. A traditional VLIW architecture consists of multiple execution units running in parallel, performing multiple 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 TMS320C6000 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.

1.4 TMS320C67x DSP Architecture

Figure 1−1 is the block diagram for the C67x DSP. 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. TMS320C67x DSP Block Diagram



1.4.1 Central Processing Unit (CPU)

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

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

The program fetch, instruction dispatch, and instruction decode units can deliver up to eight 32-bit instructions to the functional units every CPU clock cycle. The processing of instructions occurs in each of the two data paths (A and B), each of which contains four functional units (.L, .S, .M, and .D) and 16 32-bit general-purpose registers. The data paths are described in more detail in Chapter 2. A control register file provides the means to configure and control various processor operations. To understand how instructions are fetched, dispatched, decoded, and executed in the data path, see Chapter 4.

1.4.2 Internal Memory

The C67x DSP has a 32-bit, byte-addressable address space. Internal (on-chip) memory is organized in separate data and program spaces. When off-chip memory is used, these spaces are unified on most devices to a single memory space via the external memory interface (EMIF).

The C67x DSP has two 32-bit internal ports to access internal data memory. The C67x DSP has 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 (C6701 DSP only) 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 TMS320C5000™ 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 an overview of the peripherals available on the C6000 DSP, refer to the *TM320C6000 DSP Peripherals Overview 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.

2.1 Introduction

The components of the data path for the TMS320C67x CPU are shown in Figure 2−1. These components consist of:

- Two general-purpose register files (A and B)
- Eight functional units (.L1, .L2, .S1, .S2, .M1, .M2, .D1, and .D2)
- Two load-from-memory 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)

2.2 General-Purpose Register Files

There are two general-purpose register files (A and B) in the C6000 data paths. For the C67x DSP, each of these files contains 16 32-bit registers (A0–A15 for file A and B0–B15 for file B), as shown in Table 2−1. For the C67x+ DSP, the register file size is doubled to 32 32-bit registers (A0–A31 for file A and B0–B21 for file B), as shown in Table 2−1. The general-purpose registers can be used for data, data address pointers, or condition registers.

The C67x DSP 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 (that is always an odd-numbered register). 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 C67x DSP cores. In assembly language syntax, a colon between the register names denotes the register pairs, and the odd-numbered register is specified first.

The additional registers are addressed by using the previously unused fifth (msb) bit of the source and register specifiers. All 64-bit register writes and reads are performed over 2 cycles as per the current C67x devices.

Figure 2−2 shows 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 2−1. TMS320C67x CPU Data Paths

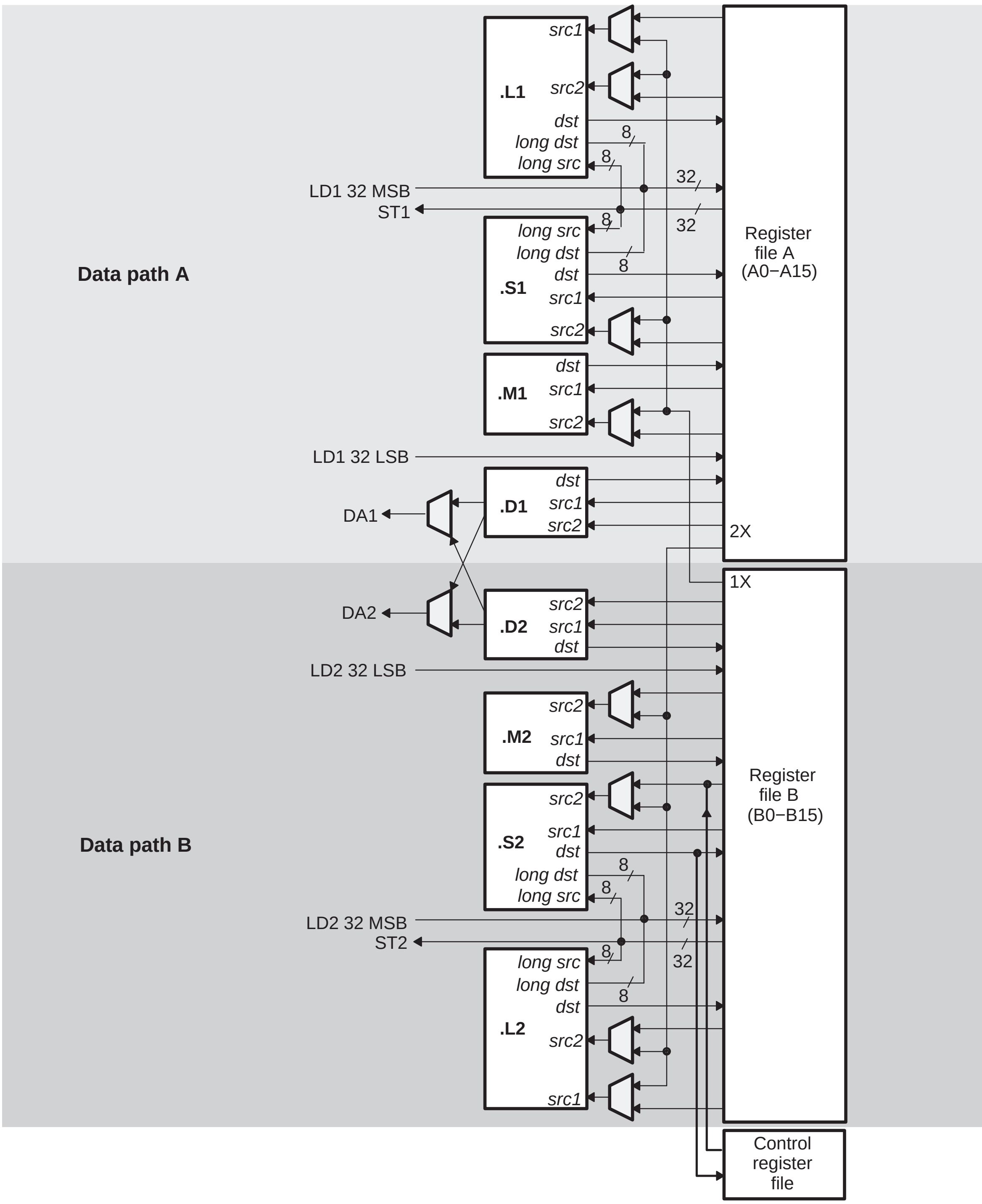
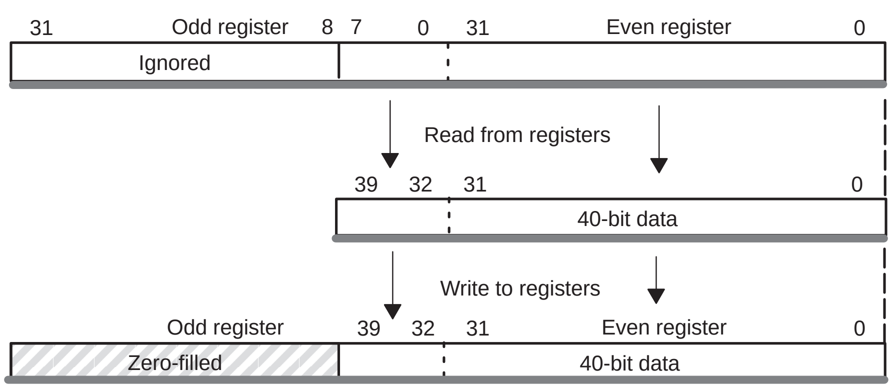


Table 2−1. 40-Bit/64-Bit Register Pairs

Register Files		Devices
A	B	
A1:A0	B1:B0	C67x DSP
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	C67x+ DSP 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 2−2. Storage Scheme for 40-Bit Data in a Register Pair



2.3 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.

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 2−1). 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.

See Appendix B for a list of the instructions that execute on each functional unit.

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	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)	Compare Reciprocal and reciprocal square-root operations Absolute value operations SP → DP conversion operations SPand DP adds and subtracts SP and DP reverse subtracts (src2 − src1)
.M unit (.M1, .M2)	16 × 16-bit multiply operations 32 × 32-bit multiply operations	Floating-point multiply operations Mixed-precision 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 doubleword with 5-bit constant offset

2.4 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 C67x DSP, 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.

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 C67x DSP, only one functional unit per data path, per execute packet, can get an operand from the opposite register file.

2.5 Memory, Load, and Store Paths

The C67x DSP 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 DSP 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.

On the C6000 architecture, some of the ports for long and doubleword operands are shared between functional units. This places a constraint on which long or doubleword operations can be scheduled on a data path in the same execute packet. See section 3.7.5.

2.6 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 C67x DSP, LD1 is comprised of LD1a and LD1b to support 64-bit loads. Similarly, T2 consists of the DA2 address path and the LD2 and ST2 data paths. For the C67x DSP, LD2 is comprised of LD2a and LD2b to support 64-bit loads.

The T1 and T2 designations appear in the functional unit fields for load and store instructions. For example, the following load instruction 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
```

2.7 Control Register File

Table 2−3 lists the control registers contained in the control register file.

Table 2−3. Control Registers

Acronym	Register Name	Section
AMR	Addressing mode register	2.7.3
CSR	Control status register	2.7.4
ICR	Interrupt clear register	2.7.5
IER	Interrupt enable register	2.7.6
IFR	Interrupt flag register	2.7.7
IRP	Interrupt return pointer register	2.7.8
ISR	Interrupt set register	2.7.9
ISTP	Interrupt service table pointer register	2.7.10
NRP	Nonmaskable interrupt return pointer register	2.7.11
PCE1	Program counter, E1 phase	2.7.12

2.7.1 Register Addresses for Accessing the Control Registers

Table 2−4 lists the register addresses for accessing the control register file. One unit (.S2) can read from and write to the control register file. Each control register is accessed by the **MVC** instruction. See the **MVC** instruction description, page 3-179, 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 control status register (CSR).

Table 2−4. Register Addresses for Accessing the Control Registers

Acronym	Register Name	Address	Read/ Write
AMR	Addressing mode register	00000	R, W
CSR	Control status register	00001	R, W
FADCR	Floating-point adder configuration	10010	R, W
FAUCR	Floating-point auxiliary configuration	10011	R, W
FMCR	Floating-point multiplier configuration	10100	R, W
ICR	Interrupt clear register	00011	W
IER	Interrupt enable register	00100	R, W
IFR	Interrupt flag register	00010	R
IRP	Interrupt return pointer	00110	R, W
ISR	Interrupt set register	00010	W
ISTP	Interrupt service table pointer	00101	R, W
NRP	Nonmaskable interrupt return pointer	00111	R, W
PCE1	Program counter, E1 phase	10000	R

Legend: R = Readable by the **MVC** instruction; W = Writeable by the **MVC** instruction

2.7.2 Pipeline/Timing of Control Register Accesses

All **MVC** instructions 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 conversely. 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 but the modification of the IFR bits occurs one clock later. For more information on the manipulation of ISR, ICR, and IFR, see section 2.7.9, section 2.7.5, and section 2.7.7.

Saturating instructions, such as **SADD**, set the saturation flag bit (SAT) in 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 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 CSR and IER, respectively, there are no delay slots between when the branch is issued and when the control register updates take effect.

2.7.3 Addressing Mode Register (AMR)

For each of the eight registers (A4–A7, B4–B7) that can perform linear or circular addressing, the addressing mode register (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 2−3 and described in Table 2−5.

Figure 2−3. Addressing Mode Register (AMR)

31–26	25–21	20–16
Reserved	BK1	BK0
R-0	R/W-0	R/W-0

15–14	13–12	11–10	9–8	7–6	5–4	3–2	1–0
B7 MODE	B6 MODE	B5 MODE	B4 MODE	A7 MODE	A6 MODE	A5 MODE	A4 MODE
R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0

Legend: R = Readable by the **MVC** instruction; W = Writeable by the **MVC** instruction; -*n* = value after reset

Table 2−5. Addressing Mode Register (AMR) Field Descriptions

Bit	Field	Value	Description
31−26	Reserved	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.
25−21	BK1	0−1Fh	Block size field 1. A 5-bit value used in calculating block sizes for circular addressing. Table 2−6 shows block size calculations for all 32 possibilities. *Block size (in bytes) =* $2^{(N+1)}$, where N is the 5-bit value in BK1
20−16	BK0	0−1Fh	Block size field 0. A 5-bit value used in calculating block sizes for circular addressing. Table 2−6 shows block size calculations for all 32 possibilities. *Block size (in bytes) =* $2^{(N+1)}$, where N is the 5-bit value in BK0
15−14	B7 MODE	0−3h	Address mode selection for register file B7.
		0	Linear modification (default at reset)
		1h	Circular addressing using the BK0 field
		2h	Circular addressing using the BK1 field
		3h	Reserved

Table 2−5. Addressing Mode Register (AMR) Field Descriptions (Continued)

Bit	Field	Value	Description
13−12	B6 MODE	0−3h	Address mode selection for register file B6.
		0	Linear modification (default at reset)
		1h	Circular addressing using the BK0 field
		2h	Circular addressing using the BK1 field
		3h	Reserved
11−10	B5 MODE	0−3h	Address mode selection for register file B5.
		0	Linear modification (default at reset)
		1h	Circular addressing using the BK0 field
		2h	Circular addressing using the BK1 field
		3h	Reserved
9−8	B4 MODE	0−3h	Address mode selection for register file B4.
		0	Linear modification (default at reset)
		1h	Circular addressing using the BK0 field
		2h	Circular addressing using the BK1 field
		3h	Reserved
7−6	A7 MODE	0−3h	Address mode selection for register file A7.
		0	Linear modification (default at reset)
		1h	Circular addressing using the BK0 field
		2h	Circular addressing using the BK1 field
		3h	Reserved
5−4	A6 MODE	0−3h	Address mode selection for register file A6.
		0	Linear modification (default at reset)
		1h	Circular addressing using the BK0 field
		2h	Circular addressing using the BK1 field
		3h	Reserved

Table 2−5. Addressing Mode Register (AMR) Field Descriptions (Continued)

Bit	Field	Value	Description
3−2	A5 MODE	0−3h	Address mode selection for register file a5.
		0	Linear modification (default at reset)
		1h	Circular addressing using the BK0 field
		2h	Circular addressing using the BK1 field
		3h	Reserved
1−0	A4 MODE	0−3h	Address mode selection for register file A4.
		0	Linear modification (default at reset)
		1h	Circular addressing using the BK0 field
		2h	Circular addressing using the BK1 field
		3h	Reserved

Table 2−6. Block Size Calculations

BK*n* Value	**Block Size**	**BK*n* Value**	**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.

2.7.4 Control Status Register (CSR)

The control status register (CSR) contains control and status bits. The CSR is shown in Figure 2−4 and described in Table 2−7. For the PWRD, EN, PCC, and DCC fields, see the device-specific data manual to see if it supports the options that these fields control.

The power-down modes and their wake-up methods are programmed by the PWRD field (bits 15−10) of CSR. The PWRD field of CSR is shown in Figure 2−5. When writing to CSR, all bits of the PWRD field should be configured at the same time. A logic 0 should be used when writing to the reserved bit (bit 15) of the PWRD field.

Figure 2−4. Control Status Register (CSR)

31–24	23–16
CPU ID	REVISION ID
R-0	R-x†

15–10	9	8	7–5	4–2	1	0
PWRD	SAT	EN	PCC	DCC	PGIE	GIE
R/W-0	R/WC-0	R-x	R/W-0	R/W-0	R/W-0	R/W-0

Legend: R = Readable by the **MVC** instruction; W = Writeable by the **MVC** instruction; WC = Bit is cleared on write; -*n* = value after reset; -x = value is indeterminate after reset

† See the device-specific data manual for the default value of this field.

Figure 2−5. PWRD Field of Control Status Register (CSR)

15	14	13	12	11	10
Reserved	Enabled or nonenabled interrupt wake	Enabled interrupt wake	PD3	PD2	PD1
R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0

Legend: R = Readable by the **MVC** instruction; W = Writeable by the **MVC** instruction; -*n* = value after reset

Table 2−7. Control Status Register (CSR) Field Descriptions

Bit	Field	Value	Description
31−24	CPU ID	0−FFh	Identifies the CPU of the device. Not writable by the **MVC** instruction.
		0−1h	Reserved
		2h	C67x CPU
		3h	C67x+ CPU
		4h−FFh	Reserved
23−16	REVISION ID	0−FFh	Identifies silicon revision of the CPU. For the most current silicon revision information, see the device-specific data manual. Not writable by the **MVC** instruction.
15−10	PWRD	0−3Fh	Power-down mode field. See Figure 2−5. Writable by the **MVC** instruction.
		0	No power-down.
		1h−8h	Reserved
		9h	Power-down mode PD1; wake by an enabled interrupt.
		Ah−10h	Reserved
		11h	Power-down mode PD1; wake by an enabled or nonenabled interrupt.
		12h−19h	Reserved
		1Ah	Power-down mode PD2; wake by a device reset.
		1Bh	Reserved
		1Ch	Power-down mode PD3; wake by a device reset.
		1D−3Fh	Reserved
9	SAT		Saturate bit. Can be cleared only by the **MVC** instruction and can be set only by a functional unit. The set by a functional unit has priority over a clear (by the **MVC** instruction), if they occur on the same cycle. The SAT bit is set one full cycle (one delay slot) after a saturate occurs. The SAT bit will not be modified by a conditional instruction whose condition is false.
		0	Any unit does not perform a saturate.
		1	Any unit performs a saturate.
8	EN		Endian mode. Not writable by the **MVC** instruction.
		0	Big endian
		1	Little endian

Table 2−7. Control Status Register (CSR) Field Descriptions (Continued)

Bit	Field	Value	Description
7−5	PCC	0−7h	Program cache control mode. Writable by the **MVC** instruction. See the *TMS320C621x/C671x DSP Two-Level Internal Memory Reference Guide* (SPRU609).
		0	Direct-mapped cache enabled
		1h	Reserved
		2h	Direct-mapped cache enabled
		3h−7h	Reserved
4−2	DCC	0−7h	Data cache control mode. Writable by the **MVC** instruction. See the *TMS320C621x/C671x DSP Two-Level Internal Memory Reference Guide* (SPRU609).
		0	2-way cache enabled
		1h	Reserved
		2h	2-way cache enabled
		3h−7h	Reserved
1	PGIE		Previous GIE (global interrupt enable). Copy of GIE bit at point when interrupt is taken. Physically the same bit as SGIE bit in the interrupt task state register (ITSR). Writeable by the **MVC** instruction.
		0	Disables saving GIE bit when an interrupt is taken.
		1	Enables saving GIE bit when an interrupt is taken.
0	GIE		Global interrupt enable. Physically the same bit as GIE bit in the task state register (TSR). Writable by the **MVC** instruction.
		0	Disables all interrupts, except the reset interrupt and NMI (nonmaskable interrupt).
		1	Enables all interrupts.

2.7.5 Interrupt Clear Register (ICR)

The interrupt clear register (ICR) allows you to manually clear the maskable interrupts (INT15−INT4) in the interrupt flag register (IFR). Writing a 1 to any of the bits in ICR causes the corresponding interrupt flag (IF*n*) to be cleared in IFR. Writing a 0 to any bit in ICR has no effect. Incoming interrupts have priority and override any write to ICR. You cannot set any bit in ICR to affect NMI or reset. The ISR is shown in Figure 2−6 and described in Table 2−8.

Note:

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

Any write to ICR is ignored by a simultaneous write to the same bit in the interrupt set register (ISR).

Figure 2−6. Interrupt Clear Register (ICR)

31–16
Reserved
R-0

15	14	13	12	11	10	9	8	7	6	5	4	3–0
IC15	IC14	IC13	IC12	IC11	IC10	IC9	IC8	IC7	IC6	IC5	IC4	Reserved
W-0												R-0

Legend: R = Read only; W = Writeable by the **MVC** instruction; -*n* = value after reset

Table 2−8. Interrupt Clear Register (ICR) Field Descriptions

Bit	Field	Value	Description
31−16	Reserved	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.
15−4	IC*n*		Interrupt clear.
		0	Corresponding interrupt flag (IF*n*) in IFR is not cleared.
		1	Corresponding interrupt flag (IF*n*) in IFR is cleared.
3−0	Reserved	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.

2.7.6 Interrupt Enable Register (IER)

The interrupt enable register (IER) enables and disables individual interrupts. The IER is shown in Figure 2−7 and described in Table 2−9.

Figure 2−7. Interrupt Enable Register (IER)

31–16
Reserved
R-0

15	14	13	12	11	10	9	8	7	6	5	4	3–2	1	0
IE15	IE14	IE13	IE12	IE11	IE10	IE9	IE8	IE7	IE6	IE5	IE4	Reserved	NMIE	1
R/W-0												R-0	R/W-0	R-1

Legend: R = Readable by the **MVC** instruction; W = Writeable by the **MVC** instruction; -*n* = value after reset

Table 2−9. Interrupt Enable Register (IER) Field Descriptions

Bit	Field	Value	Description
31−16	Reserved	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.
15−4	IE*n*		Interrupt enable. An interrupt triggers interrupt processing only if the corresponding bit is set to 1.
		0	Interrupt is disabled.
		1	Interrupt is enabled.
3−2	Reserved	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.
1	NMIE		Nonmaskable interrupt enable. An interrupt triggers interrupt processing only if the bit is set to 1. The NMIE bit is cleared at reset. After reset, you must set the NMIE bit to enable the NMI and to allow INT15−INT4 to be enabled by the GIE bit in CSR and the corresponding IER bit. You cannot manually clear the NMIE bit; a write of 0 has no effect. The NMIE bit is also cleared by the occurrence of an NMI.
		0	All nonreset interrupts are disabled.
		1	All nonreset interrupts are enabled. The NMIE bit is set only by completing a **B NRP** instruction or by a write of 1 to the NMIE bit.
0	1	1	Reset interrupt enable. You cannot disable the reset interrupt.

2.7.7 Interrupt Flag Register (IFR)

The interrupt flag register (IFR) contains the status of INT4−INT15 and NMI interrupt. Each corresponding bit in the IFR is set to 1 when that interrupt occurs; otherwise, the bits are cleared to 0. If you want to check the status of interrupts, use the **MVC** instruction to read the IFR. (See the **MVC** instruction description, page 3-179, for information on how to use this instruction.) The IFR is shown in Figure 2−8 and described in Table 2−10.

Figure 2−8. Interrupt Flag Register (IFR)

31–16
Reserved
R-0

15	14	13	12	11	10	9	8	7	6	5	4	3–2	1	0
IF15	IF14	IF13	IF12	IF11	IF10	IF9	IF8	IF7	IF6	IF5	IF4	Reserved	NMIF	0
R-0												R-0	R-0	R-0

Legend: R = Readable by the **MVC** instruction; -*n* = value after reset

Table 2−10. Interrupt Flag Register (IFR) Field Descriptions

Bit	Field	Value	Description
31−16	Reserved	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.
15−4	IF*n*		Interrupt flag. Indicates the status of the corresponding maskable interrupt. An interrupt flag may be manually set by setting the corresponding bit (IS*n*) in the interrupt set register (ISR) or manually cleared by setting the corresponding bit (IC*n*) in the interrupt clear register (ICR).
		0	Interrupt has not occurred.
		1	Interrupt has occurred.
3−2	Reserved	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.
1	NMIF		Nonmaskable interrupt flag.
		0	Interrupt has not occurred.
		1	Interrupt has occurred.
0	0	0	Reset interrupt flag.

2.7.8 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 IRP (**B IRP**) in your interrupt service routine returns to the program flow when interrupt servicing is complete. The IRP is shown in Figure 2−9.

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 IRP, any subsequent interrupt processing may overwrite that value.

Figure 2−9. Interrupt Return Pointer Register (IRP)

31 ... 0
IRP
R/W-x

Legend: R = Readable by the **MVC** instruction; W = Writeable by the **MVC** instruction; -x = value is indeterminate after reset

2.7.9 Interrupt Set Register (ISR)

The interrupt set register (ISR) allows you to manually set the maskable interrupts (INT15−INT4) in the interrupt flag register (IFR). Writing a 1 to any of the bits in ISR causes the corresponding interrupt flag (IF*n*) to be set in IFR. Writing a 0 to any bit in ISR has no effect. You cannot set any bit in ISR to affect NMI or reset. The ISR is shown in Figure 2−10 and described in Table 2−11.

Note:

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

Any write to the interrupt clear register (ICR) is ignored by a simultaneous write to the same bit in ISR.

Figure 2−10. Interrupt Set Register (ISR)

31−16
Reserved
R-0

15	14	13	12	11	10	9	8	7	6	5	4	3−0
IS15	IS14	IS13	IS12	IS11	IS10	IS9	IS8	IS7	IS6	IS5	IS4	Reserved
W-0												R-0

Legend: R = Read only; W = Writeable by the **MVC** instruction; -*n* = value after reset

Table 2−11. Interrupt Set Register (ISR) Field Descriptions

Bit	Field	Value	Description
31−16	Reserved	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.
15−4	IS*n*		Interrupt set.
		0	Corresponding interrupt flag (IF*n*) in IFR is not set.
		1	Corresponding interrupt flag (IF*n*) in IFR is set.
3−0	Reserved	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.

2.7.10 Interrupt Service Table Pointer Register (ISTP)

The interrupt service table pointer register (ISTP) is used to locate the interrupt service routine (ISR). The ISTB field identifies the base portion of the address of the interrupt service table (IST) and the HPEINT field identifies the specific interrupt and locates the specific fetch packet within the IST. The ISTP is shown in Figure 2−11 and described in Table 2−12. See section 5.1.2.2 on page 5-9 for a discussion of the use of the ISTP.

Figure 2−11. Interrupt Service Table Pointer Register (ISTP)

31–16
ISTB
R/W-0

15–10	9–5	4	3	2	1	0
ISTB	HPEINT	0	0	0	0	0
R/W-0	R-0	R-0				

Legend: R = Readable by the **MVC** instruction; W = Writeable by the **MVC** instruction; -*n* = value after reset

Table 2−12. Interrupt Service Table Pointer Register (ISTP) Field Descriptions

Bit	Field	Value	Description
31−10	ISTB	0−3F FFFFh	Interrupt service table base portion of the IST address. This field is cleared to 0 on reset; therefore, 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 clears the ISTB to 0. See Example 5−1.
9−5	HPEINT	0−1Fh	Highest priority enabled interrupt that is currently pending. This field indicates the number (related bit position in the IFR) of the highest priority interrupt (as defined in Table 5−1 on page 5-3) 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 0. The corresponding interrupt need not be enabled by NMIE (unless it is NMI) or by GIE.
4−0	−	0	Cleared to 0 (fetch packets must be aligned on 8-word (32-byte) boundaries).

2.7.11 Nonmaskable Interrupt (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 NRP (**B NRP**) in your interrupt service routine returns to the program flow when NMI servicing is complete. The NRP is shown in Figure 2−12.

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 NRP, any subsequent interrupt processing may overwrite that value.

Figure 2−12. NMI Return Pointer Register (NRP)

31 … 0
NRP
R/W-x

Legend: R = Readable by the **MVC** instruction; W = Writeable by the **MVC** instruction; -x = value is indeterminate after reset

2.7.12 E1 Phase Program Counter (PCE1)

The E1 phase program counter (PCE1), shown in Figure 2−13, contains the 32-bit address of the fetch packet in the E1 pipeline phase.

Figure 2−13. E1 Phase Program Counter (PCE1)

31 … 0
PCE1
R-x

Legend: R = Readable by the **MVC** instruction; -x = value is indeterminate after reset

2.8 Control Register File Extensions

The C67x DSP 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 2−13 lists the additional registers used. 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 2−13. Control Register File Extensions

Acronym	Register Name	Section
FADCR	Floating-point adder configuration register	2.8.1
FAUCR	Floating-point auxiliary configuration register	2.8.2
FMCR	Floating-point multiplier configuration register	2.8.3

2.8.1 Floating-Point Adder Configuration Register (FADCR)

The floating-point adder 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: .L2 uses bits 31−16 and .L1 uses bits 15−0. FADCR is shown in Figure 2−14 and described in Table 2−14.

Note:

For the C67x+ DSP, the **ADDSP**, **ADDDP**, **SUBSP**, and **SUBDP** instructions executing in the .S functional unit use the rounding mode from and set the warning bits in FADCR. The warning bits in FADCR are the logical-OR of the warnings produced on the .L functional unit and the warnings produced by the ADDSP/ADDDP/SUBSP/SUBDP instructions on the .S functional unit (but not other instructions executing on the .S functional unit).

Figure 2−14. Floating-Point Adder Configuration Register (FADCR)

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	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0

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	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0

Legend: R = Readable by the **MVC** instruction; W = Writeable by the **MVC** instruction; -*n* = value after reset

Table 2−14. Floating-Point Adder Configuration Register (FADCR) Field Descriptions

Bit	Field	Value	Description
31−27	Reserved	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.
26−25	RMODE	0−3h	Rounding mode select for .L2.
		0	Round toward nearest representable floating-point number
		1h	Round toward 0 (truncate)
		2h	Round toward infinity (round up)
		3h	Round toward negative infinity (round down)
24	UNDER		Result underflow status for .L2.
		0	Result does not underflow.
		1	Result underflows.
23	INEX		Inexact results status for .L2.
		0	
		1	Result differs from what would have been computed had the exponent range and precision been unbounded; never set with INVAL.
22	OVER		Result overflow status for .L2.
		0	Result does not overflow.
		1	Result overflows.
21	INFO		Signed infinity for .L2.
		0	Result is not signed infinity.
		1	Result is signed infinity.

Table 2−14. Floating-Point Adder Configuration Register (FADCR) Field Descriptions (Continued)

Bit	Field	Value	Description
20	INVAL		
		0	A signed NaN (SNaN) is not a source.
		1	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	DEN2		Denormalized number select for .L2 *src2*.
		0	*src2* is not a denormalized number.
		1	*src2* is a denormalized number.
18	DEN1		Denormalized number select for .L2 *src1*.
		0	*src1* is not a denormalized number.
		1	*src1* is a denormalized number.
17	NAN2		NaN select for .L2 *src2*.
		0	*src2* is not NaN.
		1	*src2* is NaN.
16	NAN1		NaN select for .L2 *src1*.
		0	*src1* is not NaN.
		1	*src1* is NaN.
15−11	Reserved	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.
10−9	RMODE	0−3h	Rounding mode select for .L1.
		0	Round toward nearest representable floating-point number
		1h	Round toward 0 (truncate)
		2h	Round toward infinity (round up)
		3h	Round toward negative infinity (round down)
8	UNDER		Result underflow status for .L1.
		0	Result does not underflow.
		1	Result underflows.

Table 2−14. Floating-Point Adder Configuration Register (FADCR) Field Descriptions (Continued)

Bit	Field	Value	Description
7	INEX		Inexact results status for .L1.
		0	
		1	Result differs from what would have been computed had the exponent range and precision been unbounded; never set with INVAL.
6	OVER		Result overflow status for .L1.
		0	Result does not overflow.
		1	Result overflows.
5	INFO		Signed infinity for .L1.
		0	Result is not signed infinity.
		1	Result is signed infinity.
4	INVAL		
		0	A signed NaN (SNaN) is not a source.
		1	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.
3	DEN2		Denormalized number select for .L1 *src2*.
		0	*src2* is not a denormalized number.
		1	*src2* is a denormalized number.
2	DEN1		Denormalized number select for .L1 *src1*.
		0	*src1* is not a denormalized number.
		1	*src1* is a denormalized number.
1	NAN2		NaN select for .L1 *src2*.
		0	*src2* is not NaN.
		1	*src2* is NaN.
0	NAN1		NaN select for .L1 *src1*.
		0	*src1* is not NaN.
		1	*src1* is NaN.

2.8.2 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: .S2 uses bits 31−16 and .S1 uses bits 15−0. FAUCR is shown in Figure 2−15 and described in Table 2−15.

Note:

For the C67x+ DSP, the **ADDSP**, **ADDDP**, **SUBSP**, and **SUBDP** instructions executing in the .S functional unit use the rounding mode from and set the warning bits in the floating-point adder configuration register (FADCR). The warning bits in FADCR are the logical-OR of the warnings produced on the .L functional unit and the warnings produced by the ADDSP/ADDDP/SUBSP/SUBDP instructions on the .S functional unit (but not other instructions executing on the .S functional unit).

Figure 2−15. Floating-Point Auxiliary Configuration Register (FAUCR)

31–27	26	25	24	23	22	21	20	19	18	17	16
Reserved	DIV0	UNORD	UND	INEX	OVER	INFO	INVAL	DEN2	DEN1	NAN2	NAN1
R-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0

15–11	10	9	8	7	6	5	4	3	2	1	0
Reserved	DIV0	UNORD	UND	INEX	OVER	INFO	INVAL	DEN2	DEN1	NAN2	NAN1
R-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0

Legend: R = Readable by the **MVC** instruction; W = Writeable by the **MVC** instruction; -*n* = value after reset

Table 2−15. Floating-Point Auxiliary Configuration Register (FAUCR) Field Descriptions

Bit	Field	Value	Description
31−27	Reserved	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.
26	DIV0		Source to reciprocal operation for .S2.
		0	0 is not source to reciprocal operation.
		1	0 is source to reciprocal operation.

Table 2−15. Floating-Point Auxiliary Configuration Register (FAUCR) Field Descriptions (Continued)

Bit	Field	Value	Description
25	UNORD		Source to a compare operation for .S2
		0	NaN is not a source to a compare operation.
		1	NaN is a source to a compare operation.
24	UND		Result underflow status for .S2.
		0	Result does not underflow.
		1	Result underflows.
23	INEX		Inexact results status for .S2.
		0	
		1	Result differs from what would have been computed had the exponent range and precision been unbounded; never set with INVAL.
22	OVER		Result overflow status for .S2.
		0	Result does not overflow.
		1	Result overflows.
21	INFO		Signed infinity for .S2.
		0	Result is not signed infinity.
		1	Result is signed infinity.
20	INVAL		
		0	A signed NaN (SNaN) is not a source.
		1	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	DEN2		Denormalized number select for .S2 *src2*.
		0	*src2* is not a denormalized number.
		1	*src2* is a denormalized number.
18	DEN1		Denormalized number select for .S2 *src1*.
		0	*src1* is not a denormalized number.
		1	*src1* is a denormalized number.

Table 2−15. Floating-Point Auxiliary Configuration Register (FAUCR) Field Descriptions (Continued)

Bit	Field	Value	Description
17	NAN2		NaN select for .S2 *src2*.
		0	*src2* is not NaN.
		1	*src2* is NaN.
16	NAN1		NaN select for .S2 *src1*.
		0	*src1* is not NaN.
		1	*src1* is NaN.
15−11	Reserved	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.
10	DIV0		Source to reciprocal operation for .S1.
		0	0 is not source to reciprocal operation.
		1	0 is source to reciprocal operation.
9	UNORD		Source to a compare operation for .S1
		0	NaN is not a source to a compare operation.
		1	NaN is a source to a compare operation.
8	UND		Result underflow status for .S1.
		0	Result does not underflow.
		1	Result underflows.
7	INEX		Inexact results status for .S1.
		0	
		1	Result differs from what would have been computed had the exponent range and precision been unbounded; never set with INVAL.
6	OVER		Result overflow status for .S1.
		0	Result does not overflow.
		1	Result overflows.

Table 2−15. Floating-Point Auxiliary Configuration Register (FAUCR) Field Descriptions (Continued)

Bit	Field	Value	Description
5	INFO		Signed infinity for .S1.
		0	Result is not signed infinity.
		1	Result is signed infinity.
4	INVAL		
		0	A signed NaN (SNaN) is not a source.
		1	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.
3	DEN2		Denormalized number select for .S1 *src2*.
		0	*src2* is not a denormalized number.
		1	*src2* is a denormalized number.
2	DEN1		Denormalized number select for .S1 *src1*.
		0	*src1* is not a denormalized number.
		1	*src1* is a denormalized number.
1	NAN2		NaN select for .S1 *src2*.
		0	*src2* is not NaN.
		1	*src2* is NaN.
0	NAN1		NaN select for .S1 *src1*.
		0	*src1* is not NaN.
		1	*src1* is NaN.

2.8.3 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: .M2 uses bits 31−16 and .M1 uses bits 15−0. FMCR is shown in Figure 2−16 and described in Table 2−16.

Figure 2−16. Floating-Point Multiplier Configuration Register (FMCR)

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	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0

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	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0

Legend: R = Readable by the **MVC** instruction; W = Writeable by the **MVC** instruction; -*n* = value after reset

Table 2−16. Floating-Point Multiplier Configuration Register (FMCR) Field Descriptions

Bit	Field	Value	Description
31−27	Reserved	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.
26−25	RMODE	0−3h	Rounding mode select for .M2.
		0	Round toward nearest representable floating-point number
		1h	Round toward 0 (truncate)
		2h	Round toward infinity (round up)
		3h	Round toward negative infinity (round down)
24	UNDER		Result underflow status for .M2.
		0	Result does not underflow.
		1	Result underflows.

Table 2−16. Floating-Point Multiplier Configuration Register (FMCR) Field Descriptions (Continued)

Bit	Field	Value	Description
23	INEX		Inexact results status for .M2.
		0	
		1	Result differs from what would have been computed had the exponent range and precision been unbounded; never set with INVAL.
22	OVER		Result overflow status for .M2.
		0	Result does not overflow.
		1	Result overflows.
21	INFO		Signed infinity for .M2.
		0	Result is not signed infinity.
		1	Result is signed infinity.
20	INVAL		
		0	A signed NaN (SNaN) is not a source.
		1	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	DEN2		Denormalized number select for .M2 *src2*.
		0	*src2* is not a denormalized number.
		1	*src2* is a denormalized number.
18	DEN1		Denormalized number select for .M2 *src1*.
		0	*src1* is not a denormalized number.
		1	*src1* is a denormalized number.
17	NAN2		NaN select for .M2 *src2*.
		0	*src2* is not NaN.
		1	*src2* is NaN.
16	NAN1		NaN select for .M2 *src1*.
		0	*src1* is not NaN.
		1	*src1* is NaN.

Table 2−16. Floating-Point Multiplier Configuration Register (FMCR) Field Descriptions (Continued)

Bit	Field	Value	Description
15−11	Reserved	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.
10−9	RMODE	0−3h	Rounding mode select for .M1.
		0	Round toward nearest representable floating-point number
		1h	Round toward 0 (truncate)
		2h	Round toward infinity (round up)
		3h	Round toward negative infinity (round down)
8	UNDER		Result underflow status for .M1.
		0	Result does not underflow.
		1	Result underflows.
7	INEX		Inexact results status for .M1.
		0	
		1	Result differs from what would have been computed had the exponent range and precision been unbounded; never set with INVAL.
6	OVER		Result overflow status for .M1.
		0	Result does not overflow.
		1	Result overflows.
5	INFO		Signed infinity for .M1.
		0	Result is not signed infinity.
		1	Result is signed infinity.
4	INVAL		
		0	A signed NaN (SNaN) is not a source.
		1	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.
3	DEN2		Denormalized number select for .M1 *src2*.
		0	*src2* is not a denormalized number.
		1	*src2* is a denormalized number.

Table 2−16. Floating-Point Multiplier Configuration Register (FMCR) Field Descriptions (Continued)

Bit	Field	Value	Description
2	DEN1		Denormalized number select for .M1 *src1*.
		0	*src1* is not a denormalized number.
		1	*src1* is a denormalized number.
1	NAN2		NaN select for .M1 *src2*.
		0	*src2* is not NaN.
		1	*src2* is NaN.
0	NAN1		NaN select for .M1 *src1*.
		0	*src1* is not NaN.
		1	*src1* is NaN.

Chapter 3

Instruction Set

This chapter describes the assembly language instructions of the TMS320C67x DSP. Also described are parallel operations, conditional operations, resource constraints, and addressing modes.

The C67x floating-point DSP uses all of the instructions available to the TMS320C62x™ DSP but it also uses other instructions that are specific to the C67x DSP. These specific instructions are for 32-bit integer multiply, double-word load, and floating-point operations, including addition, subtraction, and multiplication.

Topic **Page**

3.1 Instruction Operation and Execution Notations

Table 3−1 explains the symbols used in the instruction descriptions.

Table 3−1. 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_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-significant byte position in 32-bit register (bits 16-23)
byte3	8-bit value in the most-significant 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
$b_{y..z}$	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, see section 3.6
cstn	n-bit constant field (for example, cst5)
dint	64-bit integer value (two registers)
dp	Double-precision floating-point register value
dp(x)	Convert x to dp
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

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

Symbol	Meaning
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
int(x)	Convert x to integer
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
op	Opfields
or	Bitwise OR
R	Any general-purpose register
rcp(x)	Reciprocal approximation of x
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	n-bit signed constant field
sdint	Signed 64-bit integer value (two registers)
se	Sign-extend

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

Symbol	Meaning
sint	Signed 32-bit integer value
slong	Signed 40-bit integer value
sllong	Signed 64-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
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
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
−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
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, ucst5)
uint	Unsigned 32-bit integer value
ulong	Unsigned 40-bit integer value
ullong	Unsigned 64-bit integer value
ulsb16	Unsigned 16-bit integer value in lower half of 32-bit register

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

Symbol	Meaning
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
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)
xint	32-bit integer value that can optionally use cross path
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
xsp	Single-precision floating-point register 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
xulsb16	Unsigned 16 LSB of register that can optionally use cross path
xumsb16	Unsigned 16 MSB of register 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
$\rightarrow$	Assignment
+	Addition
++	Increment by 1
$\times$	Multiplication
−	Subtraction
==	Equal to

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

Symbol	Meaning
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to
<<	Shift left
>>	Shift right
>>s	Shift right with sign extension
>>z	Shift right with a zero fill
~	Logical inverse
&	Logical AND

3.2 Instruction Syntax and Opcode Notations

Table 3−2 explains the syntaxes and opcode fields used in the instruction descriptions.

The C64x CPU 32-bit opcodes are mapped in Appendix C through Appendix G.

Table 3−2. Instruction Syntax and Opcode Notations

Symbol	Meaning
baseR	base address register
CC	
creg	3-bit field specifying a conditional register, see section 3.6
cst	constant
csta	constant a
cstb	constant b
cstn	n-bit constant field
dst	destination
dstms	
dw	doubleword; 0 = word, 1 = doubleword
ii_n	bit n of the constant *ii*
ld/st	load or store; 0 = store, 1 = load
mode	addressing mode, see section 3.8
offsetR	register offset
op	opfield; field within opcode that specifies a unique instruction
op_n	bit n of the opfield
p	parallel execution; 0 = next instruction is not executed in parallel, 1 = next instruction is executed in parallel
r	LDDW instruction
rsv	reserved
s	side A or B for destination; 0 = side A, 1 = side B.
sc	scaling mode; 0 = nonscaled, *offsetR/ucst5* is not shifted; 1 = scaled, *offsetR/ucst5* is shifted
scstn	n-bit signed constant field

Table 3−2. Instruction Syntax and Opcode Notations (Continued)

Symbol	Meaning
$scst_n$	bit n of the signed constant field
sn	sign
src	source
src1	source 1
src2	source 2
srcms	
stg_n	bit n of the constant *stg*
t	side of source/destination (*src/dst*) register; 0 = side A, 1 = side B
ucstn	n-bit unsigned constant field
$ucst_n$	bit n of the unsigned constant field
unit	unit decode
x	cross path for *src2*; 0 = do not use cross path, 1 = use cross path
y	.D1 or .D2 unit; 0 = .D1 unit, 1 = .D2 unit
z	test for equality with zero or nonzero

3.3 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 32 least-significant-bits are loaded into the even register; the 32 most-significant-bits containing the sign bit and exponent are loaded into the next register (that 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 3−3 shows notations used in discussing floating-point numbers.

Table 3−3. 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 3−1 shows the fields of a single-precision floating-point number represented within a 32-bit register.

Figure 3−1. Single-Precision Floating-Point Fields

31	30 … 23	22 … 0
s	e	f

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.

Normalized:

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

Denormalized (Subnormal):

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

Table 3−4 shows the s,e, and f values for special single-precision floating-point numbers.

Table 3−4. 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

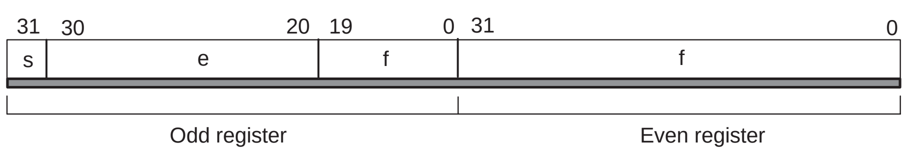
Table 3−5 shows hexadecimal and decimal values for some single-precision floating-point numbers.

Figure 3−2 shows the fields of a double-precision floating-point number represented within a pair of 32-bit registers.

Table 3−5. Hexadecimal and Decimal Representation for Selected Single-Precision Values

Symbol	**Hex Value**	**Decimal Value**
NaN_out	7FFF FFFF	QNaN
0	0000 0000	0.0
−0	8000 0000	−0.0
1	3F80 0000	1.0
2	4000 0000	2.0
LFPN	7F7F FFFF	3.40282347e+38
SFPN	0080 0000	1.17549435e−38
LDFPN	007F FFFF	1.17549421e−38
SDFPN	0000 0001	1.40129846e−45

Figure 3−2. 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.

Normalized:

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

Denormalized (Subnormal):

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

Table 3−6 shows the s, e, and f values for special double-precision floating-point numbers.

Table 3−6. 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 3−7 shows hexadecimal and decimal values for some double-precision floating-point numbers.

Table 3−7. Hexadecimal and Decimal Representation for Selected Double-Precision Values

Symbol	Hex Value	Decimal Value
NaN_out	7FFF FFFF FFFF FFFF	QNaN
0	0000 0000 0000 0000	0.0
−0	8000 0000 0000 0000	−0.0
1	3FF0 0000 0000 0000	1.0
2	4000 0000 0000 0000	2.0
LFPN	7FEF FFFF FFFF FFFF	1.7976931348623157e+308
SFPN	0010 0000 0000 0000	2.2250738585072014e−308
LDFPN	000F FFFF FFFF FFFF	2.2250738585072009e−308
SDFPN	0000 0000 0000 0001	4.9406564584124654e−324

3.4 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 3−8 shows the number of delay slots associated with each type of instruction.

The functional unit latency is equivalent to the number of cycles that must pass before the functional unit can start executing the next instruction. The double-precision floating-point addition, subtraction, multiplication, compare, and the 32-bit integer multiply instructions have a functional unit latency that is greater than 1. Most instructions have a functional unit latency of 1, meaning that the next instruction can begin execution in cycle *i* + 1. 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.

Table 3−8. Delay Slot and Functional Unit Latency

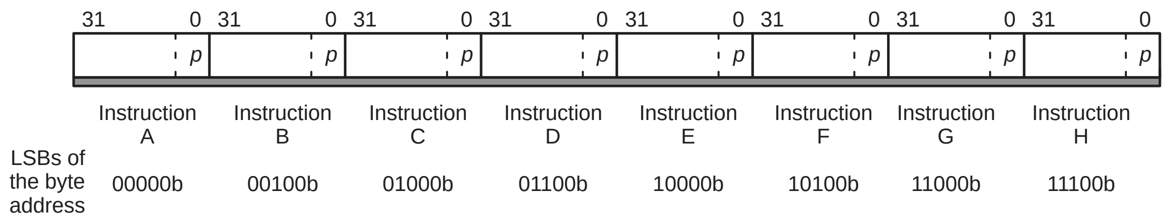
Instruction Type	Delay Slots	Functional Unit Latency	Read Cycles†	Write Cycles†
Single cycle	0	1	i	i
2-cycle DP	1	1	i	i, i + 1
DP compare	1	2	i, i + 1	1 + 1
4-cycle	3	1	i	i + 3
INTDP	4	1	i	i + 3, i + 4
Load	4	1	i	i, i + 4‡
MPYSP2DP	4	2	i	i + 3, i + 4
ADDDP/SUBDP	6	2	i, i + 1	i + 5, i + 6
MPYSPDP	6	3	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 .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 3−3. Fetch packets are aligned on 256-bit (8-word) boundaries.

Figure 3−3. 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.

On the C67x DSP, an execute packet cannot cross an 8-word boundary; therefore, the last *p*-bit in a fetch packet is always cleared to 0, and each fetch packet starts a new execute packet. On the C67x+ DSP, an execute packet can cross an 8-word boundary.

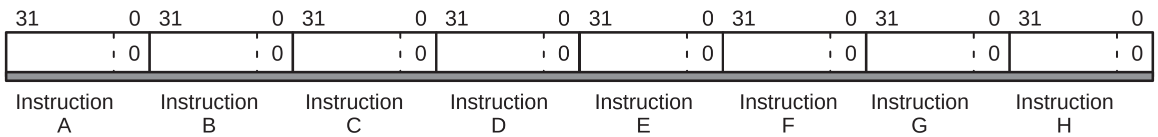
There are three types of *p*-bit patterns for fetch packets. These three *p*-bit patterns result in the following execution sequences for the eight instructions:

- Fully serial
- Fully parallel
- Partially serial

Example 3−1 through Example 3−3 show 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:



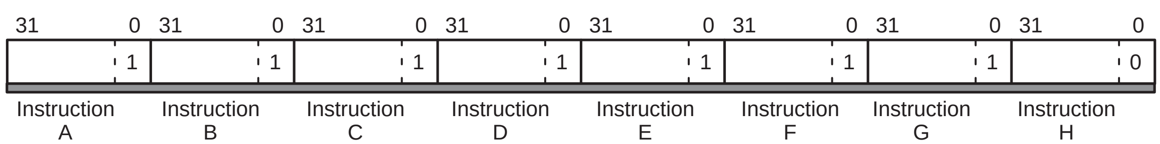
results in this execution sequence:

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:



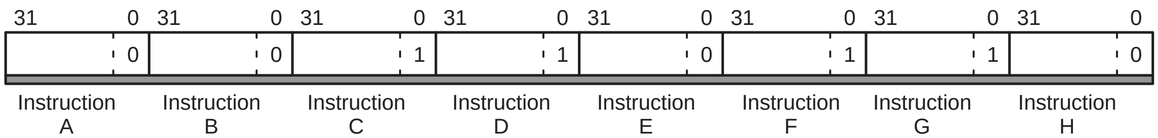
results in this execution sequence:

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 vertical bars || 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

Most 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 4. 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 3−9.

Table 3−9. 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‡

† This value is reserved for software breakpoints that are used for emulation purposes.
‡ x can be any value.

Conditional instructions are represented in code by using square brackets, [ ], surrounding the condition register name. The following execute packet contains two **ADD** instructions in parallel. The first **ADD** is conditional on B0 being nonzero. The second **ADD** is conditional on B0 being zero. The character ! indicates the inverse of the condition.

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

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

3.7 Resource Constraints

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

3.7.1 Constraints on Instructions Using the Same Functional Unit

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

The following execute packet is invalid:

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

The following execute packet is valid:

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

3.7.2 Constraints on the Same Functional Unit Writing in the Same Instruction Cycle

Two instructions using the same functional unit cannot write their results in the same instruction cycle.

3.7.3 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, 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 ;
```

The operand comes from a register file opposite of the destination, if the x bit in the instruction field is set.

3.7.4 Constraints on Loads and Stores

Load and store instructions can use an address pointer from one register file while loading to or storing from the other register file. Two load and 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
```

3.7.5 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 2.2 for the order for long pairs.

The following execute packet is invalid:

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

The following execute packet is valid:

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

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

The following execute packet is invalid:

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

The following execute packet is valid:

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

On the C67x DSP, doubleword load instructions conflict with long results from the .S units. All stores conflict with a long source on the .S unit. The following execute packet is invalid, 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 store
|| STW .D1T1 A8,*A9            ; / on the T1 path of the .D unit
```

The following code sequence is invalid:

```
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 execute packets are valid:

```
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
```

3.7.6 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 execute packets 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
```

The following execute packet is valid:

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

3.7.7 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, an **MPY** issued on cycle *i* followed by an **ADD** on cycle *i* + 1 cannot write to the same register because both instructions write a result on cycle *i* + 1. Therefore, the following code sequence is invalid 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 3−4 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 3−4. 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 ; /
```

3.7.8 Constraints on Floating-Point Instructions

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.
MPYSPDP	No other instruction can use the functional unit on cycles i and i + 1.
MPYSP2DP	No other instruction can use the functional unit on cycles i and i + 1.

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.
MPYSPDP	No other instruction on the same side can use the cross path on cycles i and i + 1.

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.
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.
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 MPYSPDP instruction cannot be scheduled on that functional unit on cycle i + 4, i + 5, or i + 6. A MPYSP2DP instruction cannot be scheduled on that functional unit on cycle i + 4, i + 5, or i + 6. A multiply (16 $\times$ 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 cycle i + 4, i + 5, or i + 6. A MPYSPDP instruction cannot be scheduled on that functional unit on cycle i + 4, i + 5, or i + 6. A MPYSP2DP instruction cannot be scheduled on that functional unit on cycle i + 4, i + 5, or i + 6. A multiply (16 $\times$ 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 $\times$ 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.

MPYSPDP	A 4-cycle instruction cannot be scheduled on that functional unit on cycle i + 2 or i + 3.
	A MPYI instruction cannot be scheduled on that functional unit on cycle i + 2 or i + 3.
	A MPYID instruction cannot be scheduled on that functional unit on cycle i + 2 or i + 3.
	A MPYDP instruction cannot be scheduled on that functional unit on cycle i + 2 or i + 3.
	A MPYSP2DP instruction cannot be scheduled on that functional unit on cycle i + 2 or i + 3.
	A multiply (16 $\times$ 16-bit) instruction cannot be scheduled on that functional unit on cycle i + 4 or i + 5 due to a write hazard on cycle i + 5 or i + 6, respectively.
MPYSP2DP	A multiply (16 $\times$ 16-bit) instruction cannot be scheduled on that functional unit on cycle i + 2 or i + 3 due to a write hazard on cycle i + 3 or i + 4, 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.

3.8 Addressing Modes

The addressing modes on the C67x DSP are linear, circular using BK0, and circular using BK1. The addressing mode is specified by the addressing mode register (AMR), described in section 2.7.3.

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 AMR to determine what type of address calculations are performed for these registers.

3.8.1 Linear Addressing Mode

3.8.1.1 LD and 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 doubleword, word, halfword, or byte access, respectively; and then performs an add or a subtract to *baseR* (depending on the operation specified).

For the preincrement, predecrement, positive offset, and negative offset address generation 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 from memory.

3.8.1.2 ADDA and 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 doubleword, 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 AMR specify the block sizes for circular addressing, see section 2.7.3.

3.8.2.1 LD and 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 in AMR is not scaled; for example, a block-size of 8 is 8 bytes, not 8 times the data size (byte, halfword, word). So, to perform circular addressing on an array of 8 words, a size of 32 should be specified, or N = 4. Example 3−4 shows an **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 in AMR for this example is 0004 0001h.

Example 3−4. LDW Instruction 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 and 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 AMR is not scaled; for example, a block size of 8 is 8 bytes, not 8 times the 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 in AMR for this example is 0004 0001h.

Example 3−5. ADDAH Instruction 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 C64x DSP has a load/store architecture, which means that the only way to access data in memory is with a load or store instruction. Table 3−10 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 3−11 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*).

Table 3−10. 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*]

Table 3−11. Address Generator Options for Load/Store

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

3.9 Instruction Compatibility

The C62x, C64x, and C67x DSPs share an instruction set. All of the instructions valid for the C62x DSP are also valid for the C67x DSP. See Appendix A for a list of the instructions that are common to the C62x, C64x, and C67x DSPs.

3.10 Instruction Descriptions

This section gives detailed information on the instruction set. Each instruction may present the following information:

- Assembler syntax
- Functional units
- Compatibility
- Operands
- Opcode
- Description
- Execution
- Pipeline
- 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 shows the various fields that make up each instruction. These fields are described in Table 3−2 on page 3-7.

There are instructions that can be executed on more than one functional unit. Table 3−12 shows how this is documented for the **ADD** instruction. This instruction has three opcode map fields: *src1*, *src2*, and *dst*. In the seventh group, 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 group, *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 3−12. Relationships Between Operands, Operand Size, Signed/Unsigned, Functional Units, and Opfields for Example Instruction (ADD)

Opcode map field used...	For operand type...	Unit	Opfield
src1 *src2* *dst*	sint xsint sint	.L1, .L2	000 0011
src1 *src2* *dst*	sint xsint slong	.L1, .L2	010 0011
src1 *src2* *dst*	xsint slong slong	.L1, .L2	010 0001
src1 *src2* *dst*	scst5 xsint sint	.L1, .L2	000 0010
src1 *src2* *dst*	scst5 slong slong	.L1, .L2	010 0000
src1 *src2* *dst*	sint xsint sint	.S1, .S2	00 0111
src1 *src2* *dst*	scst5 xsint sint	.S1, .S2	00 0110
src2 *src1* *dst*	sint sint sint	.D1, .D2	01 0000
src2 *src1* *dst*	sint ucst5 sint	.D1, .D2	01 0010

Example *The way each instruction is described*

Compatibility The C62x, C64x, and C67x DSPs share an instruction set. All of the instructions valid for the C62x DSP are also valid for the C67x DSP. This section identifies which DSP family the instruction is valid.

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

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

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

Execution for .D1, .D2 Opcodes

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

The execution describes the processing that takes place when the instruction is executed. The symbols are defined in Table 3−1 (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 4.2 (page 4-12) 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 (page 3-14) 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 — Absolute Value With Saturation

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

.unit = .L1 or .L2

Compatibility C62x, C64x, C67x, and C67x+ CPU

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	1	5	5		1	7				1	1

Opcode map field used...	For operand type...	Unit	Opfield
src2 *dst*	xsint sint	.L1, .L2	001 1010
src2 *dst*	slong slong	.L1, L2	011 1000

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

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

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

Pipeline

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

Instruction Type Single-cycle

Delay Slots 0

See Also **ABSDP, ABSSP**

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

ABSDP *Absolute Value, Double-Precision Floating-Point*

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

.unit = .S1 or .S2

Compatibility C67x and C67x+ CPU

Opcode

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

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

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

See Also **ABS, ABSSP**

Example

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

Before instruction

A1:A0	C004 0000h	0000 0000h	−2.5
A3:A2	xxxx xxxxh	xxxx xxxxh	

2 cycles after instruction

A1:A0	c004 0000h	0000 0000h	−2.5
A3:A2	4004 0000h	0000 0000h	2.5

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

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

.unit = . S1 or .S2

Compatibility C67x and C67x+ CPU

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	1	5	5		1						1	1

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

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

Execution

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

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

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

Notes:

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

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

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

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

Pipeline

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

Instruction Type Single-cycle

Delay Slots 0

Functional Unit Latency 1

See Also **ABS, ABSDP**

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

ADD — *Add Two Signed Integers Without Saturation*

Syntax

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

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

Compatibility C62x, C64x, C67x, and C67x+ CPU

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				1	1

Opcode map field used...	For operand type...	Unit	Opfield
src1 *src2* *dst*	sint xsint sint	.L1, .L2	000 0011
src1 *src2* *dst*	sint xsint slong	.L1, .L2	010 0011
src1 *src2* *dst*	xsint slong slong	.L1, .L2	010 0001
src1 *src2* *dst*	scst5 xsint sint	.L1, .L2	000 0010
src1 *src2* *dst*	scst5 slong slong	.L1, .L2	010 0000

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

Opcode map field used...	For operand type...	Unit	Opfield
src1 *src2* *dst*	sint xsint sint	.S1, .S2	00 0111
src1 *src2* *dst*	scst5 xsint sint	.S1, .S2	00 0110

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

Opcode map field used...	For operand type...	Unit	Opfield
src2 *src1* *dst*	sint sint sint	.D1, .D2	01 0000
src2 *src1* *dst*	sint ucst5 sint	.D1, .D2	01 0010

Description for .D1, .D2 Opcodes

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

Execution for .D1, .D2 Opcodes

if (cond)
$src2 + src1 \rightarrow dst$
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 **ADDDP, ADDK, ADDSP, ADDU, ADD2, SADD, SUB**

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 `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 3 `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 4 `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 — *Add Using Byte Addressing Mode*

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

.unit = .D1 or .D2

Compatibility C62x, C64x, C67x, and C67x+ CPU

Opcode

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

Opcode map field used...	For operand type...	Unit	Opfield
src2 *src1* *dst*	sint sint sint	.D1, .D2	11 0000
src2 *src1* *dst*	sint ucst*5* sint	.D1, .D2	11 0010

Description *src1* is added to *src2* using the byte 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 2.7.3, page 2-10). 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

See Also **ADD, ADDAD, ADDAH, ADDAW**

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

```
ADDAB .D1X B14,42h,A4
```

Before instruction		1 cycle after instruction	
B14	0020 1000h	A4	0020 1042h

Example 3

```
ADDAB .D2  B14,7FFFh,B4
```

Before instruction		1 cycle after instruction	
B14	0010 0000h	B4	0010 7FFFh

ADDAD *Add Using Doubleword Addressing Mode*

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

.unit = . D1 or .D2

Compatibility C67x and C67x+ CPU

Opcode

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

Opcode map field used...	For operand type...	Unit	Opfield
src2 *src1* *dst*	sint sint sint	.D1, .D2	11 1100
src2 *src1* *dst*	sint ucst5 sint	.D1, .D2	11 1101

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 2.7.3, page 2-10). *src1* is left shifted by 3 due to doubleword data sizes. The result is placed in *dst*.

> **Note:**
>
> There is no SUBAD instruction.

Execution if (cond) $src2 + (\text{src1} << 3) \rightarrow dst$
else nop

Pipeline

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

Instruction Type Single-cycle

Delay Slots 0

Functional Unit Latency 1

See Also **ADD, 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

ADDAH — *Add Using Halfword Addressing Mode*

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

.unit = .D1 or .D2

Compatibility C62x, C64x, C67x, and C67x+ CPU

Opcode

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

Opcode map field used...	For operand type...	Unit	Opfield
src2 *src1* *dst*	sint sint sint	.D1, .D2	11 0100
src2 *src1* *dst*	sint ucst*5* sint	.D1, .D2	11 0110

Description *src1* is added to *src2* using the halfword 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 2.7.3, page 2-10). *src1* is left shifted by 1. 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

See Also **ADD, ADDAB, ADDAD, ADDAW**

Example 1

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

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

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

Example 2

```
ADDAH .D1X B14,42h,A4
```

	Before instruction		1 cycle after instruction
B14	0020 1000h	A4	0020 1084h

Example 3

```
ADDAH .D2  B14,7FFFh,B4
```

	Before instruction		1 cycle after instruction
B14	0010 0000h	B4	0010 FFFEh

ADDAW *Add Using Word Addressing Mode*

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

.unit = .D1 or .D2

Compatibility C62x, C64x, C67x, and C67x+ CPU

Opcode

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

Opcode map field used...	For operand type...	Unit	Opfield
src2 *src1* *dst*	sint sint sint	.D1, .D2	11 1000
src2 *src1* *dst*	sint ucst*5* sint	.D1, .D2	11 1010

Description *src1* is added to *src2* using the word 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 2.7.3, page 2-10). *src1* is left shifted by 2. The result is placed in *dst*.

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

Pipeline

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

Instruction Type Single-cycle

Delay Slots 0

See Also **ADD, ADDAB, ADDAD, ADDAH**

Example 1

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

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

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

Example 2

```
ADDAW .D1X B14,42h,A4
```

Before instruction		1 cycle after instruction	
B14	0020 1000h	A4	0020 1108h

Example 3

```
ADDAW .D2  B14,7FFFh,B4
```

Before instruction		1 cycle after instruction	
B14	0010 0000h	B4	0011 FFFCh

ADDDP — *Add Two Double-Precision Floating-Point Values*

Syntax

ADDDP (.unit) *src1*, *src2*, *dst* (C67x and C67x+ CPU)
.unit = .L1 or .L2
or
ADDDP (.unit) *src1*, *src2*, *dst* (C67x+ CPU only)
.unit = .S1 or .S2

Compatibility

C67x and C67x+ CPU

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				1	1

Opcode map field used...	For operand type...	Unit	Opfield
src1 *src2* *dst*	dp xdp dp	.L1, .L2	001 1000
src1 *src2* *dst*	dp xdp dp	.S1, .S2	111 0010

Description

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

Execution

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

Notes:

1) This instruction takes the rounding mode from and sets the warning bits in FADCR, not FAUCR as for other .S unit instructions.

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

3) If one source is SNaN or QNaN, the result is NaN_out. If either source is SNaN, the INVAL bit is set, also.

4) If one source is +infinity and the other is −infinity, the result is NaN_out and the INVAL bit is set.

5) 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.

6) 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

7) 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

8) 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.

9) 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.

10) 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 or .S	.L or .S					

For the C67x CPU, 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.

For the C67x+ CPU, the low half of the result is written out one cycle earlier than the high half. If *dst* is used as the source for the **ADDDP**, **CMPEQDP**, **CMPLTDP**, **CMPGTDP**, **MPYDP**, **MPYSPDP**, **MPYSP2DP**, 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

See Also **ADD, ADDSP, ADDU, SUBDP**

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

ADDK — *Add Signed 16-Bit Constant to Register*

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

.unit = .S1 or .S2

Compatibility C62x, C64x, C67x, and C67x+ CPU

Opcode

31–29	28	27–23	22–7	6	5	4	3	2	1	0
creg	*z*	*dst*	*cst16*	1	0	1	0	0	*s*	*p*
3	1	5	16						1	1

Opcode map field used...	For operand type...	Unit
cst16 *dst*	scst16 uint	.S1, .S2

Description A 16-bit signed constant, *cst16*, is added to the *dst* register specified. The result is placed in *dst*.

Execution

if (cond) *cst* + *dst* $\rightarrow$ *dst*
else nop

Pipeline

Pipeline Stage	E1
Read	*cst16*
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

ADDSP — *Add Two Single-Precision Floating-Point Values*

Syntax

ADDSP (.unit) *src1*, *src2*, *dst* (C67x and C67x+ CPU)
.unit = .L1 or .L2
or
ADDSP (.unit) *src1*, *src2*, *dst* (C67x+ CPU only)
.unit = .S1 or .S2

Compatibility

C67x and C67x+ CPU

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				1	1

Opcode map field used...	For operand type...	Unit	Opfield
src1	sp	.L1, .L2	001 0000
src2	xsp		
dst	sp		
src1	sp	.S1, .S2	111 0000
src2	xsp		
dst	sp		

Description

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

Execution

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

Notes:

1) This instruction takes the rounding mode from and sets the warning bits in FADCR, not FAUCR as for other .S unit instructions.
2) If rounding is performed, the INEX bit is set.
3) If one source is SNaN or QNaN, the result is NaN_out. If either source is SNaN, the INVAL bit is set also.
4) If one source is +infinity and the other is −infinity, the result is NaN_out and the INVAL bit is set.
5) 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.
6) 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

7) 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

8) 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.
9) 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.
10) 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 or .S			

Instruction Type 4-cycle

Delay Slots 3

Functional Unit Latency 1

See Also **ADD, ADDDP, ADDU, SUBSP**

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

ADDU — *Add Two Unsigned Integers Without Saturation*

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

.unit = .L1 or .L2

Compatibility C62x, C64x, C67x, and C67x+ CPU

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				1	1

Opcode map field used...	For operand type...	Unit	Opfield
src1 *src2* *dst*	uint xuint ulong	.L1, .L2	010 1011
src1 *src2* *dst*	xuint ulong ulong	.L1, .L2	010 1001

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

Execution

if (cond)
    $src1 + src2 \rightarrow dst$
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, SADD, SUBU**

Example 1 `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 xxxxh			A5:A4	0000 0001h	0000 316Ch	4294979948‡

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

Example 2 `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

ADD2 *Add Two 16-Bit Integers on Upper and Lower Register Halves*

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

.unit = .S1 or .S2

Compatibility C62x, C64x, C67x, and C67x+ CPU

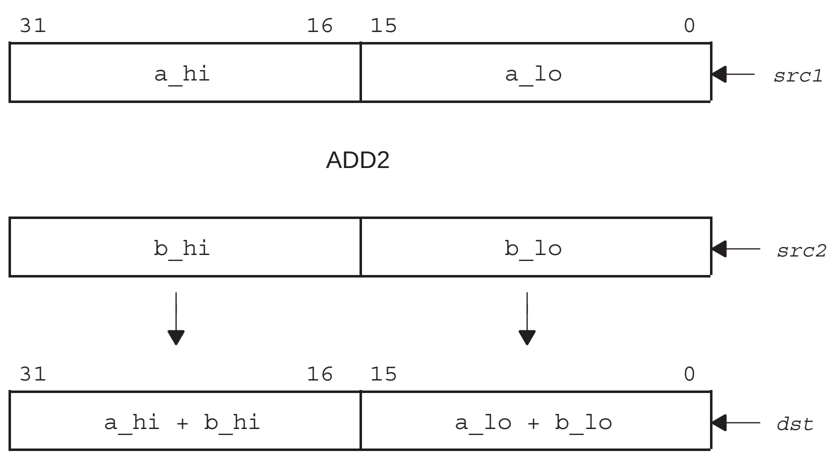
Opcode

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	1	5	5	5	1						1	1

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

Description 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.



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

Instruction Type Single-cycle

Delay Slots 0

See Also **ADD, ADDU, SUB2**

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, .L2, .S1, .S2

Compatibility C62x, C64x, C67x, and C67x+ CPU

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				1	1

Opcode map field used...	For operand type...	Unit	Opfield
src1 *src2* *dst*	uint xuint uint	.L1, .L2	111 1011
src1 *src2* *dst*	scst5 xuint uint	.L1, .L2	111 1010

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

Opcode map field used...	For operand type...	Unit	Opfield
src1 *src2* *dst*	uint xuint uint	.S1, .S2	01 1111
src1 *src2* *dst*	scst5 xuint uint	.S1, .S2	01 1110

Description Perrforms a bitwise **AND** operation between *src1* and *src2*. The result is placed in *dst.* The *scst5* operands are sign extended to 32 bits.

Execution if (cond) *src1* AND *src2* $\rightarrow$ *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

See Also **OR, XOR**

Example 1

```
AND .L1X   A1,B1,A2
```

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

Example 2

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

	Before instruction	1 cycle after instruction
A1	32E4 6936h	32E4 6936h
A3	xxxx xxxxh	0000 0006h

B — *Branch Using a Displacement*

Syntax

B (.unit) label

.unit = .S1 or .S2

Compatibility

C62x, C64x, C67x, and C67x+ CPU

Opcode

31–29	28	27–7	6	5	4	3	2	1	0
creg	*z*	*cst21*	0	0	1	0	0	*s*	*p*
3	1	21						1	1

Opcode map field used...	For operand type...	Unit
cst21	*scst21*	.S1, .S2

Description

A 21-bit signed constant, *cst21*, 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 *cst21* by the following formula:

cst21 = (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) *cst21* << 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 3-17 for information on branching into the middle of an execute packet.

Pipeline

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

Instruction Type Branch

Delay Slots 5

Example Table 3−13 gives the program counter values and actions for the following code 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 3−13. 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

Compatibility C62x, C64x, C67x, and C67x+ CPU

Opcode

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

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

Description *src2* is placed in the program fetch counter (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 only 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.
3) See section 3.5.2 on page 3-17 for information on branching into the middle of an execute packet.

Pipeline

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

Instruction Type Branch

Delay Slots 5

Example Table 3−14 gives the program counter values and actions for the following code example. In this example, the B10 register holds the value 1000 000Ch.

```
            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 3−14. 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

Compatibility

C62x, C64x, C67x, and C67x+ CPU

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	1	5			1						1	1

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

Description

IRP is placed in the program fetch counter (PFC). This instruction also moves the PGIE bit value to the GIE bit. The PGIE bit 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) IRP $\rightarrow$ PFC
else nop

Notes:

1) This instruction executes on .S2 only. PFC is the program fetch counter.
2) Refer to the chapter on interrupts for more information on IRP, PGIE, and GIE.
3) The execute packets in the delay slots of a branch cannot be interrupted. This is true regardless of whether the branch is taken.
4) See section 3.5.2 on page 3-17 for information on branching into the middle of an execute packet.

Pipeline

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

Instruction Type Branch

Delay Slots 5

Example Table 3−15 gives the program counter values and actions for the following code 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 3−15. Program Counter Values for B IRP Instruction

Cycle	Program Counter Value	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

Compatibility C62x, C64x, C67x, and C67x+ CPU

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	1	5			1						1	1

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

Description NRP is placed in the program fetch counter (PFC). This instruction also sets the NMIE bit. The PGIE bit 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.
4) See section 3.5.2 on page 3-17 for information on branching into the middle of an execute packet.

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

Example Table 3−16 gives the program counter values and actions for the following code 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 3−16. Program Counter Values for B NRP Instruction

Cycle	Program Counter Value	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

Compatibility C62x, C64x, C67x, and C67x+ CPU

Opcode Constant form

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

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

Opcode Register form

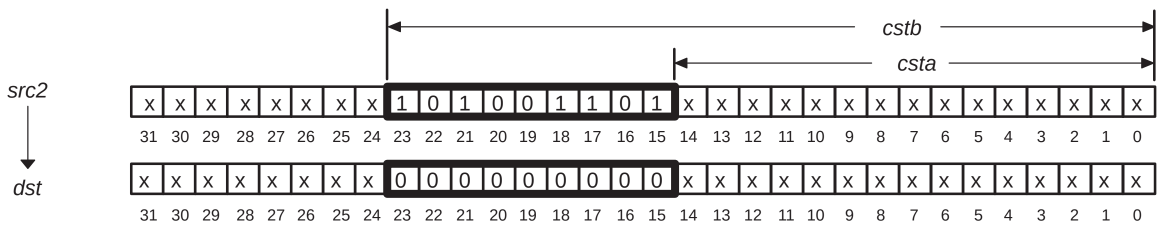
31–29	28	27–23	22–18	17–13	12	11–6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	1 1 1 0 1 1	1	0	0	0	*s*	*p*
3	1	5	5	5	1						1	1

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

Description

For *cstb* > *csta*, the field in *src2* as specified by *csta* to *cstb* is cleared to all 0s in *dst*. The *csta* and *cstb* operands may be specified as constants or in the 10 LSBs of the *src1* register, with *cstb* being bits 0−4 ($src1_{4..0}$) and *csta* being bits 5−9 ($src1_{9..5}$). *csta* is the LSB of the field and *cstb* is the MSB of the field. In other words, *csta* and *cstb* represent the beginning and ending bits, respectively, of the field to be cleared to all 0s in *dst*. The LSB location of *src2* is bit 0 and the MSB location of *src2* is bit 31.

In the following example, *csta* is 15 and *cstb* is 23. For the register version of the instruction, only the 10 LSBs of the *src1* register are valid. If any of the 22 MSBs are non-zero, the result is invalid.



For *cstb* < *csta*, the *src2* register is copied to *dst*. The *csta* and *cstb* operands may be specified as constants or in the 10 LSBs of the *src1* register, with *cstb* being bits 0−4 ($src1_{4..0}$) and *csta* being bits 5−9 ($src1_{9..5}$).

Execution

If the constant form is used when *cstb* > *csta*:

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

If the register form is used when *cstb* > *csta*:

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

Example 1

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

	Before instruction	**1 cycle after instruction**
A1	07A4 3F2Ah	07A4 3F2Ah
A2	xxxx xxxxh	07A0 000Ah

Example 2

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

	Before instruction	**1 cycle after instruction**
B1	03B6 E7D5h	03B6 E7D5h
B2	xxxx xxxxh	03B0 0001h
B3	0000 0052h	0000 0052h

CMPEQ — *Compare for Equality, Signed Integers*

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

.unit = .L1 or .L2

Compatibility C62x, C64x, C67x, and C67x+ CPU

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				1	1

Opcode map field used...	For operand type...	Unit	Opfield
src1 *src2* *dst*	sint xsint uint	.L1, .L2	101 0011
src1 *src2* *dst*	scst5 xsint uint	.L1, .L2	101 0010
src1 *src2* *dst*	xsint slong uint	.L1, .L2	101 0001
src1 *src2* *dst*	scst5 slong uint	.L1, .L2	101 0000

Description 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

See Also **CMPEQDP, CMPEQSP, CMPGT, CMPLT**

Example 1

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

	Before instruction			1 cycle after instruction	
A1	0000 04B8h	1208	A1	0000 04B8h	
A2	xxxx xxxxh		A2	0000 0000h	false
B1	0000 04B7h	1207	B1	0000 04B7h	

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 00FFh	F23A 3789h	B3:B2	0000 00FFh	F23A 3789h

CMPEQDP — *Compare for Equality, Double-Precision Floating-Point Values*

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

.unit = .S1 or .S2

Compatibility C67x and C67x+ CPU

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	1	5	5	5	1						1	1

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

Description 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
```

Special cases of inputs:

Input			FAUCR Bits	
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

See Also **CMPEQ, CMPEQSP, CMPGTDP, CMPLTDP**

Example

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

Before instruction

Register	High	Low	Value
A1:A0	4021 3333h	3333 3333h	8.6
A3:A2	C004 0000h	0000 0000h	−2.5
A4	xxxx xxxxh		

2 cycles after instruction

Register	High	Low	Value
A1:A0	4021 3333h	3333 3333h	8.6
A3:A2	C004 0000h	0000 0000h	−2.5
A4	0000 0000h		false

CMPEQSP — *Compare for Equality, Single-Precision Floating-Point Values*

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

.unit = .S1 or .S2

Compatibility C67x and C67x+ CPU

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	1	5	5	5	1						1	1

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

Description 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
```

Special cases of inputs:

Input			FAUCR Bits	
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

See Also **CMPEQ, CMPEQDP, CMPGTSP, CMPLTSP**

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

CMPGT — *Compare for Greater Than, Signed Integers*

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

.unit = .L1 or .L2

Compatibility C62x, C64x, C67x, and C67x+ CPU

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				1	1

Opcode map field used...	For operand type...	Unit	Opfield
src1 *src2* *dst*	sint xsint uint	.L1, .L2	100 0111
src1 *src2* *dst*	scst5 xsint uint	.L1, .L2	100 0110
src1 *src2* *dst*	xsint slong uint	.L1, .L2	100 0101
src1 *src2* *dst*	scst5 slong uint	.L1, .L2	100 0100

Description

Performs a signed comparison of *src1* to *src2*. If *src1* is greater than *src2*, then a 1 is written to *dst*; otherwise, a 0 is written to *dst*.

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

See Also **CMPEQ, CMPGTDP, CMPGTSP, CMPGTU, CMPLT**

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	

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

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

.unit = .S1 or .S2

Compatibility C67x and C67x+ CPU

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	1	5	5	5	1						1	1

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

Description Compares *src1* to *src2*. If *src1* is greater than *src2*, then 1 is written to *dst*; otherwise, 0 is written to *dst*.

Execution

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

Special cases of inputs:

Input			FAUCR Bits	
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 other than those shown above 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

See Also **CMPEQDP, CMPGT, CMPGTSP, CMPGTU, CMPLTDP**

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

Before instruction

A1:A0	4021 3333h	3333 3333h	8.6
A3:A2	c004 0000h	0000 0000h	−2.5
A4	XXXX XXXXh		

2 cycles after instruction

A1:A0	4021 3333h	3333 3333h	8.6
A3:A2	c004 0000h	0000 0000h	−2.5
A4	0000 0001h	true	

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

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

.unit = .S1 or .S2

Compatibility C67x and C67x+ CPU

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	1	5	5	5	1						1	1

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

Description Compares *src1* to *src2*. If *src1* is greater than *src2*, then 1 is written to *dst*; otherwise, 0 is written to *dst*.

Execution

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

Special cases of inputs:

Input			FAUCR Fields	
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 other than those shown above 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

See Also **CMPEQSP, CMPGT, CMPGTDP, CMPGTU, CMPLTSP**

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

CMPGTU *Compare for Greater Than, Unsigned Integers*

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

.unit = .L1 or .L2

Compatibility C62x, C64x, C67x, and C67x+ CPU

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				1	1

Opcode map field used...	For operand type...	Unit	Opfield
src1 *src2* *dst*	uint xuint uint	.L1, .L2	100 1111
src1 *src2* *dst*	ucst4 xuint uint	.L1, .L2	100 1110
src1 *src2* *dst*	xuint ulong uint	.L1, .L2	100 1101
src1 *src2* *dst*	ucst4 ulong uint	.L1, .L2	100 1100

Description Performs an unsigned comparison of *src1* to *src2*. If *src1* is greater than *src2*, then a 1 is written to *dst*; otherwise, a 0 is written to *dst*. Only the four LSBs are valid in the 5-bit *dst* field when the *ucst4* operand is used. If the MSB of the *dst* field is nonzero, the result is invalid.

Execution

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

Pipeline

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

Instruction Type Single-cycle

Delay Slots 0

See Also **CMPGT, CMPGTDP, CMPGTSP, CMPLTU**

Example 1

```
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

† Unsigned 32-bit integer

Example 2

```
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

† Unsigned 32-bit integer

Example 3

```
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 40-bit (long) integer

CMPLT *Compare for Less Than, Signed Integers*

Syntax

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

.unit = .L1 or .L2

Compatibility C62x, C64x, C67x, and C67x+ CPU

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				1	1

Opcode map field used...	For operand type...	Unit	Opfield
src1 *src2* *dst*	sint xsint uint	.L1, .L2	101 0111
src1 *src2* *dst*	scst5 xsint uint	.L1, .L2	101 0110
src1 *src2* *dst*	xsint slong uint	.L1, .L2	101 0101
src1 *src2* *dst*	scst5 slong uint	.L1, .L2	101 0100

Description

Performs a signed 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 $\rightarrow$ *dst*
else 0 $\rightarrow$ *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 **CMPEQ, CMPGT, CMPLTDP, CMPLTSP, CMPLTU**

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

CMPLTDP — *Compare for Less Than, Double-Precision Floating-Point Values*

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

.unit = .S1 or .S2

Compatibility C67x and C67x+ CPU

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	1	5	5	5	1						1	1

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

Description Compares *src1* to *src2*. If *src1* is less than *src2*, then 1 is written to *dst*; otherwise, 0 is written to *dst*.

Execution

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

Special cases of inputs:

Input			FAUCR Bits	
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 other than those above 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

See Also **CMPEQDP, CMPGTDP, CMPLT, CMPLTSP, CMPLTU**

Example

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

Before instruction

A1:A0	4021 3333h	3333 3333h	8.6
B3:B2	c004 0000h	0000 0000h	−2.5
A4	xxxx xxxxh		

2 cycles after instruction

A1:A0	4021 3333h	4021 3333h	8.6
B3:B2	c004 0000h	0000 0000h	−2.5
A4	0000 0000h	false	

CMPLTSP — *Compare for Less Than, Single-Precision Floating-Point Values*

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

.unit = .S1 or .S2

Compatibility C67x and C67x+ CPU

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	1	5	5	5	1						1	1

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

Description Compares *src1* to *src2*. If *src1* is less than *src2*, then 1 is written to *dst*; otherwise, 0 is written to *dst*.

Execution

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

Special cases of inputs:

Input			FAUCR Bits	
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 other than those above 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

See Also **CMPEQSP, CMPGTSP, CMPLT, CMPLTDP, CMPLTU**

Example

```
CMPLTSP .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

CMPLTU — *Compare for Less Than, Unsigned Integers*

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

.unit = .L1 or .L2

Compatibility C62x, C64x, C67x, and C67x+ CPU

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				1	1

Opcode map field used...	For operand type...	Unit	Opfield
src1 *src2* *dst*	uint xuint uint	.L1, .L2	101 1111
src1 *src2* *dst*	ucst4 xuint uint	.L1, .L2	101 1110
src1 *src2* *dst*	xuint ulong uint	.L1, .L2	101 1101
src1 *src2* *dst*	ucst4 ulong uint	.L1, .L2	101 1100

Description Performs an unsigned comparison of *src1* to *src2*. If *src1* is less than *src2*, then 1 is written to *dst*; otherwise, 0 is written to *dst*.

Execution

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

Pipeline

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

Instruction Type Single-cycle

Delay Slots 0

See Also **CMPGTU, CMPLT, CMPLTDP, CMPLTSP**

Example 1

```
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 2

```
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

† Unsigned 32-bit integer

Example 3

```
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

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

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

.unit = .L1 or .L2

Compatibility C67x and C67x+ CPU

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	1	5	5		1					1	1

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

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

See Also **DPSP, DPTRUNC, INTDP, SPINT**

Example

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

Before instruction

A1:A0	4021 3333h	3333 3333h	8.6
A4	xxxx xxxxh		

4 cycles after instruction

A1:A0	4021 3333h	3333 3333h	8.6
A4	0000 0009h		9

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

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

.unit = .L1 or .L2

Compatibility C67x and C67x+ CPU

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	1	5	5		1					1	1

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

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

See Also **DPINT, DPTRUNC, INTSP, SPDP**

Example

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

Before instruction

A1:A0	4021 3333h	3333 3333h	8.6
A4	xxxx xxxxh		

4 cycles after instruction

A1:A0	4021 3333h	4021 3333h	8.6
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

Compatibility C67x and C67x+ CPU

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	1	5	5		1					1	1

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

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

See Also **DPINT, DPSP, SPTRUNC**

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

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

Compatibility C62x, C64x, C67x, and C67x+ CPU

Opcode Constant form

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

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

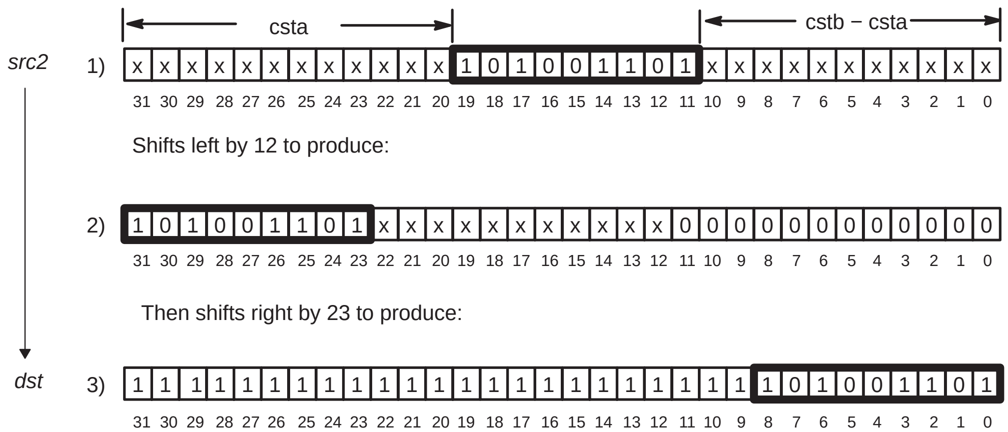
Opcode Register form

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

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

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

See Also **EXTU**

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

Compatibility C62x, C64x, C67x, and C67x+ CPU

Opcode Constant width and offset form:

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

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

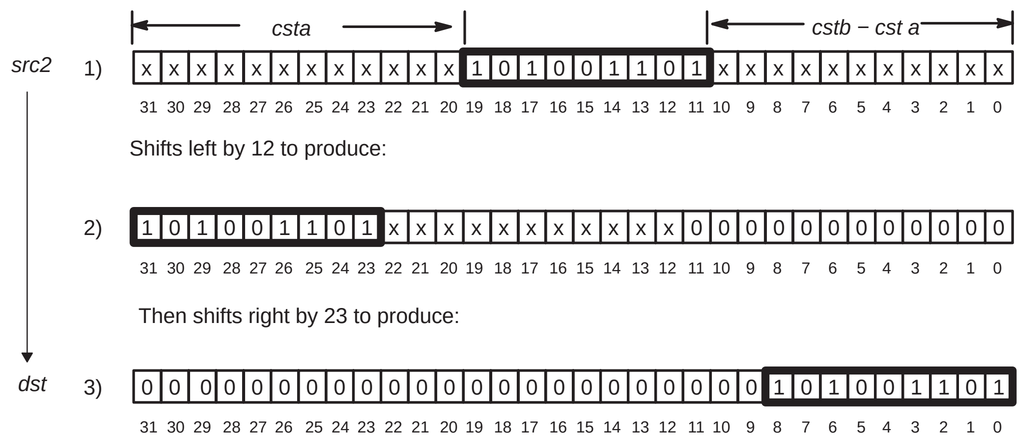
Opcode Register width and offset form:

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

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

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* $\rightarrow$ *dst*
else nop

If the register width and offset form is used:

if (cond) *src2* extu $src1_{9..5}$, $src1_{4..0}$ $\rightarrow$ *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 **EXT**

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**

.unit = none

Compatibility C62x, C64x, C67x, and C67x+ CPU

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	1

Description 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

INTDP — *Convert Signed Integer to Double-Precision Floating-Point Value*

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

.unit = .L1 or .L2

Compatibility C67x and C67x+ CPU

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

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

Description The signed 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

See Also **DPINT, INTDPU, INTSP, INTSPU**

Example `INTDP .L1x B4,A1:A0`

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

INTDPU *Convert Unsigned Integer to Double-Precision Floating-Point Value*

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

.unit = .L1 or .L2

Compatibility C67x and C67x+ CPU

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

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

Description The unsigned 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

See Also **INTDP, INTSP, INTSPU**

Example

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

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

INTSP *Convert Signed Integer to Single-Precision Floating-Point Value*

Syntax

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

.unit = .L1 or .L2

Compatibility

C67x and C67x+ CPU

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

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

Description

The signed 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

See Also

INTDP, INTDPU, INTSPU

Example

```
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

INTSPU — Convert Unsigned Integer to Single-Precision Floating-Point Value

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

.unit = .L1 or .L2

Compatibility C67x and C67x+ CPU

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

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

Description The unsigned 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

See Also **INTDP, INTDPU, INTSP**

Example

```
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

LDB(U) — *Load Byte From Memory With a 5-Bit Unsigned Constant Offset or Register Offset*

Syntax

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

.unit = .D1 or .D2

Compatibility C62x, C64x, C67x, and C67x+ CPU

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*	0	*y*	*op*	0	1	*s*	*p*
3	1	5	5	5	4		1	3			1	1

Description Loads a byte from memory to a general-purpose register (*dst*). Table 3−17 summarizes the data types supported by loads. Table 3−11 (page 3-32) 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 bits. 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.

Table 3−17. Data Types Supported by LDB(U) Instruction

Mnemonic	*op* 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

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 2.7.3, page 2-10).

For **LDB(U)**, the values are loaded into the 8 LSBs of *dst*. For **LDB**, the upper 24 bits of *dst* values are sign-extended; for **LDBU**, the upper 24 bits of *dst* are zero-filled. The *s* bit determines which file *dst* will be loaded into: *s* = 0 indicates *dst* will be loaded in the A register file and *s* = 1 indicates *dst* will be loaded in the B register file. The *r* bit should be cleared to 0.

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 0. Parentheses, ( ), can be used to set a nonscaled, constant offset. You must type either brackets or parentheses around the specified offset, if you use the optional offset parameter.

Execution

if (cond) mem $\rightarrow$ *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 4.

See Also **LDH, LDW**

Example

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

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

LDB(U) *Load Byte From Memory With a 15-Bit Unsigned Constant Offset*

Syntax

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

.unit = .D2

Compatibility

C62x, C64x, C67x, and C67x+ CPU

Opcode

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

Description

Loads a byte from memory to a general-purpose register (*dst*). Table 3−18 summarizes the data types supported by loads. The memory address is formed from a base address register 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 bits. 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 **LDB(U)**, the values are loaded into the 8 LSBs of *dst*. For **LDB**, the upper 24 bits of *dst* values are sign-extended; for **LDBU**, the upper 24 bits of *dst* are zero-filled. The *s* bit determines which file *dst* will be loaded into: *s* = 0 indicates *dst* will be loaded in the A register file and *s* = 1 indicates *dst* will be loaded in the B register file.

Square brackets, [ ], indicate that the *ucst*15 offset is left-shifted by 0. Parentheses, ( ), can be used to set a nonscaled, constant offset. You must type either brackets or parentheses around the specified offset, if you use the optional offset parameter.

Table 3−18. Data Types Supported by LDB(U) Instruction (15-Bit Offset)

Mnemonic	*op* 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

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

See Also **LDH, LDW**

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

LDDW — *Load Doubleword From Memory With an Unsigned Constant Offset or Register Offset*

Syntax

Register Offset	**Unsigned Constant Offset**
LDDW (.unit) *+*baseR[offsetR]*, *dst*	**LDDW** (.unit) *+*baseR[ucst5]*, *dst*

.unit = .D1 or .D2

Compatibility C67x and C67x+ CPU

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*	1	*y*	1 1 0	0	1	*s*	*p*
3	1	5	5	5	4		1				1	1

Description Loads a doubleword from memory into a register pair *dst_o:dst_e.* Table 3−11 (page 3-32) 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. The *dst* field must always be an even value because the **LDDW** instruction loads register pairs. Therefore, bit 23 is always zero.

The *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 parenthesis 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 2.7.3, page 2-10).

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 the **LDDW** instruction 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 doubleword address must be on a doubleword boundary (the three LSBs are zero).

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

See Also **LDB, LDH, LDW**

Example 1

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

Before instruction

A1:A0	xxxx xxxxh	xxxx xxxxh	
B10	0000 0010h		16
mem 18h	3333 3333h	4021 3333h	8.6

5 cycles after instruction

A1:A0	4021 3333h	3333 3333h	
B10	0000 0010h		16
mem 18h	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 18h	4021 3333h	3333 3333h	8.6

1 cycle after instruction

A1:A0	xxxx xxxxh	xxxx xxxxh	
A10	0000 0018h		24
mem 18h	4021 3333h	3333 3333h	8.6

5 cycles after instruction

A1:A0	4021 3333h	3333 3333h	
A10	0000 0018h		24
mem 18h	4021 3333h	3333 3333h	8.6

Big-endian mode

LDH(U) — *Load Halfword From Memory With a 5-Bit Unsigned Constant Offset or Register Offset*

Syntax

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

.unit = .D1 or .D2

Compatibility C62x, C64x, C67x, and C67x+ CPU

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*	0	*y*	*op*	0	1	*s*	*p*
3	1	5	5	5	4		1	3			1	1

Description Loads a halfword from memory to a general-purpose register (*dst*). Table 3−19 summarizes the data types supported by halfword loads. Table 3−11 (page 3-32) 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 1 bit. 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.

Table 3−19. Data Types Supported by LDH(U) Instruction

Mnemonic	*op* Field	Load Data Type	SIze	Left Shift of Offset
LDH	1 0 0	Load halfword	16	1 bit
LDHU	0 0 0	Load halfword unsigned	16	1 bit

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 2.7.3, page 2-10).

For **LDH(U)**, the values are loaded into the 16 LSBs of *dst*. For **LDH**, the upper 16 bits of *dst* are sign-extended; for **LDHU**, the upper 16 bits of *dst* are zero-filled. The *s* bit determines which file *dst* will be loaded into: *s* = 0 indicates *dst* will be loaded in the A register file and *s* = 1 indicates *dst* will be loaded in the B register file. The *r* bit should be cleared to 0.

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 1. Parentheses, ( ), can be used to set a nonscaled, constant offset. You must type either brackets or parentheses around the specified offset, if you use the optional offset parameter.

Halfword addresses must be aligned on halfword (LSB is 0) boundaries.

Execution

if (cond) mem $\rightarrow$ *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 4.

See Also **LDB, LDW**

Example

```
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

LDH(U) *Load Halfword From Memory With a 15-Bit Unsigned Constant Offset*

Syntax

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

.unit = .D2

Compatibility

C62x, C64x, C67x, and C67x+ CPU

Opcode

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

Description

Loads a halfword from memory to a general-purpose register (*dst*). Table 3−20 summarizes the data types supported by loads. The memory address is formed from a base address register 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 1 bit. 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)**, the values are loaded into the 16 LSBs of *dst*. For **LDH**, the upper 16 bits of *dst* are sign-extended; for **LDHU**, the upper 16 bits of *dst* are zero-filled. The *s* bit determines which file *dst* will be loaded into: *s* = 0 indicates *dst* will be loaded in the A register file and *s* = 1 indicates *dst* will be loaded in the B register file.

Square brackets, [ ], indicate that the *ucst*15 offset is left-shifted by 1. Parentheses, ( ), can be used to set a nonscaled, constant offset. You must type either brackets or parentheses around the specified offset, if you use the optional offset parameter.

Halfword addresses must be aligned on halfword (LSB is 0) boundaries.

Table 3−20. Data Types Supported by LDH(U) Instruction (15-Bit Offset)

Mnemonic	*op* Field	Load Data Type	SIze	Left Shift of Offset
LDH	1 0 0	Load halfword	16	1 bit
LDHU	0 0 0	Load halfword unsigned	16	1 bit

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

See Also **LDB, LDW**

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

Syntax

Register Offset	Unsigned Constant Offset
LDW (.unit) *+*baseR[offsetR]*, *dst*	**LDW** (.unit) *+*baseR[ucst5]*, *dst*

.unit = .D1 or .D2

Compatibility C62x, C64x, C67x, and C67x+ CPU

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*	0	*y*	1 1 0	0	1	*s*	*p*
3	1	5	5	5	4		1				1	1

Description Loads a word from memory to a general-purpose register (*dst*). Table 3−11 (page 3-32) 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 2 bits. 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 2.7.3, page 2-10).

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 loaded in the A register file and *s* = 1 indicates *dst* will be loaded in the B register file. The *r* bit should be cleared to 0.

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. 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 addresses must be aligned on word (two LSBs are 0) boundaries.

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 4.

See Also **LDB, LDDW, LDH**

Example 1 `LDW .D1 *A10,B1`

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

Example 2 `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 3 `LDW .D1 *++A4[1],A6`

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

LDW — *Load Word From Memory With a 15-Bit Unsigned Constant Offset*

Syntax

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

.unit = .D2

Compatibility

C62x, C64x, C67x, and C67x+ CPU

Opcode

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

Description

Load a word from memory to a general-purpose register (*dst*). The memory address is formed from a base address register 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 2 bits. 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 **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* will be loaded in the A register file and *s* = 1 indicates *dst* will be loaded in the B register file.

Square brackets, [ ], indicate that the *ucst*15 offset is left-shifted by 2. 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 addresses must be aligned on word (two LSBs are 0) boundaries.

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

See Also **LDB, LDH**

LMBD *Leftmost Bit Detection*

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

.unit = .L1 or .L2

Compatibility C62x, C64x, C67x, and C67x+ CPU

Opcode

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				1	1

Opcode map field used...	For operand type...	Unit	Opfield
src1 *src2* *dst*	uint xuint uint	.L1, .L2	110 1011
src1 *src2* *dst*	cst5 xuint uint	.L1, .L2	110 1010

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 — *Multiply Signed 16 LSB × Signed 16 LSB*

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

.unit = .M1 or .M2

Compatibility C62x, C64x, C67x, and C67x+ CPU

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

Opcode map field used...	For operand type...	Unit	Opfield
src1 *src2* *dst*	slsb16 xslsb16 sint	.M1, .M2	11001
src1 *src2* *dst*	scst5 xslsb16 sint	.M1, .M2	11000

Description The *src1* operand is multiplied by the *src2* operand. The result is placed in *dst*. The source operands are signed by default.

Execution

if (cond) lsb16(*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

See Also **MPYU, MPYSU, MPYUS, SMPY**

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

† Signed 16-LSB integer

Example 2

```
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

† Signed 16-LSB integer

MPYDP *Multiply Two Double-Precision Floating-Point Values*

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

.unit = .M1 or .M2

Compatibility C67x and C67x+ CPU

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	1	5	5	5	1							1	1

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

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) 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.

5) 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

See Also **MPY, MPYSP**

Example

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

Before instruction

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

10 cycles after instruction

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

MPYH — *Multiply Signed 16 MSB × Signed 16 MSB*

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

.unit = .M1 or .M2

Compatibility C62x, C64x, C67x, and C67x+ CPU

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

Opcode map field used...	For operand type...	Unit
src1	smsb16	.M1, .M2
src2	xsmsb16	
dst	sint	

Description The *src1* operand is multiplied by the *src2* operand. The result is placed in *dst*. The source operands are signed by default.

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

See Also **MPYHU, MPYHSU, MPYHUS, SMPYH**

Example

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

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

† Signed 16-MSB integer

MPYHL — *Multiply Signed 16 MSB × Signed 16 LSB*

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

.unit = .M1 or .M2

Compatibility C62x, C64x, C67x, and C67x+ CPU

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

Opcode map field used...	For operand type...	Unit
src1	smsb16	.M1, .M2
src2	xslsb16	
dst	sint	

Description The *src1* operand is multiplied by the *src2* operand. The result is placed in *dst*. The source operands are signed by default.

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

See Also **MPYHLU, MPYHSLU, MPYHULS, SMPYHL**

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

MPYHLU *Multiply Unsigned 16 MSB × Unsigned 16 LSB*

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

.unit = .M1 or .M2

Compatibility C62x, C64x, C67x, and C67x+ CPU

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

Opcode map field used...	For operand type...	Unit
src1 *src2* *dst*	umsb16 xulsb16 uint	.M1, .M2

Description The *src1* operand is multiplied by the *src2* operand. The result is placed in *dst*. The source operands are unsigned by default.

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

See Also **MPYHL, MPYHSLU, MPYHULS**

MPYHSLU — *Multiply Signed 16 MSB × Unsigned 16 LSB*

Syntax — **MPYHSLU** (.unit) *src1*, *src2*, *dst*

.unit = .M1 or .M2

Compatibility — C62x, C64x, C67x, and C67x+ CPU

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

Opcode map field used...	For operand type...	Unit
src1	smsb16	.M1, .M2
src2	xulsb16	
dst	sint	

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

See Also — **MPYHL, MPYHLU, MPYHULS**

MPYHSU — *Multiply Signed 16 MSB × Unsigned 16 MSB*

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

.unit = .M1 or .M2

Compatibility C62x, C64x, C67x, and C67x+ CPU

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

Opcode map field used...	For operand type...	Unit
src1 *src2* *dst*	smsb16 xumsb16 sint	.M1, .M2

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

See Also **MPYH, MPYHU, MPYHUS**

Example

```
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

MPYHU — *Multiply Unsigned 16 MSB × Unsigned 16 MSB*

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

.unit = .M1 or .M2

Compatibility C62x, C64x, C67x, and C67x+ CPU

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

Opcode map field used...	For operand type...	Unit
src1 *src2* *dst*	umsb16 xumsb16 uint	.M1, .M2

Description The *src1* operand is multiplied by the *src2* operand. The result is placed in *dst*. The source operands are unsigned by default.

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

See Also **MPYH, MPYHSU, MPYHUS**

Example

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

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

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

MPYHULS — *Multiply Unsigned 16 MSB × Signed 16 LSB*

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

.unit = .M1 or .M2

Compatibility C62x, C64x, C67x, and C67x+ CPU

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

Opcode map field used...	For operand type...	Unit
src1	umsb16	.M1, .M2
src2	xslsb16	
dst	sint	

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

See Also **MPYHL, MPYHLU, MPYHSLU**

MPYHUS — *Multiply Unsigned 16 MSB × Signed 16 MSB*

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

.unit = .M1 or .M2

Compatibility C62x, C64x, C67x, and C67x+ CPU

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

Opcode map field used...	For operand type...	Unit
src1 *src2* *dst*	umsb16 xsmsb16 sint	.M1, .M2

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

See Also **MPYH, MPYHU, MPYHSU**

MPYI *Multiply 32-Bit × 32-Bit Into 32-Bit Result*

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

.unit = .M1 or .M2

Compatibility C67x and C67x+ CPU

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

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

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

See Also **MPYID**

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 *Multiply 32-Bit × 32-Bit Into 64-Bit Result*

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

.unit = .M1 or .M2

Compatibility C67x and C67x+ CPU

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

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

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

See Also **MPYI**

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

MPYLH — *Multiply Signed 16 LSB × Signed 16 MSB*

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

.unit = .M1 or .M2

Compatibility C62x, C64x, C67x, and C67x+ CPU

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

Opcode map field used...	For operand type...	Unit
src1	slsb16	.M1, .M2
src2	xsmsb16	
dst	sint	

Description The *src1* operand is multiplied by the *src2* operand. The result is placed in *dst*. The source operands are signed by default.

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

See Also **MPYLHU, MPYLSHU, MPYLUHS, SMPYLH**

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

MPYLHU — *Multiply Unsigned 16 LSB × Unsigned 16 MSB*

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

.unit = .M1 or .M2

Compatibility C62x, C64x, C67x, and C67x+ CPU

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

Opcode map field used...	For operand type...	Unit
src1	ulsb16	.M1, .M2
src2	xumsb16	
dst	uint	

Description The *src1* operand is multiplied by the *src2* operand. The result is placed in *dst*. The source operands are unsigned by default.

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

See Also **MPYLH, MPYLSHU, MPYLUHS**

MPYLSHU — *Multiply Signed 16 LSB × Unsigned 16 MSB*

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

.unit = .M1 or .M2

Compatibility C62x, C64x, C67x, and C67x+ CPU

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

Opcode map field used...	For operand type...	Unit
src1	slsb16	.M1, .M2
src2	xumsb16	
dst	sint	

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

See Also **MPYLH, MPYLHU, MPYLUHS**

MPYLUHS — *Multiply Unsigned 16 LSB × Signed 16 MSB*

Syntax — **MPYLUHS** (.unit) *src1*, *src2*, *dst*

.unit = .M1 or .M2

Compatibility — C62x, C64x, C67x, and C67x+ CPU

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

Opcode map field used...	For operand type...	Unit
src1	ulsb16	.M1, .M2
src2	xsmsb16	
dst	sint	

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

Execution

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

Pipeline

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

Instruction Type — Multiply (16 $\times$ 16)

Delay Slots — 1

See Also — **MPYLH, MPYLHU, MPYLSHU**

MPYSP *Multiply Two Single-Precision Floating-Point Values*

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

.unit = .M1 or .M2

Compatibility C67x and C67x+ CPU

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	1	5	5	5	1							1	1

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

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

Execution

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

Notes:

1) If one source is SNaN or QNaN, the result is a signed NaN_out. If either source is SNaN, the INVAL bit is set also. The sign of NaN_out is the exclusive-OR of the input signs.
2) Signed infinity multiplied by signed infinity or a normalized number (other than signed 0) returns signed infinity. Signed infinity multiplied by signed 0 returns a signed NaN_out and sets the INVAL bit.
3) If one or both sources are signed 0, the result is signed 0 unless the other source is NaN or signed infinity, in which case the result is signed NaN_out.
4) 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.
5) 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

See Also **MPY, MPYDP, MPYSP2DP**

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

MPYSPDP — *Multiply Single-Precision Floating-Point Value × Double-Precision Floating-Point Value*

Syntax

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

.unit = .M1 or .M2

Compatibility

C67x+ CPU only

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

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

Description

The single-precision *src1* operand is multiplied by the double-precision *src2* operand to produce a double-precision result. The result is placed in *dst*.

Execution

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

Notes:

1) If one source is SNaN or QNaN, the result is a signed NaN_out. If either source is SNaN, the INVAL bit is set also. The sign of NaN_out is the exclusive-OR of the input signs.
2) Signed infinity multiplied by signed infinity or a normalized number (other than signed 0) returns signed infinity. Signed infinity multiplied by signed 0 returns a signed NaN_out and sets the INVAL bit.
3) If one or both sources are signed 0, the result is signed 0 unless the other source is NaN or signed infinity, in which case the result is signed NaN_out.
4) 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.
5) If rounding is performed, the INEX bit is set.

Pipeline

Pipeline Stage	E1	E2	E3	E4	E5	E6	E7
Read	*src1* *src2_l*	*src1* *src2_h*					
Written						*dst_l*	*dst_h*
Unit in use	.M	.M					

The low half of the result is written out one cycle earlier than the high half. If *dst* is used as the source for the **ADDDP**, **CMPEQDP**, **CMPLTDP**, **CMPGTDP**, **MPYDP**, **MPYSPDP**, **MPYSP2DP**, 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 MPYSPDP

Delay Slots 6

Functional Unit Latency 3

See Also **MPY, MPYDP, MPYSP, MPYSP2DP**

MPYSP2DP — *Multiply Two Single-Precision Floating-Point Values for Double-Precision Result*

Syntax **MPYSP2DP** (.unit) *src1*, *src2*, *dst*

.unit = .M1 or .M2

Compatibility C67x+ CPU only

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

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

Description The *src1* operand is multiplied by the *src2* operand to produce a double-precision result. The result is placed in *dst*.

Execution

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

Notes:

1) If one source is SNaN or QNaN, the result is a signed NaN_out. If either source is SNaN, the INVAL bit is set also. The sign of NaN_out is the exclusive-OR of the input signs.
2) Signed infinity multiplied by signed infinity or a normalized number (other than signed 0) returns signed infinity. Signed infinity multiplied by signed 0 returns a signed NaN_out and sets the INVAL bit.
3) If one or both sources are signed 0, the result is signed 0 unless the other source is NaN or signed infinity, in which case the result is signed NaN_out.
4) 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.
5) If rounding is performed, the INEX bit is set.

Pipeline

Pipeline Stage	E1	E2	E3	E4	E5
Read	*src1* *src2*				
Written				*dst_l*	*dst_h*
Unit in use	.M				

The low half of the result is written out one cycle earlier than the high half. If *dst* is used as the source for the **ADDDP**, **CMPEQDP**, **CMPLTDP**, **CMPGTDP**, **MPYDP**, **MPYSPDP**, **MPYSP2DP**, 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 5-cycle

Delay Slots 4

Functional Unit Latency 2

See Also **MPY, MPYDP, MPYSP, MPYSPDP**

MPYSU *Multiply Signed 16 LSB × Unsigned 16 LSB*

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

.unit = .M1 or .M2

Compatibility C62x, C64x, C67x, and C67x+ CPU

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

Opcode map field used...	For operand type...	Unit	Opfield
src1 *src2* *dst*	slsb16 xulsb16 sint	.M1, .M2	11011
src1 *src2* *dst*	scst5 xulsb16 sint	.M1, .M2	11110

Description The signed operand *src1* is multiplied by the unsigned operand *src2*. The result is placed in *dst*. 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*) × 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

See Also **MPY, MPYU, MPYUS**

Example

```
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

‡ Unsigned 16-LSB integer

MPYU — *Multiply Unsigned 16 LSB × Unsigned 16 LSB*

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

.unit = .M1 or .M2

Compatibility C62x, C64x, C67x, and C67x+ CPU

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

Opcode map field used...	For operand type...	Unit
src1	ulsb16	.M1, .M2
src2	xulsb16	
dst	uint	

Description The *src1* operand is multiplied by the *src2* operand. The result is placed in *dst*. The source operands are unsigned by default.

Execution

if (cond) lsb16(*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

See Also **MPY, MPYSU, MPYUS**

Example

```
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§

‡ Unsigned 16-LSB integer

MPYUS — *Multiply Unsigned 16 LSB × Signed 16 LSB*

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

.unit = .M1 or .M2

Compatibility C62x, C64x, C67x, and C67x+ CPU

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

Opcode map field used...	For operand type...	Unit
src1	ulsb16	.M1, .M2
src2	xslsb16	
dst	sint	

Description The unsigned operand *src1* is multiplied by the signed operand *src2*. The result is placed in *dst*. 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*) × 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

See Also **MPY, MPYU, MPYSU**

Example

```
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

MV — *Move From Register to Register*

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

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

Compatibility C62x, C64x, C67x, and C67x+ CPU

Opcode .L unit

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	1	5	5		1	7				1	1

Opcode map field used...	For operand type...	Unit	Opfield
src2 *dst*	xsint sint	.L1, .L2	000 0010
src2 *dst*	slong slong	.L1, .L2	010 0000

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*	0 0 0 0 0	x	0 0 0 1 1 0	1	0	0	0	*s*	*p*
3	1	5	5		1						1	1

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

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

Opcode map field used...	For operand type...	Unit
*src*2 *dst*	sint sint	.D1, .D2

Description The **MV** pseudo-operation moves a value from one register to another. The assembler uses the operation **ADD** (.unit) 0, *src2*, *dst* to perform this task.

Execution if (cond) 0 + *src2* → *dst*
else nop

Instruction Type Single-cycle

Delay Slots 0

MVC — *Move Between Control File and Register File*

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

.unit = .S2

Compatibility C62x, C64x, C67x, and C67x+ CPU

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	1	5	5		1	6					1	1

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	00 1111

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.

Register addresses for accessing the control registers are in Table 3−21 (page 3-181).

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	00 1110

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 3−21 (page 3-181).

Execution

if (cond) *src2* → *dst*
else nop

> **Note:**
>
> The **MVC** instruction executes only on the B side (.S2).
>
> Refer to the individual control register descriptions for specific behaviors and restrictions in accesses via the **MVC** instruction.

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	B1	F009 0001h
AMR	0000 0000h	AMR	0009 0001h

> **Note:**
>
> The six MSBs of the AMR are reserved and therefore are not written to.

Table 3−21. Register Addresses for Accessing the Control Registers

Acronym	Register Name	Address	Read/ Write
AMR	Addressing mode register	00000	R, W
CSR	Control status register	00001	R, W
FADCR	Floating-point adder configuration	10010	R, W
FAUCR	Floating-point auxiliary configuration	10011	R, W
FMCR	Floating-point multiplier configuration	10100	R, W
ICR	Interrupt clear register	00011	W
IER	Interrupt enable register	00100	R, W
IFR	Interrupt flag register	00010	R
IRP	Interrupt return pointer	00110	R, W
ISR	Interrupt set register	00010	W
ISTP	Interrupt service table pointer	00101	R, W
NRP	Nonmaskable interrupt return pointer	00111	R, W
PCE1	Program counter, E1 phase	10000	R

Legend: R = Readable by the **MVC** instruction; W = Writeable by the **MVC** instruction

MVK — *Move Signed Constant Into Register and Sign Extend*

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

.unit = .S1 or .S2

Compatibility C62x, C64x, C67x, and C67x+ CPU

Opcode

31–29	28	27–23	22–7	6	5	4	3	2	1	0
creg	*z*	*dst*	*cst16*	0	1	0	1	0	*s*	*p*
3	1	5	16						1	1

Opcode map field used...	For operand type...	Unit
cst16	scst16	.S1, .S2
dst	sint	

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

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 FFFF 8000h to 0000 7FFFh).

For example:

```
MVK  .S1   0x00008000X, A0
```

will generate a warning; whereas:

```
MVK  .S1   0xFFFF8000, A0
```

will not generate a warning.

Execution

if (cond) scst → *dst*
else nop

Pipeline

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

Instruction Type Single cycle

Delay Slots 0

See Also **MVKH, MVKL, 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

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

Syntax

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

.unit = .S1 or .S2

Compatibility C62x, C64x, C67x, and C67x+ CPU

Opcode

31–29	28	27–23	22–7	6	5	4	3	2	1	0
creg	*z*	*dst*	*cst16*	1	1	0	1	0	*s*	*p*
3	1	5	16						1	1

Opcode map field used...	For operand type...	Unit
cst16	uscst16	.S1, .S2
dst	sint	

Description The 16-bit constant, *cst16* , is loaded into the upper 16 bits of *dst*. The 16 LSBs of *dst* are unchanged. For the **MVKH** instruction, the assembler encodes the 16 MSBs of a 32-bit constant into the *cst16* field of the opcode . For the **MVKLH** instruction, the assembler encodes the 16 LSBs of a constant into the *cst16* field of the opcode.

Execution For the **MVKLH** instruction:

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

For the **MVKH** instruction:

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

Pipeline

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

Instruction Type Single-cycle

Delay Slots 0

Note:

Use the **MVK** instruction (page 3-182) to load 16-bit constants. The assembler generates a warning for any constant over 16 bits. To load 32-bit constants, such as 1234 5678h, 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	0A32 7634h

Example 2

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

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

MVKL — *Move Signed Constant Into Register and Sign Extend*

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

.unit = .S1 or .S2

Compatibility C62x, C64x, C67x, and C67x+ CPU

Opcode

31–29	28	27–23	22–7	6	5	4	3	2	1	0
creg	*z*	*dst*	*cst16*	0	1	0	1	0	*s*	*p*
3	1	5	16						1	1

Opcode map field used...	For operand type...	Unit
cst16	scst16	.S1, .S2
dst	sint	

Description The **MVKL** pseudo-operation sign extends the 16-bit constant, *cst16*, and places it in *dst*.

The **MVKL** instruction is equivalent to the **MVK** instruction (page 3-182), except that the **MVKL** instruction disables the constant range checking normally performed by the assembler/linker. This allows the **MVKL** instruction to be paired with the **MVKH** instruction (page 3-184) to generate 32-bit constants.

To load 32-bit constants, such as 1234 ABCDh, 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	.S

Instruction Type Single cycle

Delay Slots 0

See Also **MVK, MVKH, MVKLH**

Example 1

```
MVKL .S1   5678h,A8
```

	Before instruction		1 cycle after instruction
A8	xxxx xxxxh	A8	0000 5678h

Example 2

```
MVKL .S1   0C678h,A8
```

	Before instruction		1 cycle after instruction
A8	xxxx xxxxh	A8	FFFF C678h

NEG *Negate*

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

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

Compatibility C62x, C64x, C67x, and C67x+ CPU

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*	0 0 0 0 0	x	0 1 0 1 1 0	1	0	0	0	*s*	*p*
3	1	5	5		1						1	1

Opcode map field used...	For operand type...	Unit
*src*2 *dst*	xsint sint	.S1, .S2

Opcode .L unit

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	1	5	5		1	7				1	1

Opcode map field used...	For operand type...	Unit	Opfield
*src*2 *dst*	xsint sint	.L1, .L2	000 0110
*src*2 *dst*	slong slong	.L1, .L2	010 0100

Description The **NEG** pseudo-operation negates *src2* and places the result in *dst*. The assembler uses **SUB** (.unit) 0, *src2, dst* to perform this operation.

Execution

if (cond) 0 −*s src2* → *dst*
else nop

Instruction Type Single-cycle

Delay Slots 0

NOP — *No Operation*

Syntax **NOP** [*count*]

.unit = none

Compatibility C62x, C64x, C67x, and C67x+ CPU

Opcode

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

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

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

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

Execution No operation for *count* cycles

Instruction Type **NOP**

Delay Slots 0

Example 1

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

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

Example 2

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

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

NORM *Normalize Integer*

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

.unit = .L1 or .L2

Compatibility C62x, C64x, C67x, and C67x+ CPU

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	1	5	5		1	7				1	1

Opcode map field used...	For operand type...	Unit	Opfield
src2 *dst*	xsint uint	.L1, .L2	110 0011
src2 *dst*	slong uint	.L1, .L2	110 0000

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

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

src2

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

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

src2

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

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

src2

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

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

src2

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

Execution

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

Pipeline

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

Instruction Type Single-cycle

Delay Slots 0

Example 1

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

	Before instruction		1 cycle after instruction	
A1	02A3 469Fh	A1	02A3 469Fh	
A2	xxxx xxxxh	A2	0000 0005h	5

Example 2

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

	Before instruction		1 cycle after instruction	
A1	FFFF F25Ah	A1	FFFF F25Ah	
A2	xxxx xxxxh	A2	0000 0013h	19

NOT — *Bitwise NOT*

Syntax **NOT** (.unit) *src2, dst*

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

Compatibility C62x, C64x, C67x, and C67x+ CPU

Opcode .L unit

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

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

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*	1 1 1 1 1	x	0 0 1 0 1 0	1	0	0	0	*s*	*p*
3	1	5	5	5	1						1	1

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

Description The **NOT** pseudo-operation performs a bitwise **NOT** on the *src2* operand and places the result in *dst*. The assembler uses **XOR** (.unit) −1, *src2*, *dst* to perform this operation.

Execution if (cond) −1 XOR *src2* → *dst*
else nop

Instruction Type Single-cycle

Delay Slots 0

OR — *Bitwise OR*

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

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

Compatibility C62x, C64x, C67x, and C67x+ CPU

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				1	1

Opcode map field used...	For operand type...	Unit	Opfield
src1 *src2* *dst*	uint xuint uint	.L1, .L2	111 1111
src1 *src2* *dst*	scst5 xuint uint	.L1, .L2	111 1110

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

Opcode map field used...	For operand type...	Unit	Opfield
src1 *src2* *dst*	uint xuint uint	.S1, .S2	01 1011
src1 *src2* *dst*	scst5 xuint uint	.S1, .S2	01 1010

Description Performs a bitwise **OR** operation between *src1* and *src2*. The result is placed in *dst*. The *scst*5 operands are sign extended to 32 bits.

Execution

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

Pipeline

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

Instruction Type Single-cycle

Delay Slots 0

See Also **AND, XOR**

Example 1

```
OR .S1    A3,A4,A5
```

	Before instruction	1 cycle after instruction
A3	08A3 A49Fh	08A3 A49Fh
A4	00FF 375Ah	00FF 375Ah
A5	xxxx xxxxh	08FF B7DFh

Example 2

```
OR .L2    -12,B2,B8
```

	Before instruction	1 cycle after instruction
B2	0000 3A41h	0000 3A41h
B8	xxxx xxxxh	FFFF FFF5h

RCPDP *Double-Precision Floating-Point Reciprocal Approximation*

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

.unit = .S1 or .S2

Compatibility C67x and C67x+ CPU

Opcode

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

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

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:

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

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

Note:

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

Pipeline

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

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

Instruction Type 2-cycle DP

Delay Slots 1

Functional Unit Latency 1

See Also **RCPSP, RSQRDP**

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

Compatibility C67x and C67x+ CPU

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

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

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:

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

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

See Also **RCPDP, RSQRSP**

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

Compatibility C67x and C67x+ CPU

Opcode

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

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

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:

$$x[n + 1] = x[n](1.5 - (v/2) \times x[n] \times x[n])$$

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

See Also **RCPDP, RSQRSP**

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

Compatibility C67x and C67x+ CPU

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

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

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:

$$x[n + 1] = x[n](1.5 - (v/2) \times x[n] \times x[n])$$

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

Note:

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

See Also **RCPSP, RSQRDP**

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

SADD — *Add Two Signed Integers With Saturation*

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

.unit = .L1 or .L2

Compatibility C62x, C64x, C67x, and C67x+ CPU

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				1	1

Opcode map field used...	For operand type...	Unit	Opfield
src1 *src2* *dst*	sint xsint sint	.L1, .L2	001 0011
src1 *src2* *dst*	xsint slong slong	.L1, .L2	011 0001
src1 *src2* *dst*	scst5 xsint sint	.L1, .L2	001 0010
src1 *src2* *dst*	scst5 slong slong	.L1, .L2	011 0000

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

See Also **ADD, SSUB**

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

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

1 cycle after instruction

Register			
A5:A4	0000 0000h	7C83 39B1h	
A7:A6	0000 0000h	8DAD 7953h	2376956243†
B2	112A 3FA2h		
CSR	0001 0100h		

2 cycles after instruction

Register		
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

Compatibility C62x, C64x, C67x, and C67x+ CPU

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	1	5	5		1					1	1

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

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

Execution

```
if (cond)    {
             if (src2 > (2^31 − 1) )
                 (2^31 − 1) → dst
             else if (src2 < −2^31)
                 −2^31 → dst
             else src2_31..0 → dst
             }
else nop
```

Pipeline

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

Instruction Type Single-cycle

Delay Slots 0

Example 1 `SAT .L2 B1:B0,B5`

	Before instruction		1 cycle after instruction		2 cycles after instruction	
B1:B0	0000 001Fh | 3413 539Ah	B1:B0	0000 001Fh | 3413 539Ah	B1:B0	0000 001Fh | 3413 539Ah	
B5	xxxx xxxxh	B5	7FFF FFFFh	B5	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

Compatibility

C62x, C64x, C67x, and C67x+ CPU

Opcode

Constant form:

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

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

Opcode

Register form:

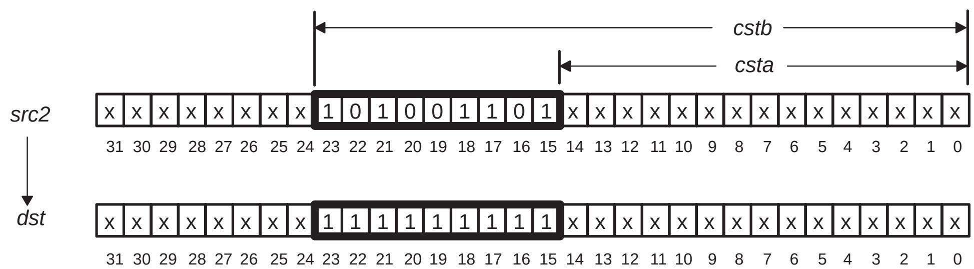
31–29	28	27–23	22–18	17–13	12	11–6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	1 1 1 0 1 1	1	0	0	0	*s*	*p*
3	1	5	5	5	1						1	1

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

Description

For *cstb* > *csta*, the field in *src2* as specified by *csta* to *cstb* is set to all 1s in *dst*. The *csta* and *cstb* operands may be specified as constants or in the 10 LSBs of the *src1* register, with *cstb* being bits 0−4 ($src1_{4..0}$) and *csta* being bits 5−9 ($src1_{9..5}$). *csta* is the LSB of the field and *cstb* is 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 in *dst*. The LSB location of *src2* is bit 0 and the MSB location of *src2* is bit 31.

In the following example, *csta* is 15 and *cstb* is 23. For the register version of the instruction, only the 10 LSBs of the *src1* register are valid. If any of the 22 MSBs are non-zero, the result is invalid.



For *cstb* < *csta*, the *src2* register is copied to *dst*. The *csta* and *cstb* operands may be specified as constants or in the 10 LSBs of the *src1* register, with *cstb* being bits 0−4 ($src1_{4..0}$) and *csta* being bits 5−9 ($src1_{9..5}$).

Execution

If the constant form is used when *cstb* > *csta*:

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

If the register form is used when *cstb* > *csta*:

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

See Also **CLR**

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 0197h	B1	0000 0197h
B2	xxxx xxxxh	B2	9EFF FA31h

SHL — *Arithmetic Shift Left*

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

.unit = .S1 or .S2

Compatibility C62x, C64x, C67x, and C67x+ CPU

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					1	1

Opcode map field used...	For operand type...	Unit	Opfield
src2 *src1* *dst*	xsint uint sint	.S1, .S2	11 0011
src2 *src1* *dst*	slong uint slong	.S1, .S2	11 0001
src2 *src1* *dst*	xuint uint ulong	.S1, .S2	01 0011
src2 *src1* *dst*	xsint ucst5 sint	.S1, .S2	11 0010
src2 *src1* *dst*	slong ucst5 slong	.S1, .S2	11 0000
src2 *src1* *dst*	xuint ucst5 ulong	.S1, .S2	01 0010

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

See Also **SHR, SSHL**

Example 1

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

	Before instruction	1 cycle after instruction
A0	29E3 D31Ch	29E3 D31Ch
A1	xxxx xxxxh	9E3D 31C0h

Example 2

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

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

Example 3

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

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

SHR — *Arithmetic Shift Right*

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

.unit = .S1 or .S2

Compatibility C62x, C64x, C67x, and C67x+ CPU

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					1	1

Opcode map field used...	For operand type...	Unit	Opfield
src2 *src1* *dst*	xsint uint sint	.S1, .S2	11 0111
src2 *src1* *dst*	slong uint slong	.S1, .S2	11 0101
src2 *src1* *dst*	xsint ucst5 sint	.S1, .S2	11 0110
src2 *src1* *dst*	slong ucst5 slong	.S1, .S2	11 0100

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 value 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

See Also **SHL, SHRU**

Example 1

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

	Before instruction	1 cycle after instruction
A0	F123 63D1h	F123 63D1h
A1	xxxx xxxxh	FFF1 2363h

Example 2

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

	Before instruction	1 cycle after instruction
B0	1492 5A41h	1492 5A41h
B1	0000 0012h	0000 0012h
B2	xxxx xxxxh	0000 0524h

Example 3

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

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

SHRU *Logical Shift Right*

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

.unit = .S1 or .S2

Compatibility C62x, C64x, C67x, and C67x+ CPU

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					1	1

Opcode map field used...	For operand type...	Unit	Opfield
src2 *src1* *dst*	xuint uint uint	.S1, .S2	10 0111
src2 *src1* *dst*	ulong uint ulong	.S1, .S2	10 0101
src2 *src1* *dst*	xuint ucst5 uint	.S1, .S2	10 0110
src2 *src1* *dst*	ulong ucst5 ulong	.S1, .S2	10 0100

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 value 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

See Also **SHL, SHR**

Example

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

	Before instruction	1 cycle after instruction
A0	`F123 63D1h`	`F123 63D1h`
A1	`xxxx xxxxh`	`00F1 2363h`

SMPY — *Multiply Signed 16 LSB × Signed 16 LSB With Left Shift and Saturation*

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

.unit = .M1 or .M2

Compatibility C62x, C64x, C67x, and C67x+ CPU

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

Opcode map field used...	For operand type...	Unit
src1	slsb16	.M1, .M2
src2	xslsb16	
dst	sint	

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 8000 0000h, then the result is saturated to 7FFF FFFFh. If a saturate occurs, the SAT bit in CSR is set one cycle after *dst* is written. The source operands are signed by default.

Execution

```
if (cond)   {
            if (((src1 × src2) << 1) != 8000 0000h)
                ((src1 × src2) << 1) → dst
            else
                7FFF FFFFh → 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

See Also **MPY, SMPYH, SMPYHL, SMPYLH**

Example

```
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

‡ Signed 16-LSB integer

SMPYH — *Multiply Signed 16 MSB × Signed 16 MSB With Left Shift and Saturation*

Syntax

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

.unit = .M1 or .M2

Compatibility

C62x, C64x, C67x, and C67x+ CPU

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

Opcode map field used...	For operand type...	Unit
src1	smsb16	.M1, .M2
src2	xsmsb16	
dst	sint	

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 8000 0000h, then the result is saturated to 7FFF FFFFh. If a saturation occurs, the SAT bit in CSR is set one cycle after *dst* is written. The source operands are signed by default.

Execution

```
if (cond)     {
              if (((src1 × src2) << 1) != 8000 0000h)
                   ((src1 × src2) << 1) → dst
              else
                   7FFF FFFFh → 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

See Also

MPYH, SMPY, SMPYHL, SMPYLH

SMPYHL — *Multiply Signed 16 MSB × Signed 16 LSB With Left Shift and Saturation*

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

.unit = .M1 or .M2

Compatibility C62x, C64x, C67x, and C67x+ CPU

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

Opcode map field used...	For operand type...	Unit
src1	smsb16	.M1, .M2
src2	xslsb16	
dst	sint	

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 8000 0000h, then the result is saturated to 7FFF FFFFh. If a saturation occurs, the SAT bit in CSR is set one cycle after *dst* is written.

Execution

```
if (cond)   {
            if (((src1 × src2) << 1) != 8000 0000h)
                ((src1 × src2) << 1) → dst
            else
                7FFF FFFFh → 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

See Also **MPYHL, SMPY, SMPYH, SMPYLH**

Example

```
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

SMPYLH — *Multiply Signed 16 LSB × Signed 16 MSB With Left Shift and Saturation*

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

.unit = .M1 or .M2

Compatibility C62x, C64x, C67x, and C67x+ CPU

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

Opcode map field used...	For operand type...	Unit
src1	slsb16	.M1, .M2
src2	xsmsb16	
dst	sint	

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 8000 0000h, then the result is saturated to 7FFF FFFFh. If a saturation occurs, the SAT bit in CSR is set one cycle after *dst* is written.

Execution

```
if (cond)   {
            if (((src1 × src2) << 1) != 8000 0000h)
                ((src1 × src2) << 1) → dst
            else
                7FFF FFFFh → 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

See Also **MPYLH, SMPY, SMPYH, SMPYHL**

Example

```
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

SPDP — *Convert Single-Precision Floating-Point Value to Double-Precision Floating-Point Value*

Syntax

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

.unit = .S1 or .S2

Compatibility

C67x and C67x+ CPU

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	1	5	5		1						1	1

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

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

See Also **DPSP, INTDP, SPINT, SPTRUNC**

Example

```
SPDP .S1X  B2,A1:A0
```

Before instruction

B2	4109 999Ah		8.6
A1:A0	xxxx xxxxh	xxxx xxxxh	

2 cycles after instruction

B2	4109 999Ah		8.6
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

Compatibility C67x and C67x+ CPU

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	1	5	5		1					1	1

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

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

See Also **DPINT, INTSP, SPDP, SPTRUNC**

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

Compatibility C67x and C67x+ CPU

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	1	5	5		1					1	1

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

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

See Also **DPTRUNC, SPDP, SPINT**

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

SSHL — *Shift Left With Saturation*

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

.unit = .S1 or .S2

Compatibility C62x, C64x, C67x, and C67x+ CPU

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					1	1

Opcode map field used...	For operand type...	Unit	Opfield
src2 *src1* *dst*	xsint uint sint	.S1, .S2	10 0011
src2 *src1* *dst*	xsint ucst5 sint	.S1, .S2	10 0010

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 CSR is set one cycle after *dst* is written.

Execution

```
if (cond)    {
             if ( bit(31) through bit(31−src1) of src2 are all 1s or all 0s)
                   dst = src2 << src1;
             else if (src2 > 0)
                   saturate dst to 7FFF FFFFh;
             else if (src2 < 0)
                   saturate dst to 8000 0000h;
             }
else nop
```

Pipeline

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

Instruction Type Single-cycle

Delay Slots 0

See Also **SHL, SHR**

Example 1

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

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

Example 2

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

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

SSUB — *Subtract Two Signed Integers With Saturation*

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

.unit = .L1 or .L2

Compatibility C62x, C64x, C67x, and C67x+ CPU

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				1	1

Opcode map field used...	For operand type...	Unit	Opfield
src1 *src2* *dst*	sint xsint sint	.L1, .L2	000 1111
src1 *src2* *dst*	xsint sint sint	.L1, .L2	001 1111
src1 *src2* *dst*	scst5 xsint sint	.L1, .L2	000 1110
src1 *src2* *dst*	scst5 slong slong	.L1, .L2	010 1100

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 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

See Also **SUB**

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 — *Store Byte to Memory With a 5-Bit Unsigned Constant Offset or Register Offset*

Syntax

Register Offset	Unsigned Constant Offset
STB (.unit) *src*, *+*baseR[offsetR]*	**STB** (.unit) *src*, *+*baseR[ucst5]*

.unit = .D1 or .D2

Compatibility C62x, C64x, C67x, and C67x+ CPU

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*	0	*y*	0 1 1	0	1	*s*	*p*
3	1	5	5	5	4		1				1	1

Description Stores a byte to memory from a general-purpose register (*src*). Table 3−11 (page 3-32) 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 bits. 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 2.7.3, page 2-10).

For **STB**, the 8 LSBs of the *src* register are stored. *src* can be in either register file, regardless of the .D unit or *baseR* or *offsetR* used. The *s* bit determines which file *src* 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 cleared to 0.

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 0. Parentheses, ( ), can be used to set a nonscaled, constant offset. You must type either brackets or parentheses around the specified offset, if you use the optional offset parameter.

Execution

if (cond) *src* → mem
else nop

Pipeline

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

Instruction Type Store

Delay Slots 0

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

See Also **STH, STW**

Example

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

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

STB — *Store Byte to Memory With a 15-Bit Unsigned Constant Offset*

Syntax **STB** (.unit) *src*, *+B14/B15[*ucst15*]

.unit = .D2

Compatibility C62x, C64x, C67x, and C67x+ CPU

Opcode

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

Description Stores a byte to memory from a general-purpose register (*src*). The memory address is formed from a base address register 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 executes only on the .D2 unit.

The offset, *ucst15*, is scaled by a left-shift of 0 bits. 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**, the 8 LSBs of the *src* register are stored. *src* can be in either register file. The *s* bit determines which file *src* 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 0. Parentheses, ( ), can be used to set a nonscaled, constant offset. You must type either brackets or parentheses around the specified offset, if you use the optional offset parameter.

Execution

```
if (cond)    src → mem
else nop
```

Note:

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

Pipeline

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

Instruction Type Store

Delay Slots 0

See Also **STH, STW**

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

STH — Store Halfword to Memory With a 5-Bit Unsigned Constant Offset or Register Offset

Syntax

Register Offset	Unsigned Constant Offset
STH (.unit) *src*, *+*baseR[offsetR]*	**STH** (.unit) *src*, *+*baseR[ucst5]*

.unit = .D1 or .D2

Compatibility C62x, C64x, C67x, and C67x+ CPU

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*	0	*y*	1 0 1	0	1	*s*	*p*
3	1	5	5	5	4		1				1	1

Description Stores a halfword to memory from a general-purpose register (*src*). Table 3−11 (page 3-32) 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 1 bit. 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 2.7.3, page 2-10).

For **STH**, the 16 LSBs of the *src* register are stored. *src* can be in either register file, regardless of the .D unit or *baseR* or *offsetR* used. The *s* bit determines which file *src* 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 cleared to 0.

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 1. Parentheses, ( ), can be used to set a nonscaled, constant offset. You must type either brackets or parentheses around the specified offset, if you use the optional offset parameter.

Halfword addresses must be aligned on halfword (LSB is 0) boundaries.

Execution

if (cond) *src* → mem
else nop

Pipeline

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

Instruction Type Store

Delay Slots 0

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

See Also **STB, STW**

Example 1

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

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

Example 2

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

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

STH — *Store Halfword to Memory With a 15-Bit Unsigned Constant Offset*

Syntax

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

.unit = .D2

Compatibility

C62x, C64x, C67x, and C67x+ CPU

Opcode

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

Description

Stores a halfword to memory from a general-purpose register (*src*). The memory address is formed from a base address register 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 executes only on the .D2 unit.

The offset, *ucst15*, is scaled by a left-shift of 1 bit. 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 **STH**, the 16 LSBs of the *src* register are stored. *src* can be in either register file. The *s* bit determines which file *src* 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 1. Parentheses, ( ), can be used to set a nonscaled, constant offset. You must type either brackets or parentheses around the specified offset, if you use the optional offset parameter.

Halfword addresses must be aligned on halfword (LSB is 0) boundaries.

Execution

```
if (cond)    src → mem
else nop
```

Note:

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

Pipeline

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

Instruction Type Store

Delay Slots 0

See Also **STB, STW**

STW — *Store Word to Memory With a 5-Bit Unsigned Constant Offset or Register Offset*

Syntax

Register Offset	Unsigned Constant Offset
STW (.unit) *src*, *+*baseR[offsetR]*	**STW** (.unit) *src*, *+*baseR[ucst5]*

.unit = .D1 or .D2

Compatibility C62x, C64x, C67x, and C67x+ CPU

Opcode

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

Description Stores a word to memory from a general-purpose register (*src*). Table 3−11 (page 3-32) 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 2 bits. 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 2.7.3, page 2-10).

For **STW**, the entire 32-bits of the *src* register are stored. *src* can be in either register file, regardless of the .D unit or *baseR* or *offsetR* used. The *s* bit determines which file *src* 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 cleared to 0.

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. Parentheses, ( ), can be used to set a nonscaled, constant offset. For example, **STW** (.unit) *src*, *+*baseR*(12) represents an offset of 12 bytes; whereas, **STW** (.unit) *src*, *+*baseR*[12] 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 addresses must be aligned on word (two LSBs are 0) boundaries.

Execution

if (cond) *src* $\rightarrow$ mem
else nop

Pipeline

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

Instruction Type Store

Delay Slots 0

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

See Also **STB, STH**

Example

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

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

STW — *Store Word to Memory With a 15-Bit Unsigned Constant Offset*

Syntax

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

.unit = .D2

Compatibility

C62x, C64x, C67x, and C67x+ CPU

Opcode

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

Description

Stores a word to memory from a general-purpose register (*src*). The memory address is formed from a base address register 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 executes only on the .D2 unit.

The offset, *ucst15*, is scaled by a left-shift of 2 bits. 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 **STW**, the entire 32-bits of the *src* register are stored. *src* can be in either register file. The *s* bit determines which file *src* 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. Parentheses, ( ), can be used to set a nonscaled, constant offset. For example, **STW** (.unit) *src*, *+B14/B15(60) represents an offset of 12 bytes; whereas, **STW** (.unit) *src*, *+B14/B15[60] 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 addresses must be aligned on word (two LSBs are 0) boundaries.

Execution

if (cond) *src* → mem
else nop

> **Note:**
>
> This instruction executes only on the B side (.D2).

Pipeline

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

Instruction Type Store

Delay Slots 0

See Also **STB, STH**

SUB — *Subtract Two Signed Integers Without Saturation*

Syntax

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

or

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

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

Compatibility

C62x, C64x, C67x, and C67x+ CPU

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				1	1

Opcode map field used...	For operand type...	Unit	Opfield
src1 *src2* *dst*	sint xsint sint	.L1, .L2	000 0111
src1 *src2* *dst*	xsint sint sint	.L1, .L2	001 0111
src1 *src2* *dst*	sint xsint slong	.L1, .L2	010 0111
src1 *src2* *dst*	xsint sint slong	.L1, .L2	011 0111
src1 *src2* *dst*	scst5 xsint sint	.L1, .L2	000 0110
src1 *src2* *dst*	scst5 slong slong	.L1, .L2	010 0100

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

Opcode map field used...	For operand type...	Unit	Opfield
src1 *src2* *dst*	sint xsint sint	.S1, .S2	01 0111
src1 *src2* *dst*	scst5 xsint sint	.S1, .S2	01 0110

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

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

Opcode map field used...	For operand type...	Unit	Opfield
src2 *src1* *dst*	sint sint sint	.D1, .D2	01 0001
src2 *src1* *dst*	sint ucst*5* sint	.D1, .D2	01 0011

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 (.unit) *src1*, *scst5*, *dst* is encoded as **ADD** (.unit) −*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

See Also **ADD, SSUB, SUBC, SUBDP, SUBSP, SUBU, SUB2**

Example

```
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

SUBAB *Subtract Using Byte Addressing Mode*

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

.unit = .D1 or .D2

Compatibility C62x, C64x, C67x, and C67x+ CPU

Opcode

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

Opcode map field used...	For operand type...	Unit	Opfield
src2 *src1* *dst*	sint sint sint	.D1, .D2	11 0001
src2 *src1* *dst*	sint ucst5 sint	.D1, .D2	11 0011

Description *src1* is subtracted from *src2* using the byte addressing mode specified for *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 2.7.3, page 2-10). 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

See Also **SUB, SUBAH, SUBAW**

Example

```
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
```

SUBAH — *Subtract Using Halfword Addressing Mode*

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

.unit = .D1 or .D2

Compatibility C62x, C64x, C67x, and C67x+ CPU

Opcode

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

Opcode map field used...	For operand type...	Unit	Opfield
src2 *src1* *dst*	sint sint sint	.D1, .D2	11 0101
src2 *src1* *dst*	sint ucst5 sint	.D1, .D2	11 0111

Description *src1* is subtracted from *src2* using the halfword addressing mode specified for *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 2.7.3, page 2-10). *src1* is left shifted by 1. 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

See Also **SUB, SUBAB, SUBAW**

SUBAW — *Subtract Using Word Addressing Mode*

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

.unit = .D1 or .D2

Compatibility C62x, C64x, C67x, and C67x+ CPU

Opcode

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

Opcode map field used...	For operand type...	Unit	Opfield
src2 *src1* *dst*	sint sint sint	.D1, .D2	11 1001
src2 *src1* *dst*	sint ucst5 sint	.D1, .D2	11 1011

Description *src1* is subtracted from *src2* using the word addressing mode specified for *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 2.7.3, page 2-10). *src1* is left shifted by 2. 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

See Also **SUB, SUBAB, SUBAH**

Example

```
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 — *Subtract Conditionally and Shift—Used for Division*

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

.unit = .L1 or .L2

Compatibility C62x, C64x, C67x, and C67x+ CPU

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	1	5	5	5	1					1	1

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

Description Subtract *src2* from *src1*. If result is greater 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 *src1* by 1, and place it in *dst*. This step is commonly used in division.

Execution

```
if (cond)    {
             if (src1 − src2 ≥ 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

See Also **ADD, SSUB, SUB, SUBDP, SUBSP, SUBU, SUB2**

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	

SUBDP — *Subtract Two Double-Precision Floating-Point Values*

Syntax

SUBDP (.unit) *src1*, *src2*, *dst* (C67x and C67x+ CPU)
.unit = .L1 or .L2
or
SUBDP (.unit) *src1*, *src2*, *dst* (C67x+ CPU only)
.unit = .S1 or .S2

Compatibility C67x and C67x+ CPU

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				1	1

Opcode map field used...	For operand type...	Unit	Opfield
src1 *src2* *dst*	dp xdp dp	.L1, .L2	001 1001
src1 *src2* *dst*	xdp dp dp	.L1, .L2	001 1101
src1 *src2* *dst*	dp xdp dp	.S1, .S2	111 0011
src1 *src2* *dst*	dp xdp dp	.S1, .S2	111 0111 src2 − src1

Note:

The assembly syntax allows a cross-path operand to be used for either *src1* or *src2*. The assembler selects between the two opcodes based on which source operand in the assembly instruction requires the cross path. If *src1* requires the cross path, the assembler chooses the second (reverse) form of the instruction syntax and reverses the order of the operands in the encoded instruction.

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

Execution

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

Notes:

1) This instruction takes the rounding mode from and sets the warning bits in FADCR, not FAUCR as for other .S unit instructions.

2) The source specific warning bits set in FADCR are set according to the registers sources in the actual machine instruction and not according to the order of the sources in the assembly form.

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

4) If one source is SNaN or QNaN, the result is NaN_out. If either source is SNaN, the INVAL bit is set also.

5) If both sources are +infinity or −infinity, the result is NaN_out and the INVAL bit is set.

6) 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.

7) 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

8) 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

9) 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.

10) 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*.

11) 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 or .S	.L or .S					

For the C67x CPU, 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.

For the C67x+ CPU, the low half of the result is written out one cycle earlier than the high half. If *dst* is used as the source for the **ADDDP**, **CMPEQDP**, **CMPLTDP**, **CMPGTDP**, **MPYDP**, **MPYSPDP**, **MPYSP2DP**, 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

See Also **ADDDP, SUB, SUBSP, SUBU**

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 *Subtract Two Single-Precision Floating-Point Values*

Syntax

SUBSP (.unit) *src1*, *src2*, *dst* (C67x and C67x+ CPU)
.unit = .L1 or .L2
or
SUBSP (.unit) *src1*, *src2*, *dst* (C67x+ CPU only)
.unit = .S1 or .S2

Compatibility C67x and C67x+ CPU

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				1	1

Opcode map field used...	For operand type...	Unit	Opfield
src1 *src2* *dst*	sp xsp sp	.L1, .L2	001 0001
src1 *src2* *dst*	xsp sp sp	.L1, .L2	001 0101
src1 *src2* *dst*	sp xsp sp	.S1, .S2	111 0001
src1 *src2* *dst*	sp xsp sp	.S1, .S2	111 0101 src2 − src1

Note:

The assembly syntax allows a cross-path operand to be used for either *src1* or *src2*. The assembler selects between the two opcodes based on which source operand in the assembly instruction requires the cross path. If *src1* requires the cross path, the assembler chooses the second (reverse) form of the instruction syntax and reverses the order of the operands in the encoded instruction.

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

Execution

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

Notes:

1) This instruction takes the rounding mode from and sets the warning bits in FADCR, not FAUCR as for other .S unit instructions.
2) The source specific warning bits set in FADCR are set according to the registers sources in the actual machine instruction and not according to the order of the sources in the assembly form.
3) If rounding is performed, the INEX bit is set.
4) If one source is SNaN or QNaN, the result is NaN_out. If either source is SNaN, the INVAL bit is set also.
5) If both sources are +infinity or −infinity, the result is NaN_out and the INVAL bit is set.
6) 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.
7) 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

8) 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

9) 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.
10) 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*.
11) 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

See Also **ADDSP, SUB, SUBDP, SUBU**

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

SUBU — *Subtract Two Unsigned Integers Without Saturation*

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

.unit = .L1 or .L2

Compatibility C62x, C64x, C67x, and C67x+ CPU

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				1	1

Opcode map field used...	For operand type...	Unit	Opfield
src1 *src2* *dst*	uint xuint ulong	.L1, .L2	010 1111
src1 *src2* *dst*	xuint uint ulong	.L1, .L2	011 1111

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

Execution

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

Pipeline

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

Instruction Type Single-cycle

Delay Slots 0

See Also **ADDU, SSUB, SUB, SUBC, SUBDP, SUBSP, SUB2**

Example

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

Before instruction

A1	0000 325Ah		12810†
A2	FFFF FF12h		4294967058†
A5:A4	xxxx xxxxh	xxxx xxxxh	

1 cycle after instruction

A1	0000 325Ah		
A2	FFFF FF12h		
A5:A4	0000 00FFh	0000 3348h	−4294954168‡

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

SUB2 — *Subtract Two 16-Bit Integers on Upper and Lower Register Halves*

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

.unit = .S1 or .S2

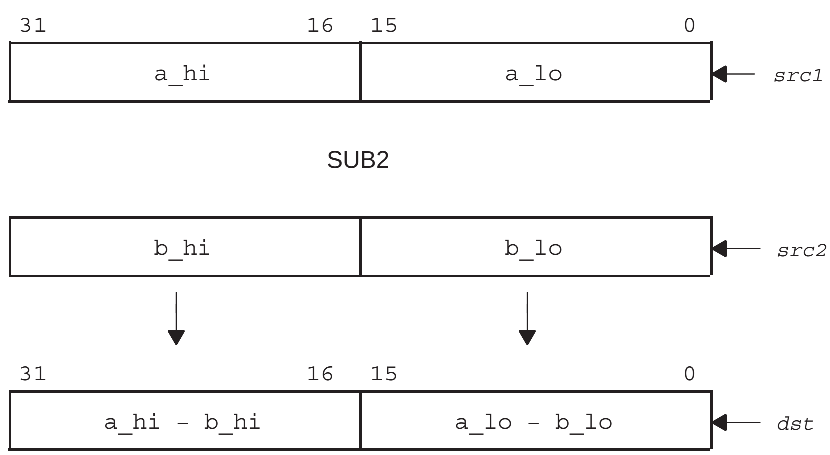
Compatibility C62x, C64x, C67x, and C67x+ CPU

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	1	5	5	5	1						1	1

Opcode map field used...	For operand type...	Unit
src1 *src*2 *dst*	sint xsint sint	.S1, .S2

Description 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*.



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	.S

Instruction Type Single-cycle

Delay Slots 0

See Also **ADD2, SSUB, SUB, SUBC, SUBU**

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 .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 — *Bitwise Exclusive OR*

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

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

Compatibility C62x, C64x, C67x, and C67x+ CPU

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				1	1

Opcode map field used...	For operand type...	Unit	Opfield
src1 *src2* *dst*	uint xuint uint	.L1, .L2	110 1111
src1 *src2* *dst*	scst5 xuint uint	.L1, .L2	110 1110

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

Opcode map field used...	For operand type...	Unit	Opfield
src1 *src2* *dst*	uint xuint uint	.S1, .S2	00 1011
src1 *src2* *dst*	scst5 xuint uint	.S1, .S2	00 1010

Description Performs a bitwise exclusive-OR (**XOR**) operation 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

See Also **AND, OR**

Example 1

```
XOR .S1    A3, A4, A5
```

	Before instruction	1 cycle after instruction
A3	0721 325Ah	0721 325Ah
A4	0019 0F12h	0019 0F12h
A5	xxxx xxxxh	0738 3D48h

Example 2

```
XOR .L2    B1, 0dh, B8
```

	Before instruction	1 cycle after instruction
B1	0000 1023h	0000 1023h
B8	xxxx xxxxh	0000 102Eh

ZERO — *Zero a Register*

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

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

Compatibility C62x, C64x, C67x, and C67x+ CPU

Opcode

Opcode map field used...	For operand type...	Unit	Opfield
dst	sint	.L1, .L2	001 0111
dst	slong	.L1, .L2	011 0111
dst	sint	.D1, .D2	01 0001
dst	sint	.S1, .S2	01 0111

Description The **ZERO** pseudo-operation fills the *dst* register with 0s by subtracting the *dst* from itself and placing the result in the *dst*.

In the case where *dst* is sint, the assembler uses the **MVK** (.unit) 0, *dst* instruction.

In the case where *dst* is slong, the assembler uses the **SUB** (.unit) *src1, src2, dst* instruction.

Execution if (cond) *dst* − *dst* → *dst*
else nop

Instruction Type Single-cycle

Delay Slots 0

Example

```
ZERO .D1   A1
```

	Before instruction	1 cycle after instruction
A1	B174 6CA1h	0000 0000h

Chapter 4

Pipeline

The C67x 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.

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**

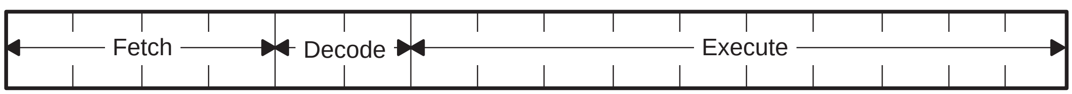
4.1 Pipeline Operation Overview

The pipeline phases are divided into three stages:

- ❑ Fetch
- ❑ Decode
- ❑ Execute

All instructions in the C67x DSP 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 DSP pipeline are shown in Figure 4−1.

Figure 4−1. Pipeline Stages



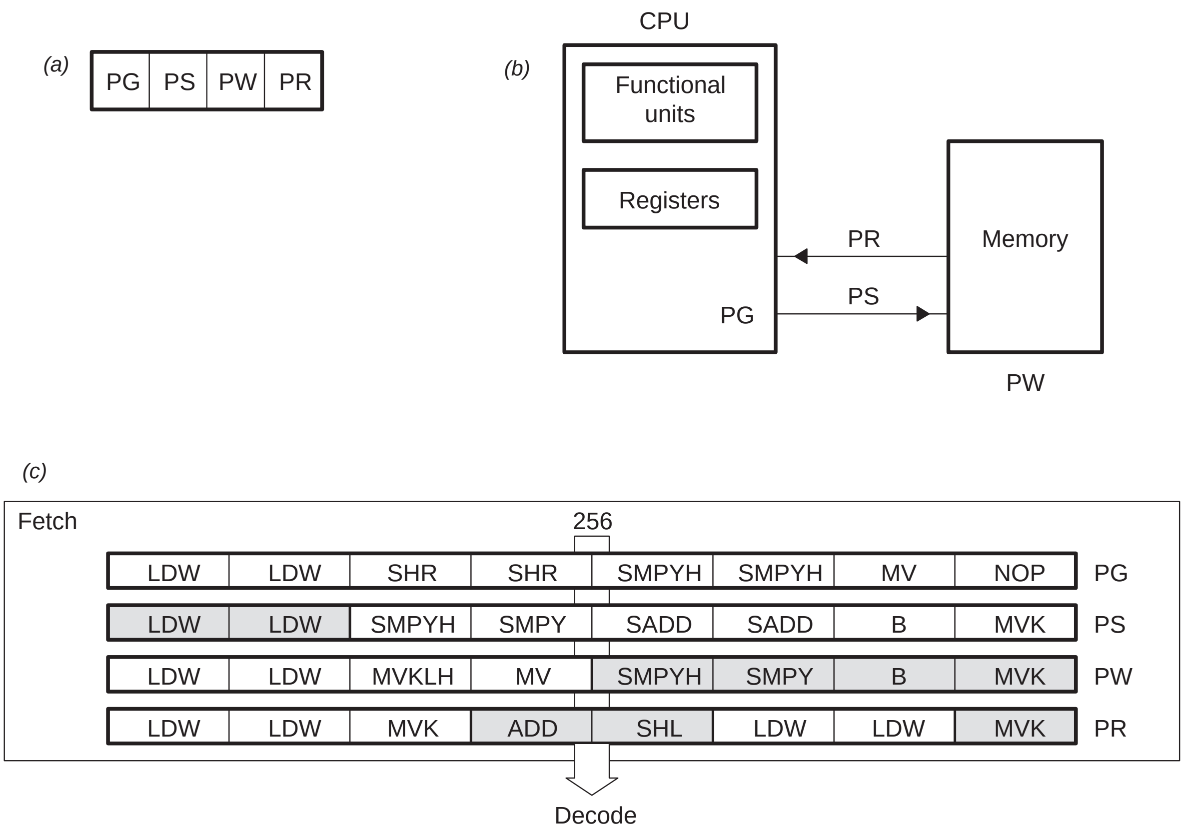
4.1.1 Fetch

The fetch phases of the pipeline are:

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

The C67x DSP uses a fetch packet (FP) of eight instructions. All eight of the instructions proceed through fetch processing together, through the PG, PS, PW, and PR phases. Figure 4−2(a) shows the fetch phases in sequential order from left to right. Figure 4−2(b) 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 4−2(c) shows fetch packets flowing through the phases of the fetch stage of the pipeline. In Figure 4−2(c), the first fetch packet (in PR) is made up of 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 4−2. Fetch Phases of the Pipeline



4.1.2 Decode

The decode phases of the pipeline are:

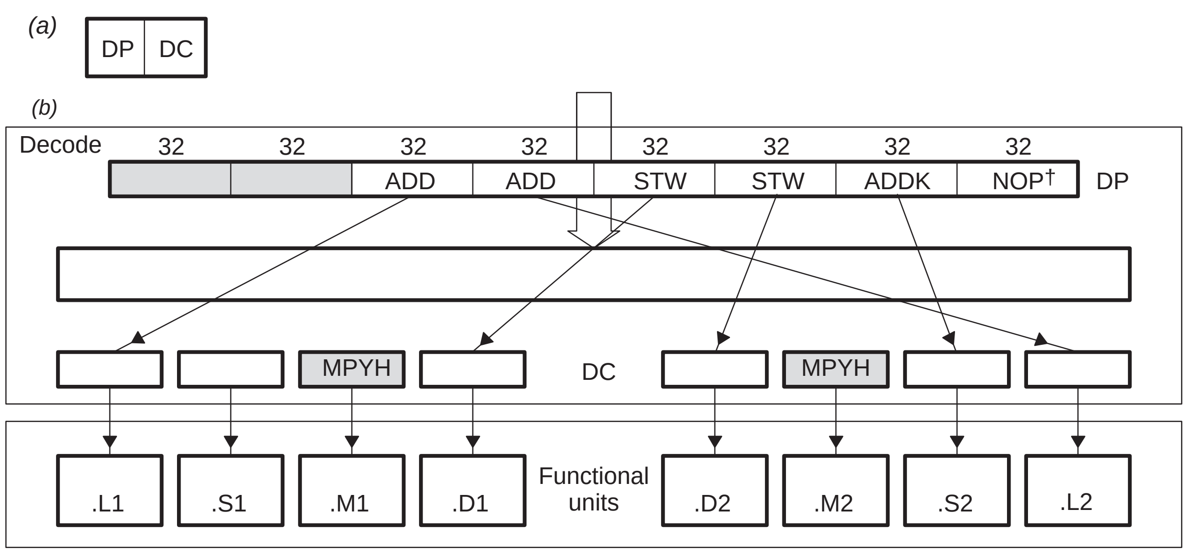
- **DP:** Instruction dispatch
- **DC:** Instruction decode

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

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

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

Figure 4−3. Decode Phases of the Pipeline



† NOP is not dispatched to a functional unit.

4.1.3 Execute

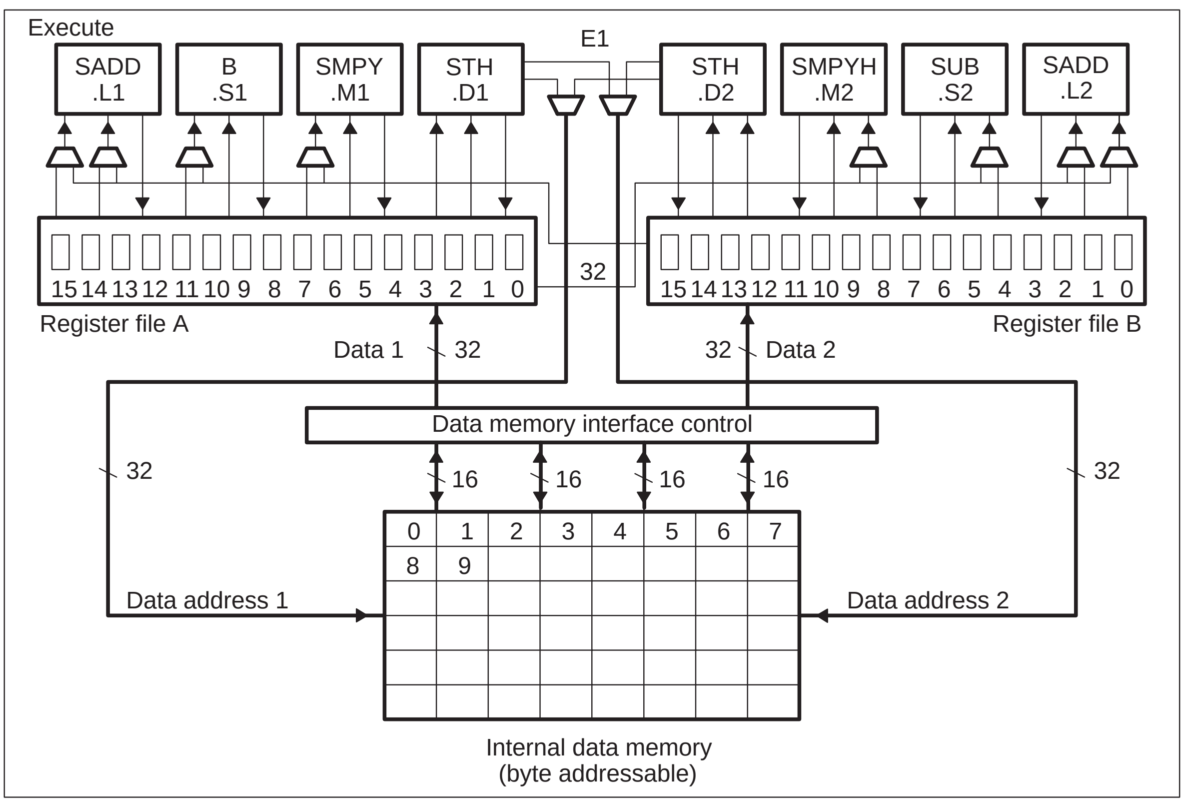
The execute portion of the pipeline is subdivided into ten phases (E1−E10), as compared to the five phases in a fixed-point pipeline. Different types of instructions require different numbers of these phases to complete their execution. These phases of the pipeline play an important role in your understanding the device state at CPU cycle boundaries. The execution of different types of instructions in the pipeline is described in section 4.2, *Pipeline Execution of Instruction Types*. Figure 4−4(a) shows the execute phases of the pipeline in sequential order from left to right. Figure 4−4(b) shows the portion of the functional block diagram in which execution occurs.

Figure 4−4. Execute Phases of the Pipeline

(a)

E1	E2	E3	E4	E5	E6	E7	E8	E9	E10

(b)



4.1.4 Pipeline Operation Summary

Figure 4−5 shows all the phases in each stage of the C67x DSP pipeline in sequential order, from left to right.

Figure 4−5. Pipeline Phases

Fetch				Decode		Execute									
PG	PS	PW	PR	DP	DC	E1	E2	E3	E4	E5	E6	E7	E8	E9	E10

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

For example, examine cycle 7 in Figure 4−6. When the instructions from FPn reach E1, the instructions in the execute packet from FPn +1 are being decoded. 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 4.4, page 4-56, for additional detail on code flowing through the pipeline. Table 4−1 summarizes the pipeline phases and what happens in each phase.

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

	Clock cycle																
Fetch packet	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**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 4−1. Operations Occurring During 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 MPYSPDP instruction, the *src1* and the lower 32 bits of *src2* 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 4−1. Operations Occurring During 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 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.[†] For MPYSPDP instruction, the *src1* and the upper 32 bits of *src2* 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 MPYI and MPYID instructions, the sources are read.[†] For 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 and MPYSP2DP instructions, the lower 32 bits of the result are written to a register file.[†]	4-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 4−1. Operations Occurring During Pipeline Phases (Continued)

Stage	Phase	Symbol	During This Phase	Instruction Type Completed
	Execute 5	E5	For load instructions, data is written into a register file.†	Load INTDP
			For INTDP and MPYSP2DP instructions, the upper 32 bits of the result are written to a register file.†	MPYSP2DP
	Execute 6	E6	For ADDDP/SUBDP and MPYSPDP instructions, the lower 32 bits of the result are written to a register file.†	ADDDP/ SUBDP, MPYSPDP
	Execute 7	E7	For ADDDP/SUBDP and MPYSPDP instructions, the upper 32 bits of the result are written to a register file.†	ADDDP/ SUBDP, MPYSPDP
	Execute 8	E8	Nothing is read or written.	
	Execute 9	E9	For MPYI instruction, the result is written to a register file.†	MPYI
			For MPYDP and MPYID instructions, the lower 32 bits of the result are written to a register file.†	MPYDP MPYID
	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.

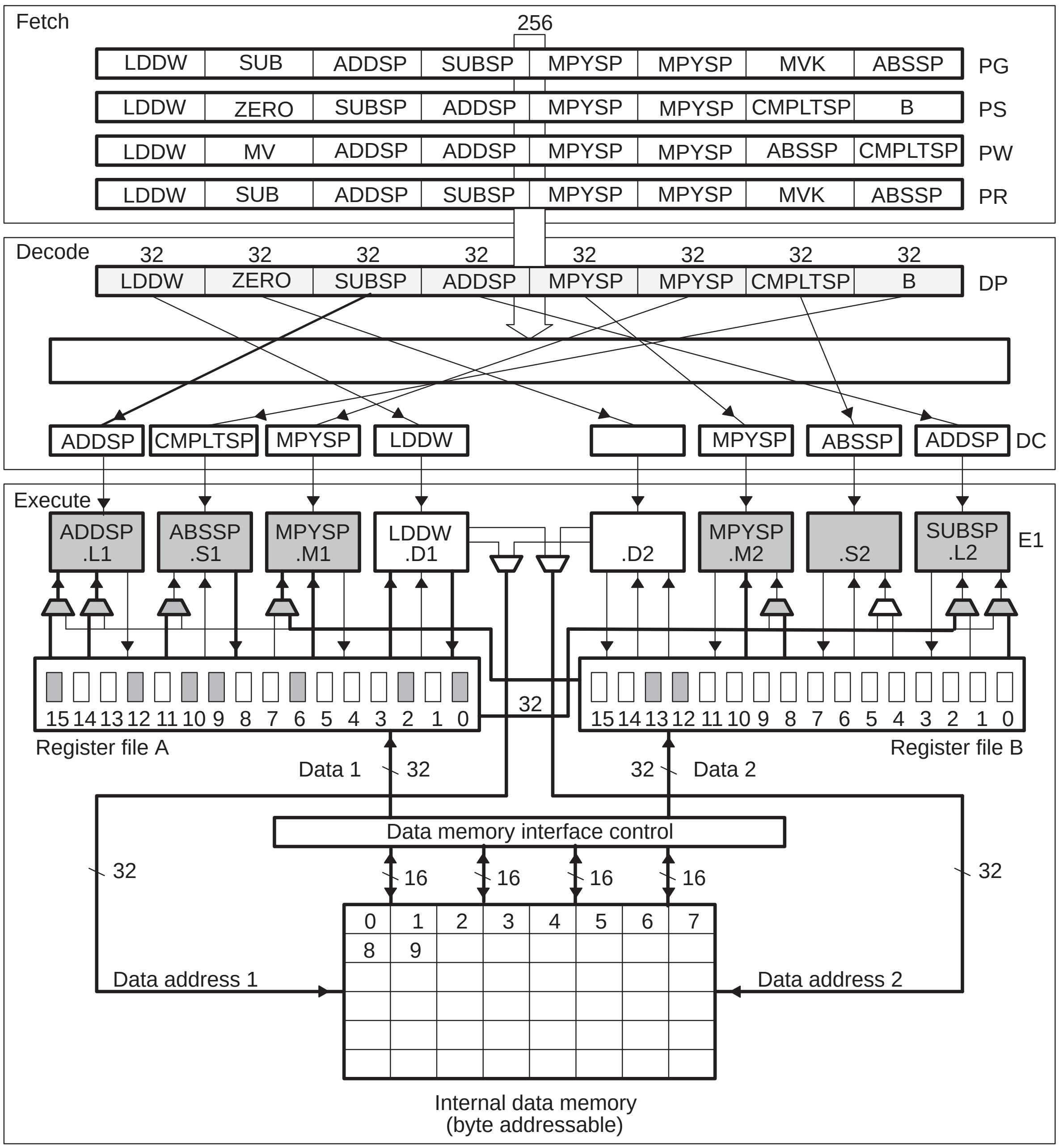
Figure 4−7 shows a functional block diagram of the pipeline stages. The pipeline operation is based on CPU cycles. A CPU cycle is the period during which a particular execute packet is in a particular pipeline phase. CPU cycle boundaries always occur at clock cycle boundaries.

As code flows through the pipeline phases, it is processed by different parts of the C67x DSP. Figure 4−7 shows a full pipeline with a fetch packet in every phase of fetch. One execute packet of eight instructions is being dispatched at the same time 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 4−1.

In the DC phase portion of Figure 4−7, one box is empty because a **NOP** was the eighth instruction in the fetch packet in DC, and no functional unit is needed for a **NOP**. Finally, Figure 4−7 shows six functional units processing code during the same cycle of the pipeline.

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

Figure 4−7. Pipeline Phases Block Diagram



Many C67x DSP 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 4.2.

Example 4−1. Execute Packet in Figure 4−7

```
         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
```

4.2 Pipeline Execution of Instruction Types

The pipeline operation of the C67x DSP instructions can be categorized into fourteen instruction types. Thirteen of these are shown in Table 4−2 (**NOP** is not included in the table), which is a mapping of operations occurring in each execution phase for the different instruction types. The delay slots and functional unit latency associated with each instruction type are listed in the bottom row. See section 3.7.8 for any instruction constraints.

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

Execution phases	Instruction Type				
	Single Cycle	**16 × 16 Multiply**	**Store**	**Load**	**Branch**
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 4.2.3 And 4.2.4 for more information on execution and delay slots for stores and loads.
‡ See section 4.2.5 for more information on branches.

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 4−2. Execution Stage Length Description for Each Instruction Type (Continued)

	Instruction Type			
Execution phases	**2-Cycle DP**	**4-Cycle**	**INTDP**	**DP Compare**
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 4−2. Execution Stage Length Description for Each Instruction Type (Continued)

Execution phases	Instruction Type			
	ADDDP/SUBDP	**MPYI**	**MPYID**	**MPYDP**
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.

Table 4−2. Execution Stage Length Description for Each Instruction Type (Continued)

	Instruction Type	
Execution phases	**MPYSPDP**	**MPYSP2DP**
E1	Read *src1* and lower *src2* and start computation	Read sources and start computation
E2	Read *src1* and upper *src2* and continue computation	Continue computation
E3	Continue computation	Continue computation
E4	Continue computation	Continue computation and write lower results to register
E5	Continue computation	Complete computation and write upper results to register
E6	Continue computation and write lower results to register	
E7	Complete computation and write upper results to register	
E8		
E9		
E10		
Delay slots	6	4
Functional unit latency	3	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.

4.2.1 Single-Cycle Instructions

Single-cycle instructions complete execution during the E1 phase of the pipeline (see Table 4−3). Figure 4−8 shows the fetch, decode, and execute phases of the pipeline that single-cycle instructions use.

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

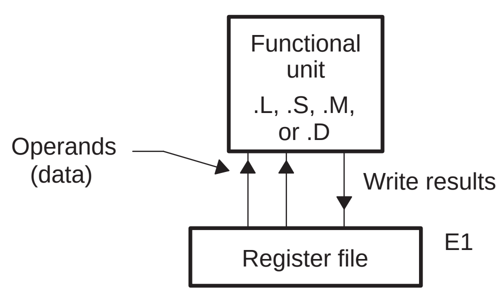
Table 4−3. Single-Cycle Instruction Execution

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

Figure 4−8. Single-Cycle Instruction Phases

PG	PS	PW	PR	DP	DC	E1

Figure 4−9. Single-Cycle Instruction Execution Block Diagram



4.2.2 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 4−4). Figure 4−10 shows the fetch, decode, and execute phases of the pipeline that the multiply instructions use.

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

Table 4−4. 16 × 16-Bit Multiply Instruction Execution

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

Figure 4−10. Multiply Instruction Phases

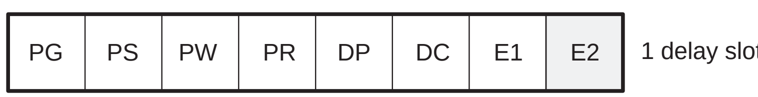
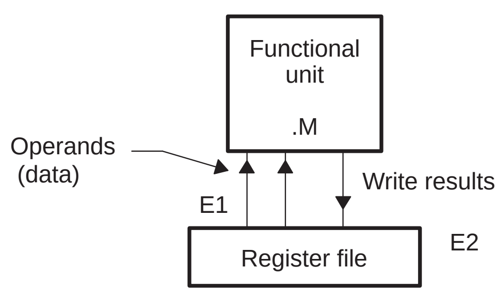


Figure 4−11. Multiply Instruction Execution Block Diagram



4.2.3 Store Instructions

Store instructions require phases E1 through E3 of the pipeline to complete their operations (see Table 4−5). Figure 4−12 shows the fetch, decode, and execute phases of the pipeline that the store instructions use.

Figure 4−13 shows the operations occurring in the pipeline phases for a store instruction. 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. There is additional explanation of why stores have zero delay slots in section 4.2.4.

Table 4−5. Store Instruction Execution

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

Figure 4−12. Store Instruction Phases

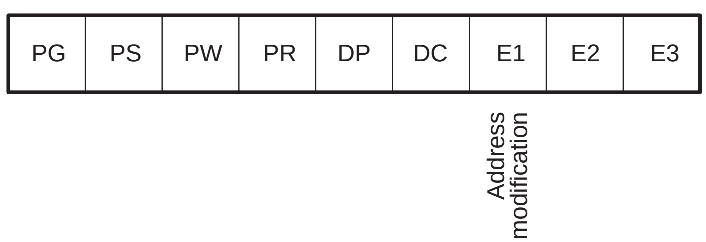
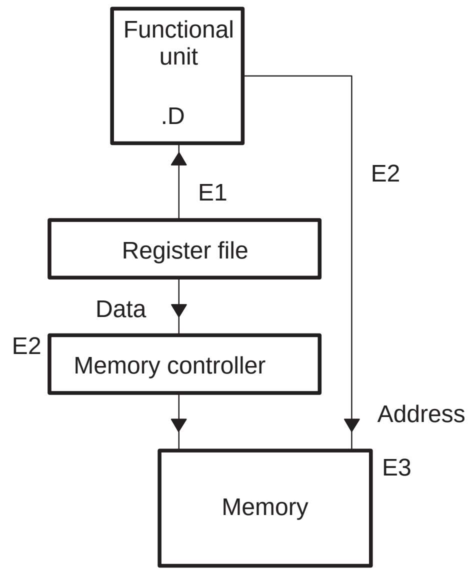


Figure 4−13. Store Instruction Execution Block Diagram



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

4.2.4 Load Instructions

Data loads require five, E1−E5, of the pipeline execute phases to complete their operations (see Table 4−6). Figure 4−14 shows the fetch, decode, and execute phases of the pipeline that the load instructions use.

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

Table 4−6. Load Instruction Execution

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

Figure 4−14. Load Instruction Phases

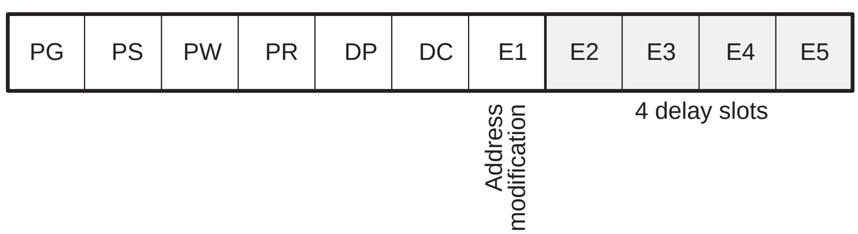
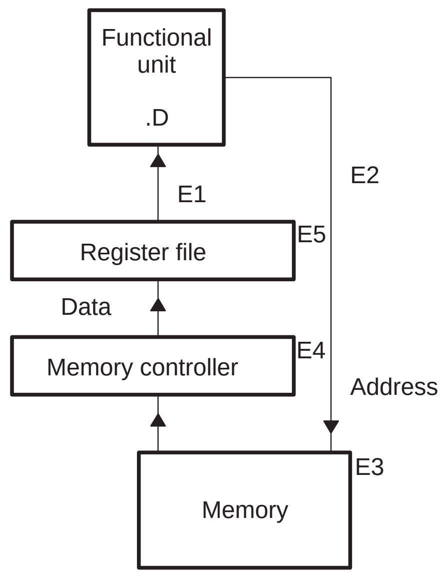


Figure 4−15. Load Instruction Execution Block Diagram



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

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

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

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

4.2.5 Branch Instructions

Although branch takes one execute phase, there are five delay slots between the execution of the branch and execution of the target code (see Table 4−7). Figure 4−16 shows the pipeline phases used by the branch instruction and branch target code. The delay slots are shaded.

Figure 4−17 shows a branch instruction 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.

Table 4−7. Branch Instruction Execution

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

Figure 4−16. Branch Instruction Phases

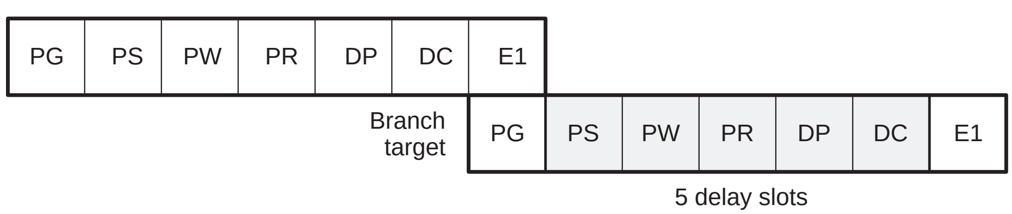
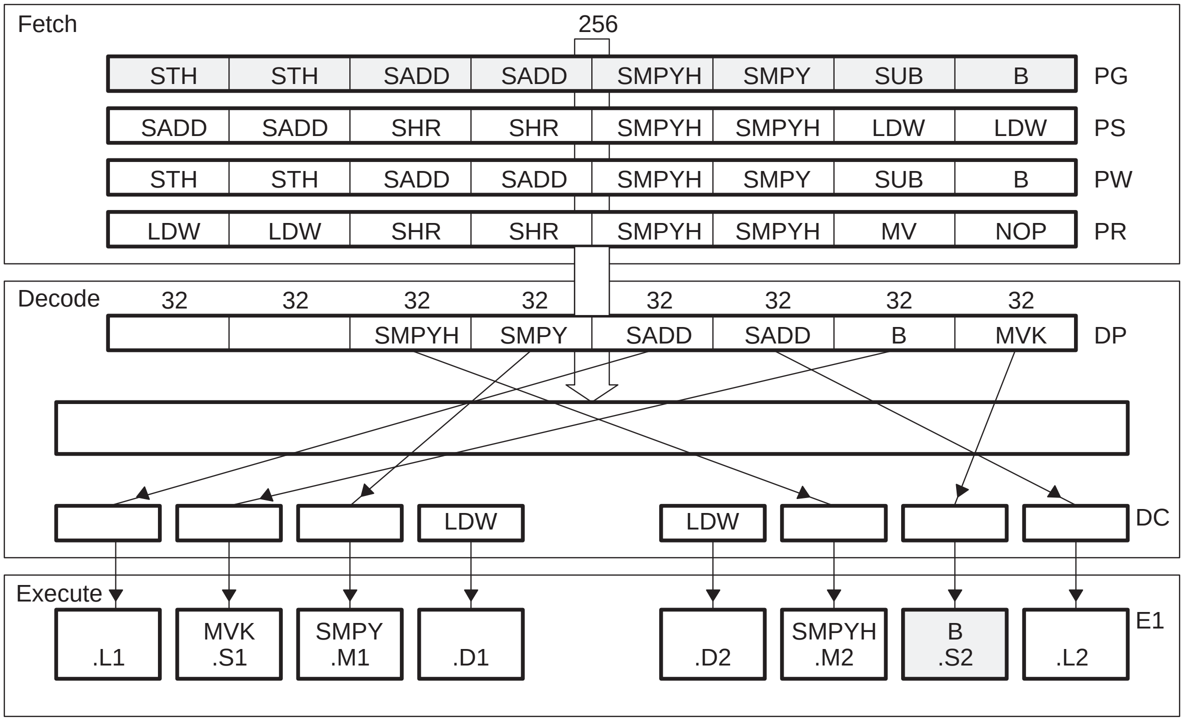


Figure 4−17. Branch Instruction Execution Block Diagram



4.2.6 Two-Cycle DP Instructions

Two-cycle DP instructions use both the E1 and E2 phases of the pipeline to complete their operations (see Table 4−8). 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 two-cycle DP instructions are executed on the .S units. The status is written to the FAUCR on E1. Figure 4−18 shows the fetch, decode, and execute phases of the pipeline that the two-cycle DP instructions use.

Table 4−8. Two-Cycle DP Instruction Execution

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

Figure 4−18. Two-Cycle DP Instruction Phases

PG	PS	PW	PR	DP	DC	E1	E2	
								1 delay slot

4.2.7 Four-Cycle Instructions

Four-cycle instructions use the E1 through E4 phases of the pipeline to complete their operations (see Table 4−9). The following instructions are four-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 four-cycle instructions are executed on the .M or .L units. The status is written to the FMCR or FADCR on E4. Figure 4−19 shows the fetch, decode, and execute phases of the pipeline that the four-cycle instructions use.

Table 4−9. Four-Cycle Instruction Execution

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

Figure 4−19. Four-Cycle Instruction Phases

PG	PS	PW	PR	DP	DC	E1	E2	E3	E4

3 delay slots

4.2.8 INTDP Instruction

The INTDP instruction uses the E1 through E5 phases of the pipeline to complete its operations (see Table 4−10). *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 unit. The status is written to the FADCR on E4. Figure 4−20 shows the fetch, decode, and execute phases of the pipeline that the INTDP instruction uses.

Table 4−10. INTDP Instruction Execution

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

Figure 4−20. INTDP Instruction Phases

PG	PS	PW	PR	DP	DC	E1	E2	E3	E4	E5

4 delay slots

4.2.9 DP Compare Instructions

The DP compare instructions use the E1 and E2 phases of the pipeline to complete their operations (see Table 4−11). 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 4−21 shows the fetch, decode, and execute phases of the pipeline that the DP compare instruction uses.

Table 4−11. DP Compare Instruction Execution

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

Figure 4−21. DP Compare Instruction Phases

PG	PS	PW	PR	DP	DC	E1	E2	1 delay slot

4.2.10 ADDDP/SUBDP Instructions

The ADDDP/SUBDP instructions use the E1 through E7 phases of the pipeline to complete their operations (see Table 4−12). 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 4−22 shows the fetch, decode, and execute phases of the pipeline that the ADDDP/SUBDP instructions use.

Table 4−12. ADDDP/SUBDP Instruction 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 or .S	.L or .S					

Figure 4−22. ADDDP/SUBDP Instruction Phases

PG	PS	PW	PR	DP	DC	E1	E2	E3	E4	E5	E6	E7

6 delay slots

4.2.11 MPYI Instruction

The MPYI instruction uses the E1 through E9 phases of the pipeline to complete its operations (see Table 4−13). 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 4−23 shows the fetch, decode, and execute phases of the pipeline that the MPYI instruction uses.

Table 4−13. MPYI Instruction 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 4−23. MPYI Instruction Phases

PG	PS	PW	PR	DP	DC	E1	E2	E3	E4	E5	E6	E7	E8	E9

8 delay slots

4.2.12 MPYID Instruction

The MPYID instruction uses the E1 through E10 phases of the pipeline to complete its operations (see Table 4−14). 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 4−24 shows the fetch, decode, and execute phases of the pipeline that the MPYID instruction uses.

Table 4−14. MPYID Instruction 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 4−24. MPYID Instruction Phases

PG	PS	PW	PR	DP	DC	E1	E2	E3	E4	E5	E6	E7	E8	E9	E10

9 delay slots

4.2.13 MPYDP Instruction

The MPYDP instruction uses the E1 through E10 phases of the pipeline to complete its operations (see Table 4−15). 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 4−25 shows the fetch, decode, and execute phases of the pipeline that the MPYDP instruction uses.

Table 4−15. MPYDP Instruction 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 4−25. MPYDP Instruction Phases

PG	PS	PW	PR	DP	DC	E1	E2	E3	E4	E5	E6	E7	E8	E9	E10

9 delay slots

4.2.14 MPYSPDP Instruction

The MPYSPDP instruction uses the E1 through E7 phases of the pipeline to complete its operations (see Table 4−16). *src1* is read on E1 and E2. The lower 32 bits of *src2* are read on E1, and the upper 32 bits of *src2* are read on E2. The lower 32 bits of the result are written on E6, and the upper 32 bits of the result are written on E7. The MPYSPDP instruction is executed on the .M unit. The functional unit latency for the MPYSPDP instruction is 3. Figure 4−26 shows the fetch, decode, and execute phases of the pipeline that the MPYSPDP instruction uses.

Table 4−16. MPYSPDP Instruction Execution

Pipeline Stage	E1	E2	E3	E4	E5	E6	E7
Read	*src1* *src2_l*	*src1* *src2_h*					
Written						*dst_l*	*dst_h*
Unit in use	.M	.M					

Figure 4−26. MPYSPDP Instruction Phases

PG	PS	PW	PR	DP	DC	E1	E2	E3	E4	E5	E6	E7

6 delay slots

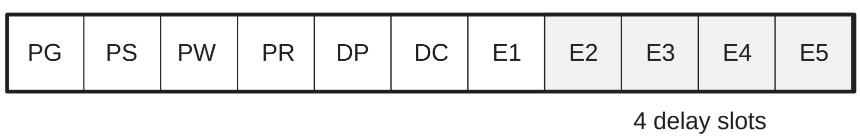
4.2.15 MPYSP2DP Instruction

The MPYSP2DP instruction uses the E1 through E5 phases of the pipeline to complete its operations (see Table 4−17). *src1* and *src2* are 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 MPYSP2DP instruction is executed on the .M unit. The functional unit latency for the MPYSP2DP instruction is 2. Figure 4−27 shows the fetch, decode, and execute phases of the pipeline that the MPYSP2DP instruction uses.

Table 4−17. MPYSP2DP Instruction Execution

Pipeline Stage	**E1**	**E2**	**E3**	**E4**	**E5**
Read	*src1* *src2*				
Written				*dst_l*	*dst_h*
Unit in use	.M				

Figure 4−27. MPYSP2DP Instruction Phases



4.3 Functional Unit Constraints

If you want 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.

4.3.1 .S-Unit Constraints

Table 4−18 shows the instruction constraints for single-cycle instructions executing on the .S unit.

Table 4−18. 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		✓
ADDDP/SUBDP		✓
ADDSP/SUBSP		✓
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-cycle instruction; R = Sources read for the instruction; W = Destinations written for the instruction; ✓ = Next instruction can enter E1 during cycle

Table 4−19 shows the instruction constraints for DP compare instructions executing on the .S unit.

Table 4−19. 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	✓
ADDDP/SUBDP		Xr	✓
ADDSP/SUBSP		Xr	✓
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-cycle 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 4−20 shows the instruction constraints for 2-cycle DP instructions executing on the .S unit.

Table 4−20. 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	✓
ADDDP/SUBDP		✓	
ADDSP/SUBSP		✓	
Branch		✓	✓
Instruction Type	**Same Side, Different Unit, Both Using Cross Path Executable**		
Single cycle		✓	✓
Load		✓	✓
Store		✓	✓
INTDP		✓	✓
ADDDP/SUBDP		✓	✓
16×16 multiply		✓	✓
4-cycle		✓	✓
MPYI		✓	✓
MPYID		✓	✓
MPYDP		✓	✓

Legend: (shaded box) = E1 phase of the single-cycle 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 4−21 shows the instruction constraints for **ADDSP/SUBSP** instructions executing on the .S unit.

Table 4−21. ADDSP/SUBSP .S-Unit Instruction Constraints

	Instruction Execution			
Cycle	1	2	3	4
ADDSP/SUBSP	R			W
Instruction Type	**Subsequent Same-Unit Instruction Executable**			
Single-cycle		✓	✓	Xw
2-cycle DP		✓	Xw	Xw
DP compare		✓	Xw	✓
ADDDP/SUBDP		✓	✓	✓
ADDSP/SUBSP		✓	✓	✓
Branch		✓	✓	✓

Legend: ▒ = E1 phase of the single-cycle 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 4−22 shows the instruction constraints for **ADDDP/SUBDP** instructions executing on the .S unit.

Table 4−22. ADDDP/SUBDP .S-Unit Instruction Constraints

	Instruction Execution						
Cycle	1	2	3	4	5	6	7
ADDDP/SUBDP	R	R				W	W
Instruction Type	**Subsequent Same-Unit Instruction Executable**						
Single-cycle		Xr	✓	✓	✓	Xw	Xw
2-cycle DP		Xr	✓	✓	Xw	Xw	Xw
DP compare		Xr	✓	✓	Xw	Xw	✓
ADDDP/SUBDP		Xr	✓	✓	✓	✓	✓
ADDSP/SUBSP		Xr	Xw	Xw	✓	✓	✓
Branch		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		✓	✓	✓	✓	✓	✓
Store		✓	✓	✓	✓	✓	✓
Branch		Xr	✓	✓	✓	✓	✓
16 $\times$ 16 multiply		Xr	✓	✓	✓	✓	✓
MPYI		Xr	✓	✓	✓	✓	✓
MPYID		Xr	✓	✓	✓	✓	✓
MPYDP		Xr	✓	✓	✓	✓	✓

Legend: = E1 phase of the single-cycle 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

Table 4−23 shows the instruction constraints for branch instructions executing on the .S unit.

Table 4−23. 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		✓	✓	✓	✓	✓	✓	✓
ADDDP/SUBDP		✓	✓	✓	✓	✓	✓	✓
ADDSP/SUBSP		✓	✓	✓	✓	✓	✓	✓
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-cycle 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

4.3.2 .M-Unit Constraints

Table 4−24 shows the instruction constraints for 16 × 16 multiply instructions executing on the .M unit.

Table 4−24. 16 × 16 Multiply .M-Unit Instruction Constraints

	Instruction Execution		
Cycle	1	2	3
16 × 16 multiply	R	W	
Instruction Type	**Subsequent Same-Unit Instruction Executable**		
16 × 16 multiply		✓	✓
4-cycle		✓	✓
MPYI		✓	✓
MPYID		✓	✓
MPYDP		✓	✓
Instruction Type	**Same Side, Different Unit, Both Using Cross Path Executable**		
Single-cycle		✓	✓
Load		✓	✓
Store		✓	✓
DP compare		✓	✓
2-cycle DP		✓	✓
Branch		✓	✓
4-cycle		✓	✓
INTDP		✓	✓
ADDDP/SUBDP		✓	✓

Legend: = E1 phase of the single-cycle instruction; R = Sources read for the instruction; W = Destinations written for the instruction; ✓ = Next instruction can enter E1 during cycle

Table 4−25 shows the instruction constraints for 4-cycle instructions executing on the .M unit.

Table 4−25. 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-cycle 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 4−26 shows the instruction constraints for **MPYI** instructions executing on the .M unit.

Table 4−26. 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	✓	✓	✓
MPYSPDP		Xr	Xr	Xr	Xu	Xu	Xu	✓	✓	✓
MPYSP2DP		Xr	Xr	Xr	Xw	Xw	Xu	✓	✓	✓
Instruction Type	**Same Side, Different Unit, Both Using Cross Path Executable**									
Single-cycle		Xr	Xr	Xr	✓	✓	✓	✓	✓	✓
Load		✓	✓	✓	✓	✓	✓	✓	✓	✓
Store		✓	✓	✓	✓	✓	✓	✓	✓	✓
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-cycle 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 4−27 shows the instruction constraints for **MPYID** instructions executing on the .M unit.

Table 4−27. 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	✓	✓	✓	✓
MPYSPDP		Xr	Xr	Xr	Xw	Xu	Xu	✓	✓	✓	✓
MPYSP2DP		Xr	Xr	Xr	Xw	Xw	Xw	✓	✓	✓	✓
Instruction Type	**Same Side, Different Unit, Both Using Cross Path Executable**										
Single-cycle		Xr	Xr	Xr	✓	✓	✓	✓	✓	✓	✓
Load		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Store		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
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-cycle 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 4−28 shows the instruction constraints for **MPYDP** instructions executing on the .M unit.

Table 4−28. 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	✓	✓	✓	✓	✓	✓	✓
MPYSPDP		Xr	Xr	Xr	Xw	Xu	Xu	✓	✓	✓	✓
MPYSP2DP		Xr	Xr	Xr	Xw	Xw	Xw	✓	✓	✓	✓
Instruction Type	**Same Side, Different Unit, Both Using Cross Path Executable**										
Single-cycle		Xr	Xr	Xr	✓	✓	✓	✓	✓	✓	✓
Load		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Store		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
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-cycle 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 4−29 shows the instruction constraints for **MPYSP** instructions executing on the .M unit.

Table 4−29. MPYSP .M-Unit Instruction Constraints

	Instruction Execution			
Cycle	1	2	3	4
MPYSP	R			W
Instruction Type	**Subsequent Same-Unit Instruction Executable**			
MPYSPDP		✓	✓	✓
MPYSP2DP		✓	✓	✓
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-cycle instruction; R = Sources read for the instruction; W = Destinations written for the instruction; ✓ = Next instruction can enter E1 during cycle

Table 4−30 shows the instruction constraints for **MPYSPDP** instructions executing on the .M unit.

Table 4−30. MPYSPDP .M-Unit Instruction Constraints

	Instruction Execution						
Cycle	1	2	3	4	5	6	7
MPYSPDP	R	R				W	W
Instruction Type	**Subsequent Same-Unit Instruction Executable**						
16 × 16 multiply		Xr	✓	✓	Xw	Xw	✓
MPYDP		Xr	Xu	Xu	✓	✓	✓
MPYI		Xr	Xu	Xu	✓	✓	✓
MPYID		Xr	Xu	Xu	✓	✓	✓
MPYSP		Xr	Xw	Xw	✓	✓	✓
MPYSPDP		Xr	Xu	✓	✓	✓	✓
MPYSP2DP		Xr	Xw	Xw	✓	✓	✓
Instruction Type	**Same Side, Different Unit, Both Using Cross Path Executable**						
Single-cycle		Xr	✓	✓	✓	✓	✓
Load		Xr	✓	✓	✓	✓	✓
Store		Xr	✓	✓	✓	✓	✓
DP compare		Xr	✓	✓	✓	✓	✓
2-cycle DP		Xr	✓	✓	✓	✓	✓
Branch		Xr	✓	✓	✓	✓	✓
4-cycle		Xr	✓	✓	✓	✓	✓
INTDP		Xr	✓	✓	✓	✓	✓
ADDDP/SUBDP		Xr	✓	✓	✓	✓	✓

Legend: ▒ = E1 phase of the single-cycle 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 4−31 shows the instruction constraints for **MPYSP2DP** instructions executing on the .M unit.

Table 4−31. MPYSP2DP .M-Unit Instruction Constraints

	Instruction Execution				
Cycle	1	2	3	4	5
MPYSP2DP	R	R		W	W
Instruction Type	**Subsequent Same-Unit Instruction Executable**				
16 × 16 multiply		✓	Xw	Xw	✓
MPYDP		Xu	✓	✓	✓
MPYI		Xu	✓	✓	✓
MPYID		Xu	✓	✓	✓
MPYSP		Xw	✓	✓	✓
MPYSPDP		Xu	✓	✓	✓
MPYSP2DP		Xw	✓	✓	✓
Instruction Type	**Same Side, Different Unit, Both Using Cross Path Executable**				
Single-cycle		Xr	✓	✓	✓
Load		Xr	✓	✓	✓
Store		Xr	✓	✓	✓
DP compare		Xr	✓	✓	✓
2-cycle DP		Xr	✓	✓	✓
Branch		Xr	✓	✓	✓
4-cycle		Xr	✓	✓	✓
INTDP		Xr	✓	✓	✓
ADDDP/SUBDP		Xr	✓	✓	✓

Legend: ▒ = E1 phase of the single-cycle 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

4.3.3 .L-Unit Constraints

Table 4−32 shows the instruction constraints for single-cycle instructions executing on the .L unit.

Table 4−32. 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-cycle instruction; R = Sources read for the instruction; W = Destinations written for the instruction; ✓ = Next instruction can enter E1 during cycle

Table 4−33 shows the instruction constraints for 4-cycle instructions executing on the .L unit.

Table 4−33. 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-cycle 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 4−34 shows the instruction constraints for **INTDP** instructions executing on the .L unit.

Table 4−34. 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-cycle 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 4−35 shows the instruction constraints for **ADDDP/SUBDP** instructions executing on the .L unit.

Table 4−35. 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		✓	✓	✓	✓	✓	✓	✓
Store		✓	✓	✓	✓	✓	✓	✓
Branch		Xr	✓	✓	✓	✓	✓	✓
16 × 16 multiply		Xr	✓	✓	✓	✓	✓	✓
MPYI		Xr	✓	✓	✓	✓	✓	✓
MPYID		Xr	✓	✓	✓	✓	✓	✓
MPYDP		Xr	✓	✓	✓	✓	✓	✓

Legend: ▒ = E1 phase of the single-cycle 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

4.3.4 .D-Unit Instruction Constraints

Table 4−36 shows the instruction constraints for load instructions executing on the .D unit.

Table 4−36. 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-cycle instruction; R = Sources read for the instruction; W = Destinations written for the instruction; ✓ = Next instruction can enter E1 during cycle

Table 4−37 shows the instruction constraints for store instructions executing on the .D unit.

Table 4−37. 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-cycle instruction; R = Sources read for the instruction; W = Destinations written for the instruction; ✓ = Next instruction can enter E1 during cycle

Table 4−38 shows the instruction constraints for single-cycle instructions executing on the .D unit.

Table 4−38. 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-cycle instruction; R = Sources read for the instruction; W = Destinations written for the instruction; ✓ = Next instruction can enter E1 during cycle

Table 4−39 shows the instruction constraints for **LDDW** instructions executing on the .D unit.

Table 4−39. 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-cycle 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

4.4 Performance Considerations

The C67x DSP pipeline is most effective when it is kept as full as the algorithms in the program allow it to be. It is useful to consider some situations that can affect pipeline performance.

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

In general, the number of execute packets in a single FP defines the flow of instructions through the pipeline. Another defining factor is the instruction types in the EP. Each type of instruction has a fixed number of execute cycles that determines when this instruction's operations are complete. Section 4.4.2 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.

4.4.1 Pipeline Operation With Multiple Execute Packets in a Fetch Packet

Referring to Figure 4−6, page 4-6, pipeline operation is shown with eight instructions in every fetch packet. Figure 4−28, however, shows the pipeline operation with a fetch packet that contains multiple execute packets. Code for Figure 4−28 might have this layout:

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

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

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

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

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

Figure 4−28. Pipeline Operation: Fetch Packets With Different Numbers of Execute Packets

		Clock cycle																
Fetch packet (FP)	**Execute packet (EP)**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**
n	**k**	PG	PS	PW	PR	DP	DC	E1	E2	E3	E4	E5	E6	E7	E8	E9	E10	
n	**k+1**						DP	DC	E1	E2	E3	E4	E5	E6	E7	E8	E9	E10
n	**k+2**							DP	DC	E1	E2	E3	E4	E5	E6	E7	E8	E9
n+1	**k+3**		PG	PS	PW	PR			DP	DC	E1	E2	E3	E4	E5	E6	E7	E8
n+2	**k+4**			PG	PS	PW	Pipeline		PR	DP	DC	E1	E2	E3	E4	E5	E6	E7
n+3	**k+5**				PG	PS	stall		PW	PR	DP	DC	E1	E2	E3	E4	E5	E6
n+4	**k+6**					PG			PS	PW	PR	DP	DC	E1	E2	E3	E4	E5
n+5	**k+7**								PG	PS	PW	PR	DP	DC	E1	E2	E3	E4
n+6	**k+8**									PG	PS	PW	PR	DP	DC	E1	E2	E3

In Figure 4−28, fetch packet n, which contains three execute packets, is shown followed by six fetch packets (n + 1 through n + 6), each with one execute packet (containing eight parallel instructions). The first fetch packet (n) goes through the program fetch phases during cycles 1−4. During these cycles, a program fetch phase is started for each of the fetch packets that follow.

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

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

4.4.2 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** results is available for use by instructions in the next execute packet.

Figure 4−29 shows how a multicycle **NOP** can drive the execution of other instructions in the same execute packet. Figure 4−29(a) shows a **NOP** in an execute packet (in parallel) with other code. The results of the **LD**, **ADD**, and **MPY** is available during the proper cycle for each instruction. Hence **NOP** has no effect on the execute packet.

Figure 4−29(b) shows the replacement of the single-cycle **NOP** with a multicycle **NOP** (**NOP 5**) in the same execute packet. The **NOP 5** causes no operation to perform other than the operations from the instructions inside its execute packet. The results of the **LD**, **ADD**, and **MPY** cannot be used by any other instructions until the **NOP 5** period has completed.

Figure 4−29. Multicycle NOP in an Execute Packet

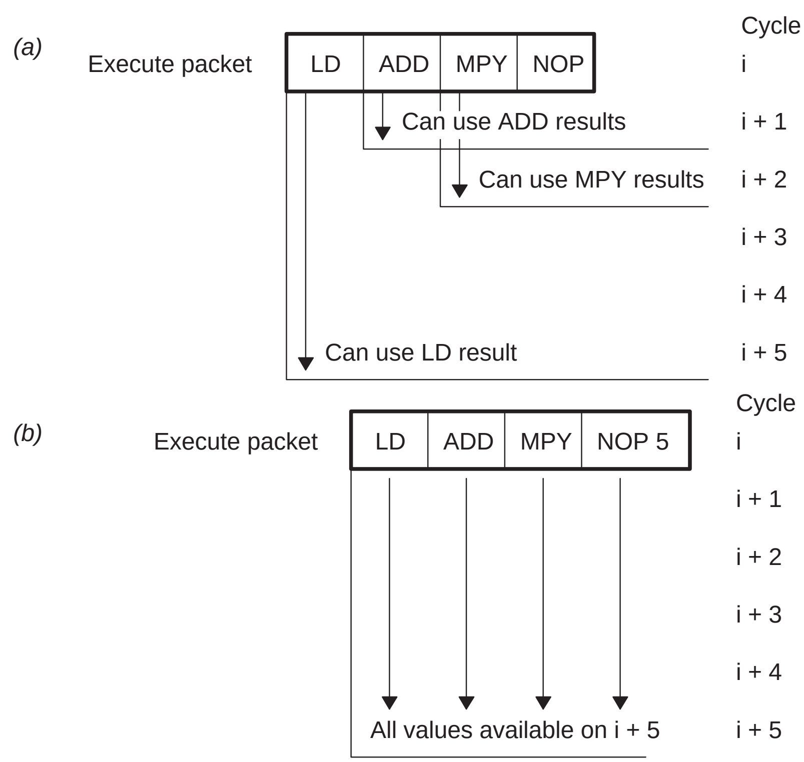
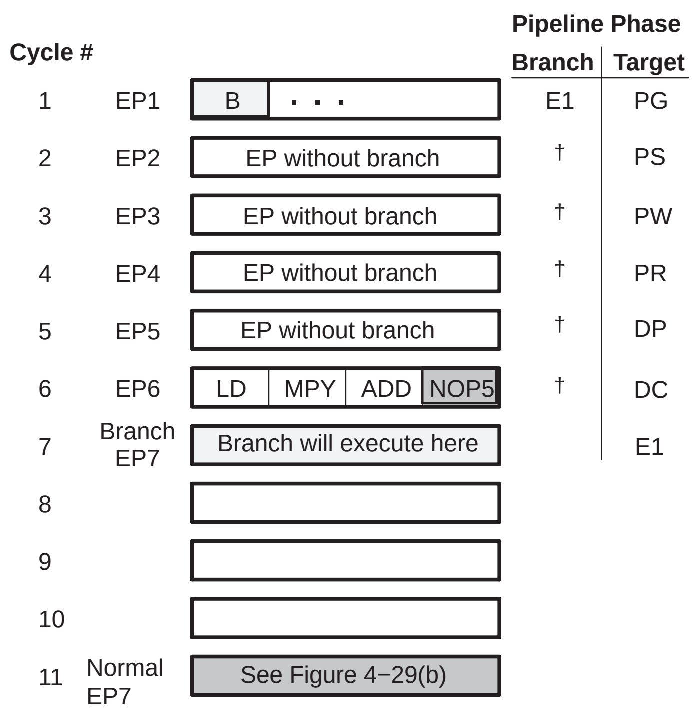


Figure 4−30 shows how a multicycle **NOP** can be affected by a branch. If the delay slots of a branch finish while a multicycle **NOP** is still dispatching **NOP**s into the pipeline, the branch overrides the multicycle **NOP** and the branch target begins execution five delay slots after the branch was issued.

Figure 4−30. Branching and Multicycle NOPs



† Delay slots of the branch

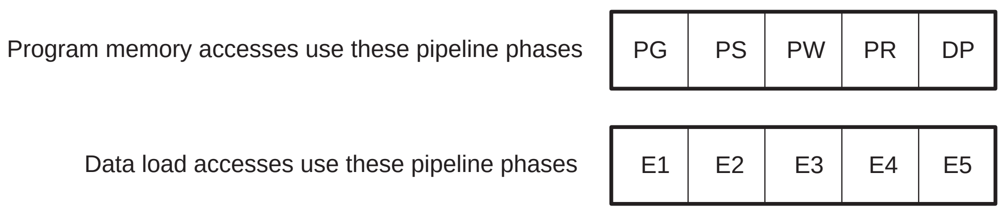
In one case, execute packet 1 (EP1) does not have a branch. The **NOP 5** in EP6 forces the CPU to wait until cycle 11 to execute EP7.

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

4.4.3 Memory Considerations

The C67x DSP has a memory configuration 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 C67x DSP to access memory at a high speed. These phases are shown in Figure 4−31.

Figure 4−31. Pipeline Phases Used During Memory Accesses



To understand the memory accesses, compare data loads and instruction fetches/dispatches. The comparison is valid because data loads and program fetches operate on internal memories of the same speed on the C67x DSP and perform the same types of operations (listed in Table 4−40) to accommodate those memories. Table 4−40 shows the operation of program fetches pipeline versus the operation of a data load.

Table 4−40. Program Memory Accesses Versus Data Load Accesses

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

Depending on the type of memory and the time required to complete an access, the pipeline may stall to ensure proper coordination of data and instructions. This is discussed in section 4.4.3.1.

In the instance where multiple accesses are made to a single ported memory, the pipeline will stall to allow the extra access to occur. This is called a memory bank hit and is discussed in section 4.4.3.2.

4.4.3.1 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 4−32 illustrates this point.

Figure 4−32. Program and Data Memory Stalls

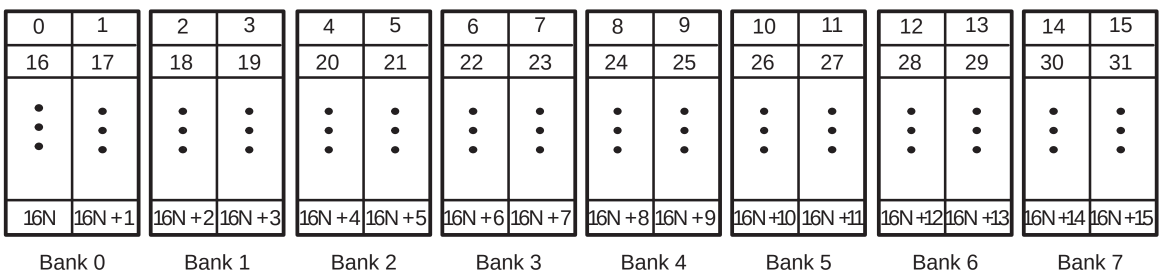
Clock cycle

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

4.4.3.2 Memory Bank Hits

Most C67x devices use an interleaved memory bank scheme, as shown in Figure 4−33. 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 4−33. 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 4−2. 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 4−41 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 *TMS320C6000 Programmer's Guide* (SPRU198).

Example 4−2. 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
```

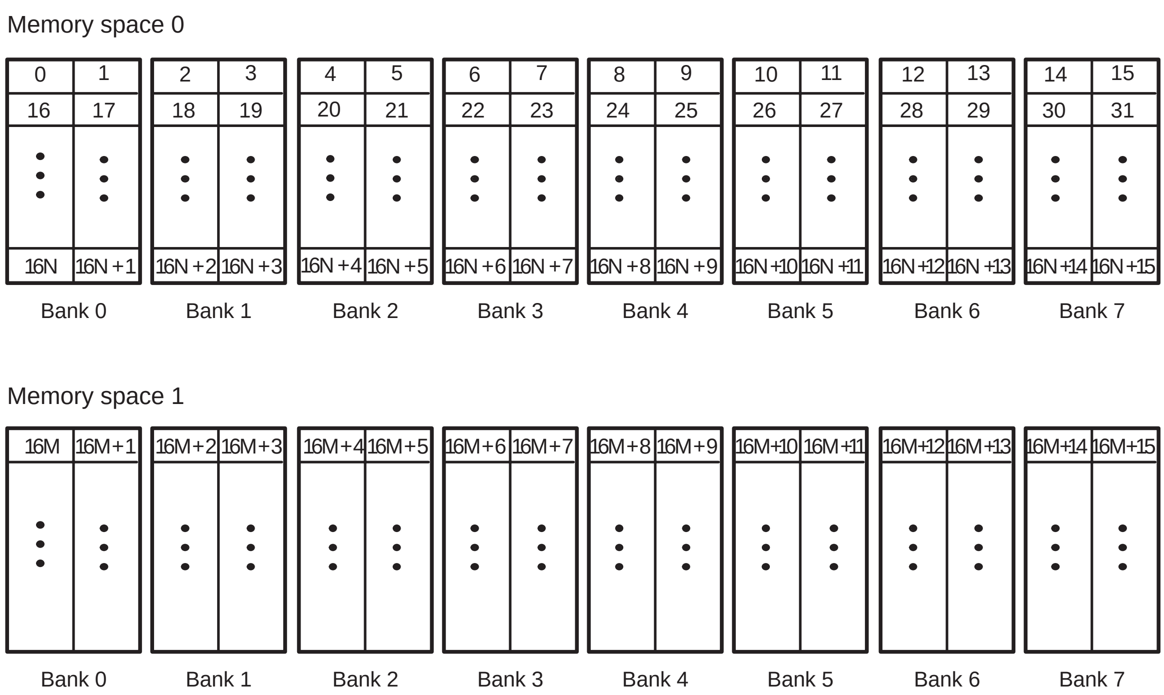
Table 4−41. Loads in Pipeline from Example 4−2

	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 4−34), 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 DSP family varies from device to device. See the device-specific data manual to determine the memory spaces in your device.

Figure 4−34. 8-Bank Interleaved Memory With Two Memory Spaces



Chapter 5

Interrupts

This chapter describes CPU interrupts, including reset and the nonmaskable interrupt (NMI). It details the related CPU control registers and their functions in controlling interrupts. It also describes interrupt processing, the method the CPU uses to detect automatically the presence of interrupts and divert program execution flow to your interrupt service code. Finally, the chapter describes the programming implications of interrupts.

Topic **Page**

5.1 Overview

Typically, DSPs work in an environment that contains multiple external asynchronous events. These events require tasks to be performed by the DSP when they occur. An interrupt is an event that stops the current process in the CPU so that the CPU can attend to the task needing completion because of the event. These interrupt sources can be on chip or off chip, such as timers, analog-to-digital converters, or other peripherals.

Servicing an interrupt involves saving the context of the current process, completing the interrupt task, restoring the registers and the process context, and resuming the original process. There are eight registers that control servicing interrupts.

An appropriate transition on an interrupt pin sets the pending status of the interrupt within the interrupt flag register (IFR). If the interrupt is properly enabled, the CPU begins processing the interrupt and redirecting program flow to the interrupt service routine.

5.1.1 Types of Interrupts and Signals Used

There are three types of interrupts on the CPUs of the TMS320C6000™ DSPs.

- Reset
- Maskable
- Nonmaskable

These three types are differentiated by their priorities, as shown in Table 5−1. The reset interrupt has the highest priority and corresponds to the $\overline{\text{RESET}}$ signal. The nonmaskable interrupt has the second highest priority and corresponds to the NMI signal. The lowest priority interrupts are interrupts 4−15 corresponding 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 device-specific data manual to see your interrupt specifications.

Table 5−1. Interrupt Priorities

Priority	Interrupt Name	Interrupt Type
Highest	Reset	Reset
	NMI	Nonmaskable
	INT4	Maskable
	INT5	Maskable
	INT6	Maskable
	INT7	Maskable
	INT8	Maskable
	INT9	Maskable
	INT10	Maskable
	INT11	Maskable
	INT12	Maskable
	INT13	Maskable
	INT14	Maskable
Lowest	INT15	Maskable

5.1.1.1 Reset ($\overline{\textit{RESET}}$)

Reset is the highest priority interrupt and is used to halt the CPU and return it to a known state. The reset interrupt is unique in a number of ways:

- $\overline{\text{RESET}}$ is an active-low signal. All other interrupts are active-high signals.
- $\overline{\text{RESET}}$ must be held low for 10 clock cycles before it goes high again to reinitialize the CPU properly.
- The instruction execution in progress is aborted and all registers are returned to their default states.
- The reset interrupt service fetch packet must be located at address 0.
- $\overline{\text{RESET}}$ is not affected by branches.

5.1.1.2 Nonmaskable Interrupt (NMI)

NMI is the second-highest priority interrupt and is generally used to alert the CPU of a serious hardware problem such as imminent power failure.

For NMI processing to occur, the nonmaskable interrupt enable (NMIE) bit in the interrupt enable register must be set to 1. If NMIE is set to 1, the only condition that can prevent NMI processing is if the NMI occurs during the delay slots of a branch (whether the branch is taken or not).

NMIE is cleared to 0 at reset to prevent interruption of the reset. It is cleared at the occurrence of an NMI to prevent another NMI from being processed. You cannot manually clear NMIE, but you can set NMIE to allow nested NMIs. While NMI is cleared, all maskable interrupts (INT4−INT15) are disabled.

5.1.1.3 Maskable Interrupts (INT4−INT15)

The 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 interrupt flags register (IFR) to 1 and there are no higher priority interrupt flag (IF) bits set in the IFR.

5.1.1.4 Interrupt Acknowledgment (IACK) and Interrupt Number (INUMn)

The IACK and INUM*n* 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 INUM*n* signal (INUM3−INUM0) indicates the number of the interrupt (bit position in the IFR) that is being processed. For example:

INUM3 = 0 (MSB)
INUM2 = 1
INUM1 = 1
INUM0 = 1 (LSB)

Together, these signals provide the 4-bit value 0111, indicating INT7 is being processed.

5.1.2 Interrupt Service Table (IST)

When the CPU begins processing an interrupt, it references the interrupt service table (IST). The IST is a table of fetch packets that contain code for servicing the interrupts. The IST consists of 16 consecutive fetch packets. Each interrupt service fetch packet (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 5−1. Because each fetch packet contains eight 32-bit instruction words (or 32 bytes), each address in the table is incremented by 32 bytes (20h) from the one adjacent to it.

Figure 5−1. Interrupt Service Table

000h	$\overline{\text{RESET}}$ ISFP
020h	NMI ISFP
040h	Reserved
060h	Reserved
080h	INT4 ISFP
0A0h	INT5 ISFP
0C0h	INT6 ISFP
0E0h	INT7 ISFP
100h	INT8 ISFP
120h	INT9 ISFP
140h	INT10 ISFP
160h	INT11 ISFP
180h	INT12 ISFP
1A0h	INT13 ISFP
1C0h	INT14 ISFP
1E0h	INT15 ISFP

Program memory

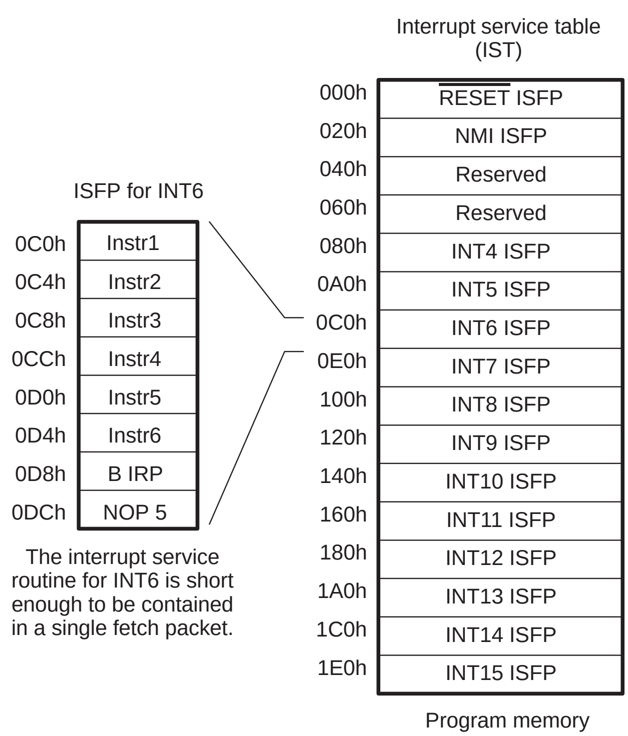
5.1.2.1 Interrupt Service Fetch Packet (ISFP)

An ISFP is a fetch packet used to service an interrupt. Figure 5−2 shows an ISFP that contains an interrupt service routine small enough to fit in a single fetch packet (FP). To branch back to the main program, the FP contains a branch to the interrupt return pointer instruction (**B IRP**). This is followed by a **NOP 5** instruction to allow the branch target to reach the execution stage of the pipeline.

Note:

If the **NOP 5** was not in the routine, the CPU would execute the next five execute packets that are associated with the next ISFP.

Figure 5−2. Interrupt Service Fetch Packet

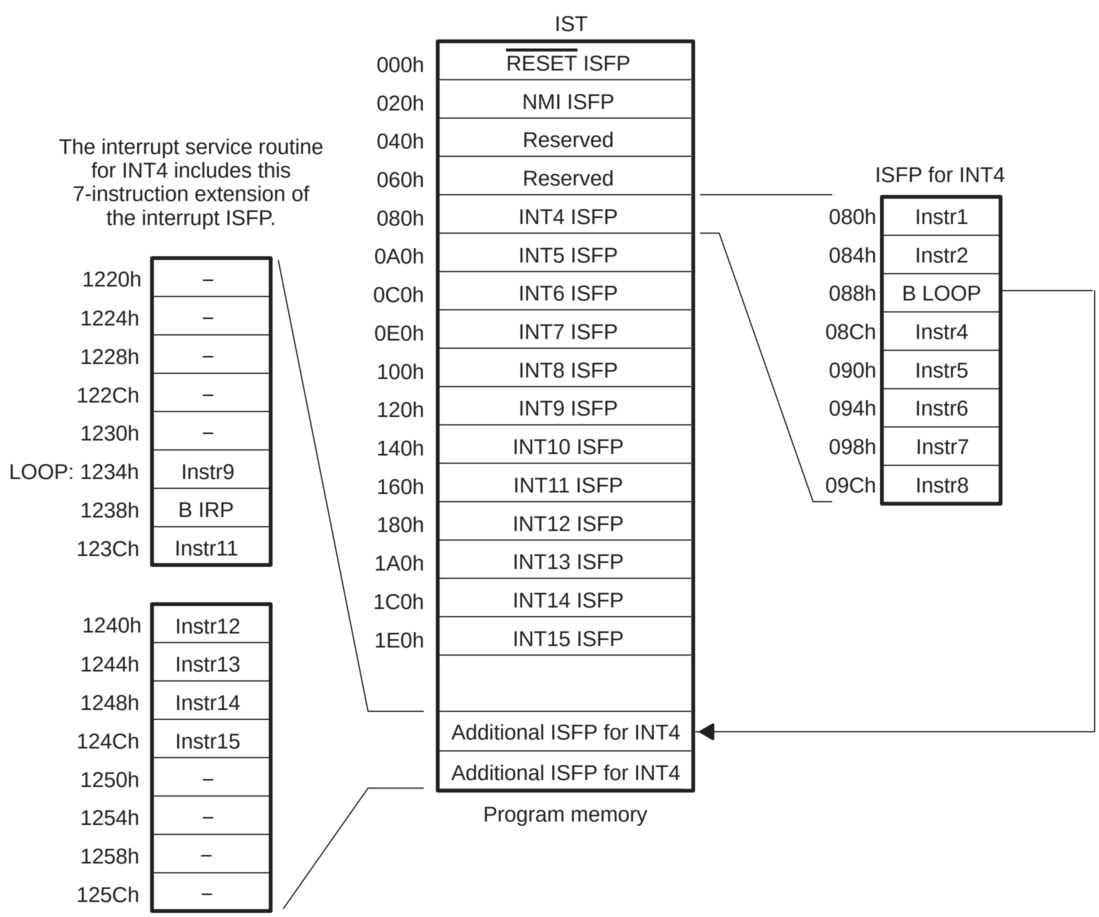


If the interrupt service routine for an interrupt is too large to fit in a single fetch packet, a branch to the location of additional interrupt service routine code is required. Figure 5−3 shows that the interrupt service routine for INT4 was too large for a single fetch packet, and a branch to memory location 1234h is required to complete the interrupt service routine.

Note:

The instruction **B LOOP** branches into the middle of a fetch packet and processes code starting at address 1234h. The CPU ignores code from address 1220h−1230h, even if it is in parallel to code at address 1234h.

Figure 5−3. Interrupt Service Table With Branch to Additional Interrupt Service Code Located Outside the IST



5.1.2.2 Interrupt Service Table Pointer (ISTP)

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 interrupt service table pointer register (ISTP). The ISTP is shown in Figure 2−11 (page 2-21) and described in Table 2−12. Example 5−1 shows the relationship of the ISTB to the table location.

Example 5−1. Relocation of Interrupt Service Table

(a) *Relocating the IST to 800h*

1) Copy the IST, located between 0h and 200h, to the memory location between 800h and A00h.

2) Write 800h to ISTP:

```
MVK 800h, A2
MVC A2, ISTP
```

ISTP = 800h = 1000 0000 0000b

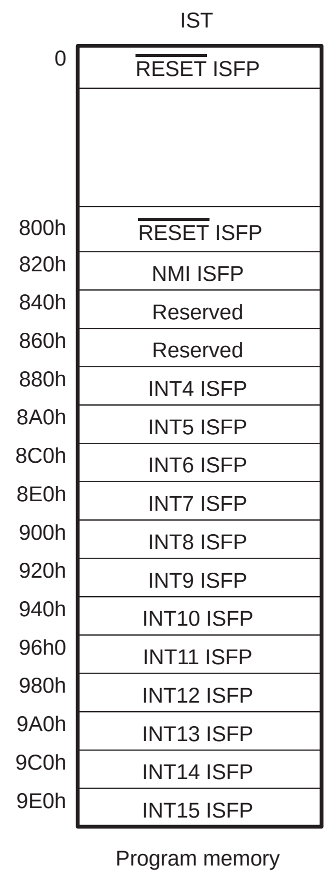
(b) *How the ISTP directs the CPU to the appropriate ISFP in the relocated IST*

Assume: IFR = BBC0h = 1011 1011 1100 0000b
IER = 1230h = 0001 0010 0011 0001b

2 enabled interrupts pending: INT9 and INT12

The 1s in the IFR indicate pending interrupts; the 1s in the IER indicate the interrupts that are enabled. INT9 has a higher priority than INT12, so HPEINT is encoded with the value for INT9, 01001b.

HPEINT corresponds to bits 9−5 of the ISTP:
ISTP = 1001 0010 0000b = 920h = address of INT9



5.1.3 Summary of Interrupt Control Registers

Table 5−2 lists the interrupt control registers on the C67x CPU.

Table 5−2. Interrupt Control Registers

Acronym	Register Name	Description	Page
CSR	Control status register	Allows you to globally set or disable interrupts	2-13
ICR	Interrupt clear register	Allows you to clear flags in the IFR manually	2-16
IER	Interrupt enable register	Allows you to enable interrupts	2-17
IFR	Interrupt flag register	Shows the status of interrupts	2-18
IRP	Interrupt return pointer register	Contains the return address used on return from a maskable interrupt. This return is accomplished via the B IRP instruction.	2-19
ISR	Interrupt set register	Allows you to set flags in the IFR manually	2-20
ISTP	Interrupt service table pointer register	Pointer to the beginning of the interrupt service table	2-21
NRP	Nonmaskable interrupt return pointer register	Contains the return address used on return from a nonmaskable interrupt. This return is accomplished via the B NRP instruction.	2-22

5.2 Globally Enabling and Disabling Interrupts

The control status register (CSR) contains two fields that control interrupts: GIE and PGIE, as shown in Figure 2−4 (page 2-13) and described in Table 2−7 (page 2-14). The global interrupt enable (GIE) allows you to enable or disable all maskable interrupts:

- 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 CSR is the PGIE bit and holds the previous value of GIE when a maskable interrupt is processed. During maskable interrupt processing, the value of the GIE bit is copied to the PGIE bit, and the GIE bit is cleared. The previous value of the PGIE bit is lost. The GIE bit is cleared during a maskable interrupt to prevent another maskable interrupt from occurring before the device state has been saved. Upon returning from an interrupt, by way of the **B IRP** instruction, the content of the PGIE bit is copied back to the GIE bit. The PGIE bit remains unchanged.

The purpose of the PGIE bit is to record the value of the GIE bit at the time the interrupt processing begins. This is necessary because interrupts are detected in parallel with instruction execution. Typically, the GIE bit is 1 when an interrupt is taken. However, if an interrupt is detected in parallel with an **MVC** instruction that clears the GIE bit, the GIE bit may be cleared by the **MVC** instruction after the interrupt processing begins. Because the PGIE bit records the state of the GIE bit after all instructions have completed execution, the PGIE bit captures the fact that the GIE bit was cleared as the interrupt was taken.

For example, suppose the GIE bit is set to 1 as the sequence of code shown in Example 5−2 is entered. An interrupt occurs, and the CPU detects it just as the CPU is executing the **MVC** instruction that writes a 0 to the GIE bit. Interrupt processing begins. Meanwhile, the 0 is written to the GIE bit as the **MVC** instruction completes. During the interrupt dispatch, this updated value of the GIE bit is copied to the PGIE bit, leaving the PGIE bit cleared to 0. Later, upon returning from the interrupt (using the **B IRP** instruction), the PGIE bit is copied to the GIE bit. As a result, the code following the **MVC** instruction recognizes the GIE bit is cleared to 0, as directed by the **MVC** instruction, despite having taken the interrupt.

Example 5−2 and Example 5−3 show code examples for disabling and enabling maskable interrupts globally, respectively.

Example 5−2. 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 5−3. 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
```

5.3 Individual Interrupt Control

Servicing interrupts effectively requires individual control of all three types of interrupts: reset, nonmaskable, and maskable. Enabling and disabling individual interrupts is done with the interrupt enable register (IER). The status of pending interrupts is stored in the interrupt flag register (IFR). Manual interrupt processing can be accomplished through the use of the interrupt set register (ISR) and interrupt clear register (ICR). The interrupt return pointers restore context after servicing nonmaskable and maskable interrupts.

5.3.1 Enabling and Disabling Interrupts

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 2−7 (page 2-17) and described in Table 2−9.

When NMIE = 0, all nonreset interrupts are disabled, preventing interruption of an NMI. The NMIE bit is cleared at reset to prevent any interruption of process or initialization until you enable NMI. After reset, you must set the NMIE bit to enable the NMI and to allow INT15−INT4 to be enabled by the GIE bit in CSR and the corresponding IER bit. You cannot manually clear the NMIE bit; the NMIE bit is unaffected by a write of 0. The NMIE bit is also cleared by the occurrence of an NMI. If cleared, the NMIE bit is set only by completing a **B NRP** instruction or by a write of 1 to the NMIE bit. Example 5−4 and Example 5−5 show code for enabling and disabling individual interrupts, respectively.

Example 5−4. Code Sequence to Enable an Individual Interrupt (INT9)

```
        MVK    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 5−5. 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
```

5.3.2 Status of Interrupts

The interrupt flag register (IFR) contains the status of INT4−INT15 and NMI. Each interrupt's corresponding bit in 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 IFR. The IFR is shown in Figure 2−8 (page 2-18) and described in Table 2−10.

5.3.3 Setting and Clearing Interrupts

The interrupt set register (ISR) and the interrupt clear register (ICR) allow you to set or clear maskable interrupts manually in IFR. Writing a 1 to IS4−IS15 in ISR causes the corresponding interrupt flag to be set in IFR. Similarly, writing a 1 to a bit in ICR causes the corresponding interrupt flag to be cleared. Writing a 0 to any bit of either ISR or ICR has no effect. Incoming interrupts have priority and override any write to ICR. You cannot set or clear any bit in ISR or ICR to affect NMI or reset. The ISR is shown in Figure 2−10 (page 2-20) and described in Table 2−11. The ICR is shown in Figure 2−6 (page 2-16) and described in Table 2−8.

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 IFR until two cycles after the write to ISR or ICR.

Any write to ICR is ignored by a simultaneous write to the same bit in ISR.

Example 5−6 and Example 5−7 show code examples to set and clear individual interrupts, respectively.

Example 5−6. Code to Set an Individual Interrupt (INT6) and Read the Flag Register

```
        MVK    40h,B3
        MVC    B3,ISR
        NOP
        MVC    IFR,B4
```

Example 5−7. Code to Clear an Individual Interrupt (INT6) and Read the Flag Register

```
        MVK    40h,B3
        MVC    B3,ICR
        NOP
        MVC    IFR,B4
```

5.3.4 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.

5.3.4.1 CPU State After $\overline{\text{RESET}}$

After $\overline{\text{RESET}}$, the control registers and bits contain the following values:

- AMR, ISR, ICR, IFR, and ISTP = 0
- IER = 1h
- IRP and NRP = undefined
- CSR bits 15−0 = 100h in little-endian mode
= 000h in big-endian mode

5.3.4.2 Returning From Nonmaskable Interrupts

The NMI return pointer register (NRP), shown in Figure 2−12 (page 2-22), 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 NRP (**B NRP**) in your interrupt service routine returns to the program flow when NMI servicing is complete. Example 5−8 shows how to return from an NMI.

Example 5−8. Code to Return From NMI

```
       B      NRP          ; return, sets NMIE
       NOP    5            ; delay slots
```

5.3.4.3 Returning From Maskable Interrupts

The interrupt return pointer register (IRP), shown in Figure 2−9 (page 2-19), 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 IRP (**B IRP**) in your interrupt service routine returns to the program flow when interrupt servicing is complete. Example 5−9 shows how to return from a maskable interrupt.

Example 5−9. Code to Return from a Maskable Interrupt

```
       B      IRP          ; return, moves PGIE to GIE
       NOP    5            ; delay slots
```

5.4 Interrupt Detection and Processing

When an interrupt occurs, it sets a flag in the interrupt flag register (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.

5.4.1 Setting the Nonreset Interrupt Flag

Figure 5−4 shows the processing of a nonreset interrupt (INTm). The flag (IFm) for INTm in the IFR is set following the low-to-high transition of the INTm signal on the CPU boundary. This transition is detected on a clock-cycle by clock-cycle basis and is not affected by memory stalls that might extend a CPU cycle. Once there is a low-to-high transition on an external interrupt pin (cycle 1), it takes two clock cycles for the signal to reach the CPU boundary (cycle 3). When the interrupt signal enters the CPU, it is has been detected (cycle 4). Two clock cycles after detection, the interrupt's corresponding flag bit in the IFR is set (cycle 6).

In Figure 5−4, IFm is set during CPU cycle 6. You could attempt to clear IFm by using an **MVC** instruction to write a 1 to bit m of the ICR in execute packet n + 3 (during CPU cycle 4). However, in this case, the automated write by the interrupt detection logic takes precedence and IFm remains set.

Figure 5−4 assumes 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.

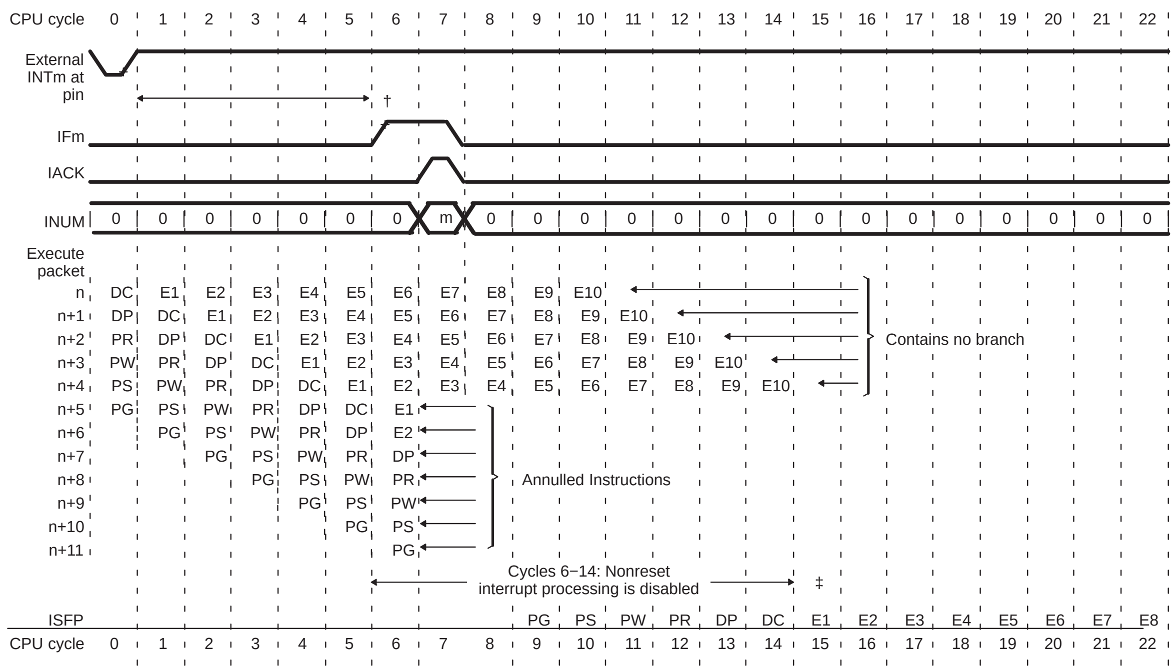
5.4.2 Conditions for Processing a Nonreset Interrupt

In clock cycle 4 of Figure 5−4, 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 5−4. Nonreset Interrupt Detection and Processing: Pipeline Operation



† IFm is set on the next CPU cycle boundary after a 4-clock cycle delay after the rising edge of INTm.

‡ After this point, interrupts are still disabled. All nonreset interrupts are disabled when NMIE = 0. All maskable interrupts are disabled when GIE = 0.

5.4.3 Actions Taken During Nonreset Interrupt Processing

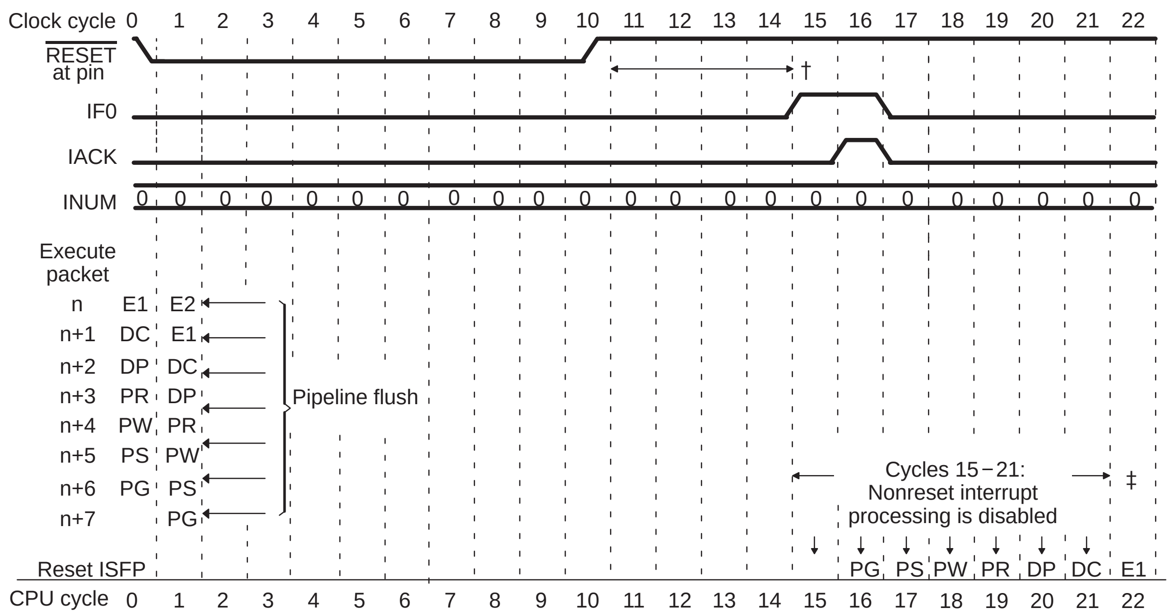
During CPU cycles 6 through 14 of Figure 5−4, the following interrupt processing actions occur:

- Processing of subsequent nonreset interrupts is disabled.
- For all interrupts except NMI, the PGIE bit is set to the value of the GIE bit and then the GIE bit is cleared.
- For NMI, the NMIE bit 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 NRP (in the case of NMI) or IRP (for all other interrupts).
- During cycle 7, IACK is asserted and the proper INUM*n* signals are asserted to indicate which interrupt is being processed. The timings for these signals in Figure 5−4 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 device-specific data manual for your timing values.
- IFm is cleared during cycle 8.
- 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 9.

5.4.4 Setting the $\overline{\text{RESET}}$ Interrupt Flag

$\overline{\text{RESET}}$ must be held low for a minimum of 10 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 5−5, IF0 is set during CPU cycle 15. This transition is detected on a clock-cycle by clock-cycle basis and is not affected by memory stalls that might extend a CPU cycle.

Figure 5−5. $\overline{\text{RESET}}$ Interrupt Detection and Processing: Pipeline Operation



† IF0 is set on the next CPU cycle boundary after a 4-clock cycle delay after the rising edge of $\overline{\text{RESET}}$.
‡ After this point, interrupts are still disabled. All nonreset interrupts are disabled when NMIE = 0. All maskable interrupts are disabled when GIE = 0.

5.4.5 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 5.3.4.1.

During CPU cycles 15 through 21 of Figure 5−5, the following reset processing actions occur:

- ❑ Processing of subsequent nonreset interrupts is disabled because the GIE and NMIE bits 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 INUM*n* 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 the GIE bit, the NMIE bit, and IER to allow interrupts to be processed.

5.5 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.

5.5.1 General Performance

- **Overhead**. Overhead for all CPU interrupts is 9 cycles. You can see this in Figure 5−4, where no new instructions are entering the E1 pipeline phase during CPU cycles 6 through 14.
- **Latency**. Interrupt latency is 13 cycles (21 cycles for $\overline{\text{RESET}}$). In Figure 5−4, although the interrupt is active in cycle 2, execution of interrupt service code does not begin until cycle 15.
- **Frequency**. The logic clears the nonreset interrupt (IFm) on cycle 8, with any incoming interrupt having highest priority. Thus, an interrupt is can be recognized every second cycle. Also, because a low-to-high transition is necessary, an interrupt can occur only every second cycle. However, the frequency of interrupt processing depends on the time required for interrupt service and whether you reenable interrupts during processing, thereby allowing nested interrupts. Effectively, only two occurrences of a specific interrupt can be recognized in two cycles.

5.5.2 Pipeline Interaction

Because the serial or parallel encoding of fetch packets does not affect the DC 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 5−4 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 NOPs (including the **IDLE** instruction) 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 NRP or IRP. This prevents returning to an **IDLE** instruction or a multicycle **NOP** that was interrupted.

5.6 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.

5.6.1 Single Assignment Programming

Using the same register to store different variables (called here: multiple assignment) can result in unpredictable operation when the code can be interrupted.

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.

Example 5−10 shows a code fragment which stores two variables into A1 using multiple assignment. Example 5−11 shows equivalent code using the single assignment programming method which stores the two variables into two different registers.

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 5−10. 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 5−11, the single assignment method is used. The register A1 is assigned only to the **ADD** input and not to the result of the **LDW**. Regardless of the value of A6 with or without an interrupt, A1 does not change before it is summed with A2. Result A3 is equal to A2.

Example 5−10. Code Without Single Assignment: Multiple Assignment of A1

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

Example 5−11. Code Using Single Assignment

```
        LDW    .D1    *A0,A6
        ADD    .L1    A1,A2,A3
        NOP    3
        MPY    .M1    A6,A4,A5      ; uses A6
```

5.6.2 Nested Interrupts

Generally, when the CPU enters an interrupt service routine, interrupts are disabled. However, when the interrupt service routine is for one of the maskable interrupts (INT4−INT15), an NMI can interrupt processing of the maskable interrupt. In other words, an NMI can interrupt a maskable interrupt, but neither an NMI nor a maskable interrupt can interrupt an NMI.

There may be times when you want to allow an interrupt service routine to be interrupted by another (particularly higher priority) interrupt. Even though the processor by default does not allow interrupt service routines to be interrupted unless the source is an NMI, it is possible to nest interrupts under software control. To allow nested interrupts, the interrupt service routine must perform the following initial steps in addition to its normal work of saving any registers (including control registers) that it modifies:

1) The contents of IRP (or NRP) must be saved
2) The contents of the PGIE bit must be saved
3) The GIE bit must be set to 1

Prior to returning from the interrupt service routine, the code must restore the registers saved above as follows:

1) The GIE bit must be first cleared to 0
2) The PGIE bit saved value must be restored
3) The IRP (or NRP) saved value must be restored

Although steps 2 and 3 above may be performed in any order, it is important that the GIE bit is cleared first. This means that the GIE and PGIE bits must be restored with separate writes to CSR. If these bits are not restored separately, then it is possible that the PGIE bit is overwritten by nested interrupt processing just as interrupts are being disabled.

Example 5−12 shows a simple assembly interrupt handler that allows nested interrupts on the C67x CPU. This example saves its context to the system stack, pointed to by B15. This assumes that the C runtime conventions are being followed. The example code is not optimized, to aid in readability.

Example 5−13 shows a C-based interrupt handler that allows nested interrupts. The steps are similar, although the compiler takes care of allocating the stack and saving CPU registers. For more information on using C to access control registers and write interrupt handlers, see the *TMS320C6000 Optimizing C Compiler Users Guide*, SPRU187.

Example 5−12. Assembly Interrupt Service Routine That Allows Nested Interrupts

```
_isr:
       STW    B0, *B15--[4]       ; Save B0, allocate 4 words of stack
       STW    B1, *B15[1]         ; Save B1 on stack

       MVC    IRP, B0
       STW    B0, *B15[2]         ; Save IRP on stack

       MVC    CSR, B0
       STW    B0, *B15[3]         ; Save CSR (and thus PGIE) on stack

       OR     B0, 1, B1
       MVC    B1, CSR             ; Enable interrupts

       ; Interrupt service code goes here.
       ; Interrupts may occur while this code executes.

       MVC    CSR, B0             ;\
       AND    B0, -2, B1          ; |-- Disable interrupts.
       MVC    B1, CSR             ;/    (Set GIE to 0)

       LDW    *B15[3], B0         ; get saved value of CSR into B0
       NOP    4                   ; wait for LDW *B15[3] to finish
       MVC    B0, CSR             ; Restore PGIE

       LDW    *B15[2], B0         ; get saved value of IRP into B1
       NOP    4
       MVC    B0, IRP             ; Restore IRP

       B      IRP                 ; Return from interrupt
||     LDW    *B15[1], B1         ; Restore B1

       LDW    *++B15[4], B0       ; Restore B0, release stack.

       NOP    4                   ; wait for B IRP and LDW to complete.
```

Example 5−13. C Interrupt Service Routine That Allows Nested Interrupts

```
/* c6x.h contains declarations of the C6x control registers             */
#include <c6x.h>

interrupt void isr(void)
{
       unsigned old_csr;
       unsigned old_irp;

       old_irp = IRP       ;/* Save IRP                                 */
       old_csr = CSR       ;/* Save CSR (and thus PGIE)                 */

       CSR = old_csr | 1   ;/* Enable interrupts                        */

       /* Interrupt service code goes here.                             */
       /* Interrupts may occur while this code executes                 */

       CSR = CSR & −2      ;/* Disable interrupts                       */
       CSR = old_csr       ;/* Restore CSR (and thus PGIE)              */
       IRP = old_irp       ;/* Restore IRP                              */
}
```

5.6.3 Manual Interrupt Processing

You can poll the IFR and IER to detect interrupts manually and then branch to the value held in the ISTP as shown below in Example 5−14.

The code sequence begins by copying the address of the highest priority interrupt from the ISTP to the register B2. The next instruction extracts the number of the interrupt, which is used later to clear the interrupt. The branch to the interrupt service routine comes next with a parallel instruction to set up the ICR word.

The last five instructions fill the delay slots of the branch. First, the 32-bit return address is stored in the B2 register and then copied to the interrupt return pointer (IRP). Finally, the number of the highest priority interrupt, stored in B1, is used to shift the ICR word in B1 to clear the interrupt.

Example 5−14. 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)
```

5.6.4 Traps

A trap behaves like an interrupt, but is created and controlled with software. The trap condition can be stored in any one of the conditional registers: A1, A2, B0, B1, or B2. If the trap condition is valid, a branch to the trap handler routine processes the trap and the return.

Example 5−15 and Example 5−16 show a trap call and the return code sequence, respectively. In the first code sequence, the address of the trap handler code is loaded into register B0 and the branch is called. In the delay slots of the branch, the context is saved in the B0 register, the GIE bit is cleared to disable maskable interrupts, and the return pointer is stored in the B1 register. If the trap handler were within the 21-bit offset for a branch using a displacement, the **MVKH** instructions could be eliminated, thus shortening the code sequence.

The trap is processed with the code located at the address pointed to by the label TRAP_HANDLER. If the B0 or B1 registers are needed in the trap handler, their contents must be stored to memory and restored before returning. The code shown in Example 5−16 should be included at the end of the trap handler code to restore the context prior to the trap and return to the TRAP_RETURN address.

Example 5−15. 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 5−16. Code Sequence for Trap Return

```
B       B1                                  ; return
MVC     B0,CSR                              ; restore CSR
NOP     4                                   ; delay slots
```

Appendix A

Instruction Compatibility

The C62x, C64x, and C67x DSPs share an instruction set. All of the instructions valid for the C62x DSP are also valid for the C67x and C67x+ DSPs. The C67x DSP adds specific instructions for 32-bit integer multiply, doubleword load, and floating-point operations. Table A−1 lists the instructions that are common to the C62x, C64x, C67x, and C67x+ DSPs.

Table A−1. Instruction Compatibility Between C62x, C64x, C67x, and C67x+ DSPs

Instruction	Page	C62x DSP	C64x DSP	C67x DSP	C67x+ DSP
ABS	3-37	✓	✓	✓	✓
ABSDP	3-39			✓	✓
ABSSP	3-41			✓	✓
ADD	3-43	✓	✓	✓	✓
ADDAB	3-47	✓	✓	✓	✓
ADDAD	3-49			✓	✓
ADDAH	3-51	✓	✓	✓	✓
ADDAW	3-53	✓	✓	✓	✓
ADDDP	3-55			✓	✓
ADDK	3-58	✓	✓	✓	✓
ADDSP	3-59			✓	✓
ADDU	3-62	✓	✓	✓	✓
ADD2	3-64	✓	✓	✓	✓
AND	3-66	✓	✓	✓	✓

Table A−1. Instruction Compatibility Between C62x, C64x, C67x, and C67x+ DSPs (Continued)

Instruction	Page	C62x DSP	C64x DSP	C67x DSP	C67x+ DSP
B displacement	3-68	✓	✓	✓	✓
B register	3-70	✓	✓	✓	✓
B IRP	3-72	✓	✓	✓	✓
B NRP	3-74	✓	✓	✓	✓
CLR	3-76	✓	✓	✓	✓
CMPEQ	3-79	✓	✓	✓	✓
CMPEQDP	3-81			✓	✓
CMPEQSP	3-83			✓	✓
CMPGT	3-85	✓	✓	✓	✓
CMPGTDP	3-88			✓	✓
CMPGTSP	3-90			✓	✓
CMPGTU	3-92	✓	✓	✓	✓
CMPLT	3-94	✓	✓	✓	✓
CMPLTDP	3-97			✓	✓
CMPLTSP	3-99			✓	✓
CMPLTU	3-101	✓	✓	✓	✓
DPINT	3-103			✓	✓
DPSP	3-105			✓	✓
DPTRUNC	3-107			✓	✓
EXT	3-109	✓	✓	✓	✓
EXTU	3-112	✓	✓	✓	✓
IDLE	3-115	✓	✓	✓	✓
INTDP	3-116			✓	✓
INTDPU	3-118			✓	✓

Table A−1. Instruction Compatibility Between C62x, C64x, C67x, and C67x+ DSPs (Continued)

Instruction	Page	C62x DSP	C64x DSP	C67x DSP	C67x+ DSP
INTSP	3-120			✓	✓
INTSPU	3-121			✓	✓
LDB memory	3-122	✓	✓	✓	✓
LDB memory (15-bit offset)	3-125	✓	✓	✓	✓
LDBU memory	3-122	✓	✓	✓	✓
LDBU memory (15-bit offset)	3-125	✓	✓	✓	✓
LDDW	3-127			✓	✓
LDH memory	3-130	✓	✓	✓	✓
LDH memory (15-bit offset)	3-133	✓	✓	✓	✓
LDHU memory	3-130	✓	✓	✓	✓
LDHU memory (15-bit offset)	3-133	✓	✓	✓	✓
LDW memory	3-135	✓	✓	✓	✓
LDW memory (15-bit offset)	3-138	✓	✓	✓	✓
LMBD	3-140	✓	✓	✓	✓
MPY	3-142	✓	✓	✓	✓
MPYDP	3-144			✓	✓
MPYH	3-146	✓	✓	✓	✓
MPYHL	3-148	✓	✓	✓	✓
MPYHLU	3-150	✓	✓	✓	✓
MPYHSLU	3-151	✓	✓	✓	✓
MPYHSU	3-152	✓	✓	✓	✓
MPYHU	3-153	✓	✓	✓	✓
MPYHULS	3-154	✓	✓	✓	✓
MPYHUS	3-155	✓	✓	✓	✓

Table A−1. Instruction Compatibility Between C62x, C64x, C67x, and C67x+ DSPs (Continued)

Instruction	Page	C62x DSP	C64x DSP	C67x DSP	C67x+ DSP
MPYI	3-156			✓	✓
MPYID	3-158			✓	✓
MPYLH	3-160	✓	✓	✓	✓
MPYLHU	3-162	✓	✓	✓	✓
MPYLSHU	3-163	✓	✓	✓	✓
MPYLUHS	3-164	✓	✓	✓	✓
MPYSP	3-165			✓	✓
MPYSPDP	3-167				✓
MPYSP2DP	3-169				✓
MPYSU	3-171	✓	✓	✓	✓
MPYU	3-173	✓	✓	✓	✓
MPYUS	3-175	✓	✓	✓	✓
MV	3-177	✓	✓	✓	✓
MVC	3-179	✓	✓	✓	✓
MVK	3-182	✓	✓	✓	✓
MVKH	3-184	✓	✓	✓	✓
MVKL	3-186	✓	✓	✓	✓
MVKLH	3-184	✓	✓	✓	✓
NEG	3-188	✓	✓	✓	✓
NOP	3-189	✓	✓	✓	✓
NORM	3-191	✓	✓	✓	✓
NOT	3-193	✓	✓	✓	✓
OR	3-194	✓	✓	✓	✓
RCPDP	3-196			✓	✓
RCPSP	3-198			✓	✓

Table A−1. Instruction Compatibility Between C62x, C64x, C67x, and C67x+ DSPs (Continued)

Instruction	Page	C62x DSP	C64x DSP	C67x DSP	C67x+ DSP
RSQRDP	3-200			✓	✓
RSQRSP	3-202			✓	✓
SADD	3-204	✓	✓	✓	✓
SAT	3-207	✓	✓	✓	✓
SET	3-209	✓	✓	✓	✓
SHL	3-212	✓	✓	✓	✓
SHR	3-214	✓	✓	✓	✓
SHRU	3-216	✓	✓	✓	✓
SMPY	3-218	✓	✓	✓	✓
SMPYH	3-220	✓	✓	✓	✓
SMPYHL	3-221	✓	✓	✓	✓
SMPYLH	3-223	✓	✓	✓	✓
SPDP	3-225			✓	✓
SPINT	3-227			✓	✓
SPTRUNC	3-229			✓	✓
SSHL	3-231	✓	✓	✓	✓
SSUB	3-233	✓	✓	✓	✓
STB memory	3-235	✓	✓	✓	✓
STB memory (15-bit offset)	3-237	✓	✓	✓	✓
STH memory	3-239	✓	✓	✓	✓
STH memory (15-bit offset)	3-242	✓	✓	✓	✓
STW memory	3-244	✓	✓	✓	✓
STW memory (15-bit offset)	3-246	✓	✓	✓	✓

Table A−1. Instruction Compatibility Between C62x, C64x, C67x, and C67x+ DSPs (Continued)

Instruction	Page	C62x DSP	C64x DSP	C67x DSP	C67x+ DSP
SUB	3-248	✓	✓	✓	✓
SUBAB	3-252	✓	✓	✓	✓
SUBAH	3-254	✓	✓	✓	✓
SUBAW	3-255	✓	✓	✓	✓
SUBC	3-257	✓	✓	✓	✓
SUBDP	3-259			✓	✓
SUBSP	3-262			✓	✓
SUBU	3-265	✓	✓	✓	✓
SUB2	3-267	✓	✓	✓	✓
XOR	3-269	✓	✓	✓	✓
ZERO	3-271	✓	✓	✓	✓

Appendix B

Mapping Between Instruction and Functional Unit

Table B−1 lists the instructions that execute on each functional unit.

Table B−1. Functional Unit to Instruction Mapping

	Functional Unit			
Instruction	.L Unit	.M Unit	.S Unit	.D Unit
ABS	✓			
ABSDP			✓	
ABSSP			✓	
ADD	✓		✓	✓
ADDAB				✓
ADDAD				✓
ADDAH				✓
ADDAW				✓
ADDDP	✓		✓§	
ADDK			✓	
ADDSP	✓		✓§	
ADDU	✓			
ADD2			✓	
AND	✓		✓	

† S2 only
‡ D2 only
§ C67x+ DSP-specific instruction

Table B−1. Functional Unit to Instruction Mapping (Continued)

	Functional Unit			
Instruction	**.L Unit**	**.M Unit**	**.S Unit**	**.D Unit**
B displacement			✓	
B register			✓†	
B IRP			✓†	
B NRP			✓†	
CLR			✓	
CMPEQ	✓			
CMPEQDP			✓	
CMPEQSP			✓	
CMPGT	✓			
CMPGTDP			✓	
CMPGTSP			✓	
CMPGTU	✓			
CMPLT	✓			
CMPLTDP			✓	
CMPLTSP			✓	
CMPLTU	✓			
DPINT	✓			
DPSP	✓			
DPTRUNC	✓			
EXT			✓	
EXTU			✓	
IDLE				

† S2 only
‡ D2 only
§ C67x+ DSP-specific instruction

Table B−1. Functional Unit to Instruction Mapping (Continued)

	Functional Unit			
Instruction	**.L Unit**	**.M Unit**	**.S Unit**	**.D Unit**
INTDP	✓			
INTDPU	✓			
INTSP	✓			
INTSPU	✓			
LDB memory				✓
LDB memory (15-bit offset)				✓‡
LDBU memory				✓
LDBU memory (15-bit offset)				✓‡
LDDW				✓
LDH memory				✓
LDH memory (15-bit offset)				✓‡
LDHU memory				✓
LDHU memory (15-bit offset)				✓‡
LDW memory				✓
LDW memory (15-bit offset)				✓‡
LMBD	✓			
MPY		✓		
MPYDP		✓		
MPYH		✓		
MPYHL		✓		
MPYHLU		✓		
MPYHSLU		✓		
MPYHSU		✓		

† S2 only
‡ D2 only
§ C67x+ DSP-specific instruction

Table B−1. Functional Unit to Instruction Mapping (Continued)

	Functional Unit			
Instruction	.L Unit	.M Unit	.S Unit	.D Unit
MPYHU		✓		
MPYHULS		✓		
MPYHUS		✓		
MPYI		✓		
MPYID		✓		
MPYLH		✓		
MPYLHU		✓		
MPYLSHU		✓		
MPYLUHS		✓		
MPYSP		✓		
MPYSPDP§		✓		
MPYSP2DP§		✓		
MPYSU		✓		
MPYU		✓		
MPYUS		✓		
MV	✓		✓	✓
MVC			✓†	
MVK			✓	
MVKH			✓	
MVKL			✓	
MVKLH			✓	
NEG	✓		✓	
NOP				

† S2 only
‡ D2 only
§ C67x+ DSP-specific instruction

Table B−1. Functional Unit to Instruction Mapping (Continued)

	Functional Unit			
Instruction	**.L Unit**	**.M Unit**	**.S Unit**	**.D Unit**
NORM	✓			
NOT	✓		✓	
OR	✓		✓	
RCPDP			✓	
RCPSP			✓	
RSQRDP			✓	
RSQRSP			✓	
SADD	✓			
SAT	✓			
SET			✓	
SHL			✓	
SHR			✓	
SHRU			✓	
SMPY		✓		
SMPYH		✓		
SMPYHL		✓		
SMPYLH		✓		
SPDP			✓	
SPINT	✓			
SPTRUNC	✓			
SSHL			✓	
SSUB	✓			
STB memory				✓

† S2 only
‡ D2 only
§ C67x+ DSP-specific instruction

Table B−1. Functional Unit to Instruction Mapping (Continued)

	Functional Unit			
Instruction	**.L Unit**	**.M Unit**	**.S Unit**	**.D Unit**
STB memory (15-bit offset)				✓‡
STH memory				✓
STH memory (15-bit offset)				✓‡
STW memory				✓
STW memory (15-bit offset)				✓‡
SUB	✓		✓	✓
SUBAB				✓
SUBAH				✓
SUBAW				✓
SUBC	✓			
SUBDP	✓		✓§	
SUBSP	✓		✓§	
SUBU	✓		✓	
SUB2			✓	
XOR	✓		✓	
ZERO	✓		✓	✓

† S2 only
‡ D2 only
§ C67x+ DSP-specific instruction

Appendix C

.D Unit Instructions and Opcode Maps

This appendix lists the instructions that execute in the .D functional unit and illustrates the opcode maps for these instructions.

Topic **Page**

C.1 Instructions Executing in the .D Functional Unit

Table C−1 lists the instructions that execute in the .D functional unit.

Table C−1. Instructions Executing in the .D Functional Unit

Instruction	Instruction
ADD	LDW memory
ADDAB	LDW memory (15-bit offset)‡
ADDAD	MV
ADDAH	STB memory
ADDAW	STB memory (15-bit offset)‡
LDB memory	STH memory
LDB memory (15-bit offset)‡	STH memory (15-bit offset)‡
LDBU memory	STW memory
LDBU memory (15-bit offset)‡	STW memory (15-bit offset)‡
LDDW	SUB
LDH memory	SUBAB
LDH memory (15-bit offset)‡	SUBAH
LDHU memory	SUBAW
LDHU memory (15-bit offset)‡	ZERO

† S2 only
‡ D2 only

C.2 Opcode Map Symbols and Meanings

Table C−2 lists the symbols and meanings used in the opcode maps.

Table C−2. .D Unit Opcode Map Symbol Definitions

Symbol	Meaning
baseR	base address register
creg	3-bit field specifying a conditional register
dst	destination. For compact instructions, *dst* is coded as an offset from either A16 or B16 depending on the value of the *t* bit.
mode	addressing mode, see Table C−3
offsetR	register offset
op	opfield; field within opcode that specifies a unique instruction
p	parallel execution; 0 = next instruction is not executed in parallel, 1 = next instruction is executed in parallel
r	LDDW instruction
s	side A or B for destination; 0 = side A, 1 = side B. For compact instructions, side of base address (*ptr*) register; 0 = side A, 1 = side B.
src	source. For compact instructions, *src* is coded as an offset from either A16 or B16 depending on the value of the *t* bit.
src1	source 1
src2	source 2
x	cross path for *src2*; 0 = do not use cross path, 1 = use cross path
y	.D1 or .D2 unit; 0 = .D1 unit, 1 = .D2 unit
z	test for equality with zero or nonzero

Table C−3. Address Generator Options for Load/Store

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

C.3 32-Bit Opcode Maps

The C67x CPU 32-bit opcodes used in the .D unit are mapped in Figure C−1 through Figure C−4.

Figure C−1. 1 or 2 Sources Instruction Format

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

Figure C−2. Extended .D Unit 1 or 2 Sources Instruction Format

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			4					1	1

Figure C−3. Load/Store Basic Operations

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

Figure C−4. Load/Store Long-Immediate Operations

31–29	28	27–23	22–8	7	6–4	3	2	1	0
creg	*z*	*dst*	*offsetR*	*y*	*op*	1	1	*s*	*p*
3	1	5	15	1	3			1	1

Appendix D

.L Unit Instructions and Opcode Maps

This appendix lists the instructions that execute in the .L functional unit and illustrates the opcode maps for these instructions.

Topic **Page**

D.1 Instructions Executing in the .L Functional Unit

Table D−1 lists the instructions that execute in the .L functional unit.

Table D−1. Instructions Executing in the .L Functional Unit

Instruction	Instruction
ABS	LMBD
ADD	MV
ADDDP	NEG
ADDSP	NORM
ADDU	NOT
AND	OR
CMPEQ	SADD
CMPGT	SAT
CMPGTU	SPINT
CMPLT	SPTRUNC
CMPLTU	SSUB
DPINT	SUB
DPSP	SUBC
DPTRUNC	SUBDP
INTDP	SUBSP
INTDPU	SUBU
INTSP	XOR
INTSPU	ZERO

D.2 Opcode Map Symbols and Meanings

Table D−2 lists the symbols and meanings used in the opcode maps.

Table D−2. .L Unit Opcode Map Symbol Definitions

Symbol	Meaning
creg	3-bit field specifying a conditional register
dst	destination
op	opfield; field within opcode that specifies a unique instruction
p	parallel execution; 0 = next instruction is not executed in parallel, 1 = next instruction is executed in parallel
s	side A or B for destination; 0 = side A, 1 = side B
src1	source 1
src2	source 2
x	cross path for *src2*; 0 = do not use cross path, 1 = use cross path
z	test for equality with zero or nonzero

D.3 32-Bit Opcode Maps

The C67x CPU 32-bit opcodes used in the .L unit are mapped in Figure D−1 through Figure D−3.

Figure D−1. 1 or 2 Sources Instruction Format

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				1	1

Figure D−2. 1 or 2 Sources, Nonconditional Instruction Format

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

Figure D−3. Unary Instruction Format

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

Appendix E

.M Unit Instructions and Opcode Maps

This appendix lists the instructions that execute in the .M functional unit and illustrates the opcode maps for these instructions.

Topic **Page**

E.1 Instructions Executing in the .M Functional Unit

Table E−1 lists the instructions that execute in the .M functional unit.

Table E−1. Instructions Executing in the .M Functional Unit

Instruction	Instruction
MPY	MPYLHU
MPYDP	MPYLSHU
MPYH	MPYLUHS
MPYHL	MPYSP
MPYHLU	MPYSPDP[§]
MPYHSLU	MPYSP2DP[§]
MPYHSU	MPYSU
MPYHU	MPYU
MPYHULS	MPYUS
MPYHUS	SMPY
MPYI	SMPYH
MPYID	SMPYHL
MPYLH	SMPYLH

[§] C67x+ DSP-specific instruction

E.2 Opcode Map Symbols and Meanings

Table E−2 lists the symbols and meanings used in the opcode maps.

Table E−2. .M Unit Opcode Map Symbol Definitions

Symbol	**Meaning**
creg	3-bit field specifying a conditional register
dst	destination
op	opfield; field within opcode that specifies a unique instruction
p	parallel execution; 0 = next instruction is not executed in parallel, 1 = next instruction is executed in parallel
s	side A or B for destination; 0 = side A, 1 = side B
src1	source 1
src2	source 2
x	cross path for *src2*; 0 = do not use cross path, 1 = use cross path
z	test for equality with zero or nonzero

E.3 32-Bit Opcode Maps

The C67x CPU 32-bit opcodes used in the .M unit are mapped in Figure E−1 through Figure E−3.

Figure E−1. Extended M-Unit with Compound Operations

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		5					1	1

Figure E−2. Extended .M Unit 1 or 2 Sources, Nonconditional Instruction Format

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

Figure E−3. Extended .M-Unit Unary Instruction Format

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

Appendix F

.S Unit Instructions and Opcode Maps

This appendix lists the instructions that execute in the .S functional unit and illustrates the opcode maps for these instructions.

Topic **Page**

F.1 Instructions Executing in the .S Functional Unit

Table F−1 lists the instructions that execute in the .S functional unit.

Table F−1. Instructions Executing in the .S Functional Unit

Instruction	Instruction
ABSDP	MVKH
ABSSP	MVKL
ADD	MVKLH
ADDDP§	NEG
ADDK	NOT
ADDSP§	OR
ADD2	RCPDP
AND	RCPSP
B displacement	RSQRDP
B register†	RSQRSP
B IRP†	SET
B NRP†	SHL
CLR	SHR
CMPEQDP	SHRU
CMPEQSP	SPDP
CMPGTDP	SSHL
CMPGTSP	SUB
CMPLTDP	SUBDP§
CMPLTSP	SUBSP§
EXT	SUBU
EXTU	SUB2
MV	XOR
MVC†	ZERO
MVK	

† S2 only
§ C67x+ DSP-specific instruction

F.2 Opcode Map Symbols and Meanings

Table F−2 lists the symbols and meanings used in the opcode maps.

Table F−2. .S Unit Opcode Map Symbol Definitions

Symbol	Meaning
creg	3-bit field specifying a conditional register
csta	constant a
cstb	constant b
cstn	n-bit constant field
dst	destination
h	MVK or MVKH/MVKLH instruction; 0 = MVK, 1 = MVKH/MVKLH
op	opfield; field within opcode that specifies a unique instruction
p	parallel execution; 0 = next instruction is not executed in parallel, 1 = next instruction is executed in parallel
s	side A or B for destination; 0 = side A, 1 = side B
src1	source 1
src2	source 2
x	cross path for *src2*; 0 = do not use cross path, 1 = use cross path
z	test for equality with zero or nonzero

F.3 32-Bit Opcode Maps

The C67x CPU 32-bit opcodes used in the .S unit are mapped in Figure F−1 through Figure F−11.

Figure F−1. 1 or 2 Sources Instruction Format

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					1	1

Figure F−2. Extended .S Unit 1 or 2 Sources Instruction Format

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			4					1	1

Figure F−3. Extended .S Unit 1 or 2 Sources, Nonconditional Instruction Format

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

Figure F−4. Unary Instruction Format

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

Figure F−5. Extended .S Unit Branch Conditional, Immediate Instruction Format

31–29	28	27–7	6	5	4	3	2	1	0
creg	*z*	*cst21*	0	0	1	0	0	*s*	*p*
3	1	21						1	1

Figure F−6. Call Unconditional, Immediate with Implied NOP 5 Instruction Format

31 29	28	27 7	6	5	4	3	2	1	0
0 0 0	*z*	*cst21*	0	0	1	0	0	*s*	*p*
	1	21						1	1

Figure F−7. Branch with NOP Constant Instruction Format

31 29	28	27 16	15 13	12	11 6	5	4	3	2	1	0
creg	*z*	*src2*	*src1*	0	0 0 0 1 0 0	1	0	0	0	*s*	*p*
3	1	12	3							1	1

Figure F−8. Branch with NOP Register Instruction Format

31 29	28	27 23	22 18	17 16	15 13	12	11 6	5	4	3	2	1	0
creg	*z*	0 0 0 0 1	*src2*	0 0	*src1*	x	0 0 1 1 0 1	1	0	0	0	*s*	*p*
3	1		5		3	1						1	1

Figure F−9. Branch Instruction Format

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

Figure F−10. MVK Instruction Format

31 29	28	27 23	22 7	6	5	4	3	2	1	0
creg	*z*	*dst*	*cst16*	*h*	1	0	1	0	*s*	*p*
3	1	5	16						1	1

Figure F−11.Field Operations

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	1	5	5	5	5	2					1	1

Appendix G

No Unit Specified Instructions and Opcode Maps

This appendix lists the instructions that execute with no unit specified and illustrates the opcode maps for these instructions.

For a list of the instructions that execute in the .D functional unit, see Appendix C. For a list of the instructions that execute in the .L functional unit, see Appendix D. For a list of the instructions that execute in the .M functional unit, see Appendix E. For a list of the instructions that execute in the .S functional unit, see Appendix F.

Topic **Page**

G.1 Instructions Executing With No Unit Specified

Table G−1 lists the instructions that execute with no unit specified.

Table G−1. Instructions Executing With No Unit Specified

Instruction
IDLE
NOP

G.2 Opcode Map Symbols and Meanings

Table G−2 lists the symbols and meanings used in the opcode maps.

Table G−2. No Unit Specified Instructions Opcode Map Symbol Definitions

Symbol	Meaning
creg	3-bit field specifying a conditional register
csta	constant a
cstb	constant b
cstn	n-bit constant field
ii_n	bit n of the constant *ii*
N3	3-bit field
op	opfield; field within opcode that specifies a unique instruction
p	parallel execution; 0 = next instruction is not executed in parallel, 1 = next instruction is executed in parallel
s	side A or B for destination; 0 = side A, 1 = side B.
stg_n	bit n of the constant *stg*
z	test for equality with zero or nonzero

G.3 32-Bit Opcode Maps

The C67x CPU 32-bit opcodes used in the no unit instructions are mapped in Figure G−1 through Figure G−3.

Figure G−1. Loop Buffer Instruction Format

31 29	28	27 23	22 18	17	16 13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*cstb*	*csta*	1	*op*	0	0	0	0	0	0	0	0	0	0	0	*s*	*p*
3	1	5	5		4												1	1

Figure G−2. NOP and IDLE Instruction Format

31 29	28	27 18	17	16 13	12	11	10	9	8	7	6	5	4	3	2	1	0
0 0 0	1	Reserved (0)	0	*op*	0	0	0	0	0	0	0	0	0	0	0	*s*	*p*
		10		4												1	1

Figure G−3. Emulation/Control Instruction Format

31 29	28	27 23	22 18	17	16 13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	Reserved (0)	*cst5*	0	*op*	0	0	0	0	0	0	0	0	0	0	0	*s*	*p*
3	1	5	5		4												1	1

Appendix H

Revision History

Table H−1 lists the changes made since the previous version of this document.

Table H−1. Document Revision History

Page	Additions/Modifications/Deletions
3-14	Changed second paragraph. Deleted third paragraph.
3-77	Changed Description.
3-210	Changed Description.
3-211	Changed B1 register value.
3-271	Added Example.

Index

R