TMS320C64x/C64x+ DSP CPU and Instruction Set

Reference Guide

Literature Number: SPRU732J
July 2010

TEXAS INSTRUMENTS

Contents

List of Figures

List of Tables

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 C62x™ and C64x™ DSPs are code-compatible.

The TMS320C64x+™ DSP is an enhancement of the C64x DSP with added functionality and an expanded instruction set. This document describes the CPU architecture, pipeline, instruction set, and interrupts of the C64x and C64x+ DSPs.

Notational Conventions

This document uses the following conventions.

- Any reference to the C64x DSP or C64x CPU also applies, unless otherwise noted, to the C64x+ DSP and C64x+ 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.

SPRU190 — *TMS320C6000 DSP Peripherals Overview Reference Guide.* Provides an overview and briefly describes the peripherals available on the TMS320C6000 family of digital signal processors (DSPs).

SPRU395 — *TMS320C64x Technical Overview.* Provides an introduction to the TMS320C64x digital signal processors (DSPs) of the TMS320C6000 DSP family.

SPRU656 — *TMS320C6000 DSP Cache User's Guide.* Explains the fundamentals of memory caches and describes how the two-level cache-based internal memory architecture in the TMS320C621x/C671x/C64x digital signal processors (DSPs) of the TMS320C6000 DSP family can be efficiently used in DSP applications. Shows how to maintain coherence with external memory, how to use DMA to reduce memory latencies, and how to optimize your code to improve cache efficiency. The internal memory architecture in the C621x/C671x/C64x DSPs is organized in a two-level hierarchy consisting of a dedicated program cache (L1P) and a dedicated data cache (L1D) on the first level. Accesses by the CPU to the these first level caches can complete without CPU pipeline stalls. If the data requested by the CPU is not contained in cache, it is fetched from the next lower memory level, L2 or external memory.

SPRU862 — *TMS320C64x+ DSP Cache User's Guide.* Explains the fundamentals of memory caches and describes how the two-level cache-based internal memory architecture in the TMS320C64x+ digital signal processor (DSP) of the TMS320C6000 DSP family can be efficiently used in DSP applications. Shows how to maintain coherence with external memory, how to use DMA to reduce memory latencies, and how to optimize your code to improve cache efficiency. The internal memory architecture in the C64x+ DSP is organized in a two-level hierarchy consisting of a dedicated program cache (L1P) and a dedicated data cache (L1D) on the first level. Accesses by the CPU to the these first level caches can complete without CPU pipeline stalls. If the data requested by the CPU is not contained in cache, it is fetched from the next lower memory level, L2 or external memory.

SPRU871 — *TMS320C64x+ DSP Megamodule Reference Guide.* Describes the TMS320C64x+ digital signal processor (DSP) megamodule. Included is a discussion on the internal direct memory access (IDMA) controller, the interrupt controller, the power-down controller, memory protection, bandwidth management, and the memory and cache.

SPRAA84 — *TMS320C64x to TMS320C64x+ CPU Migration Guide.* Describes migrating from the Texas Instruments TMS320C64x digital signal processor (DSP) to the TMS320C64x+ DSP. The objective of this document is to indicate differences between the two cores. Functionality in the devices that is identical is not included.

SPRU186 — *TMS320C6000 Assembly Language Tools User's Guide.* Describes the assembly language tools (assembler, linker, and other tools used to develop assembly language code), assembler directives, macros, common object file format, and symbolic debugging directives for the TMS320C6000 platform of devices (including the C64x+ and C67x+ generations).

SPRU187 — *TMS320C6000 Optimizing Compiler User's Guide.* Describes the TMS320C6000 C compiler and the assembly optimizer. This C compiler accepts ANSI standard C source code and produces assembly language source code for the TMS320C6000 platform of devices (including the C64x+ and C67x+ generations). The assembly optimizer helps you optimize your assembly code.

SPRU198 — *TMS320C6000 Programmer's Guide.* Reference for programming the TMS320C6000 digital signal processors (DSPs). Before you use this manual, you should install your code generation and debugging tools. Includes a brief description of the C6000 DSP architecture and code development flow, includes C code examples and discusses optimization methods for the C code, describes the structure of assembly code and includes examples and discusses optimizations for the assembly code, and describes programming considerations for the C64x DSP.

TEXAS INSTRUMENTS

Chapter 1
SPRU732J–July 2010

Introduction

The TMS320C6000™ digital signal processor (DSP) platform is part of the TMS320™ DSP family. The TMS320C62x™ DSP generation and the TMS320C64x™ DSP generation comprise fixed-point devices in the C6000™ DSP platform, and the TMS320C67x™ DSP generation comprises floating-point devices in the C6000 DSP platform. The C62x™ and C64x™ DSPs are code-compatible. 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 TMS320C64x+™ DSP is an enhancement of the C64x DSP with added functionality and an expanded instruction set.

Any reference to the C64x DSP or C64x CPU also applies, unless otherwise noted, to the C64x+ DSP and C64x+ 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 8000 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	Digital radios/TVs	Disk drive control
Antiskid brakes	Educational toys	Engine control
Cellular telephones	Music synthesizers	Laser printer control
Digital radios	Pagers	Motor control
Engine control	Power tools	Robotics control
Global positioning	Radar detectors	Servo control
Navigation	Solid-state answering machines	
Vibration analysis		
Voice commands		
General-Purpose	**Graphics/Imaging**	**Industrial**
Adaptive filtering	3-D transformations	Numeric control
Convolution	Animation/digital maps	Power-line monitoring
Correlation	Homomorphic processing	Robotics
Digital filtering	Image compression/transmission	Security access
Fast Fourier transforms	Image enhancement	
Hilbert transforms	Pattern recognition	
Waveform generation	Robot vision	
Windowing	Workstations	
Instrumentation	**Medical**	**Military**
Digital filtering	Diagnostic equipment	Image processing
Function generation	Fetal monitoring	Missile guidance
Pattern matching	Hearing aids	Navigation
Phase-locked loops	Patient monitoring	Radar processing
Seismic processing	Prosthetics	Radio frequency modems
Spectrum analysis	Ultrasound equipment	Secure communications
Transient analysis		Sonar processing
Telecommunications		**Voice/Speech**
1200- to 56 600-bps modems	Faxing	Speaker verification
Adaptive equalizers	Future terminals	Speech enhancement
ADPCM transcoders	Line repeaters	Speech recognition
Base stations	Personal communications systems (PCS)	Speech synthesis
Cellular telephones	Personal digital assistants (PDA)	Speech vocoding
Channel multiplexing	Speaker phones	Text-to-speech
Data encryption	Spread spectrum communications	Voice mail
Digital PBXs	Digital subscriber loop (xDSL)	
Digital speech interpolation (DSI)	Video conferencing	
DTMF encoding/decoding	X.25 packet switching	
Echo cancellation		

1.3 TMS320C64x DSP Features and Options

The C6000 devices execute up to eight 32-bit instructions per cycle. The C64x CPU consists of 64 general-purpose 32-bit registers and eight functional units. These eight functional units contain:

- Two multipliers
- Six ALUs

The C6000 generation has a complete set of optimized development tools, including an efficient C compiler, an assembly optimizer for simplified assembly-language programming and scheduling, and a Windows® operating system-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 most 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 C64x and C64x+ devices include these additional features:

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

In addition to the features of the C64x device, the C64x+ devices include these additional features:

- Compact instructions: Common instructions (AND, ADD, LD, MPY) have 16-bit versions to reduce code size.
- Protected mode operation: A two-level system of privileged program execution to support higher capability operating systems and system features such as memory protection.
- Exceptions support for error detection and program redirection to provide robust code execution
- Hardware support for modulo loop operation to reduce code size
- Each multiplier can perform 32 × 32 bit multiplies
- Additional instructions to support complex multiplies allowing up to eight 16-bit multiply/add/subtracts per clock cycle

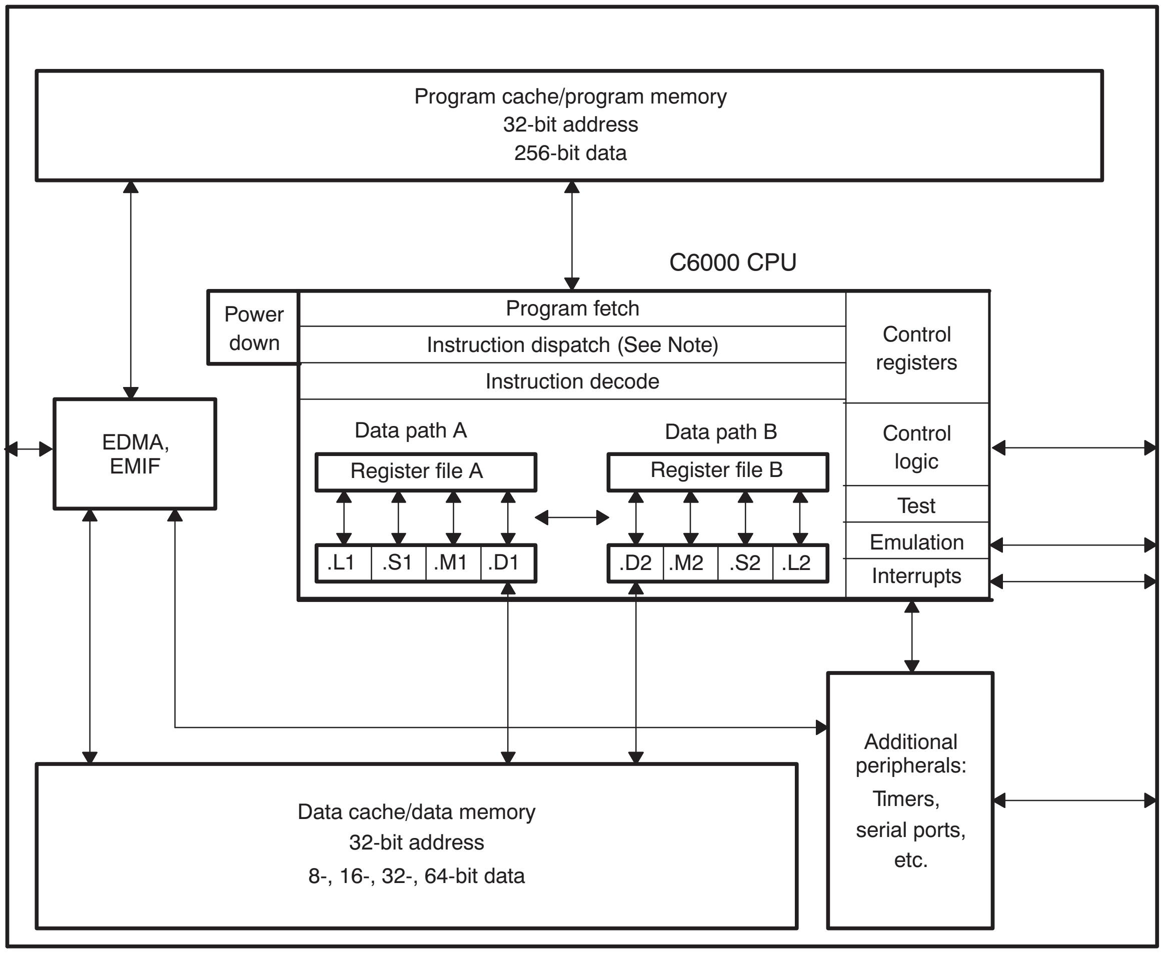
The VelociTI architecture of the C6000 platform of devices make them the first off-the-shelf DSPs to use advanced VLIW to achieve high performance through increased instruction-level parallelism. A traditional VLIW architecture consists of multiple execution units running in parallel, performing multiple 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 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 TMS320C64x/C64x+ DSP Architecture

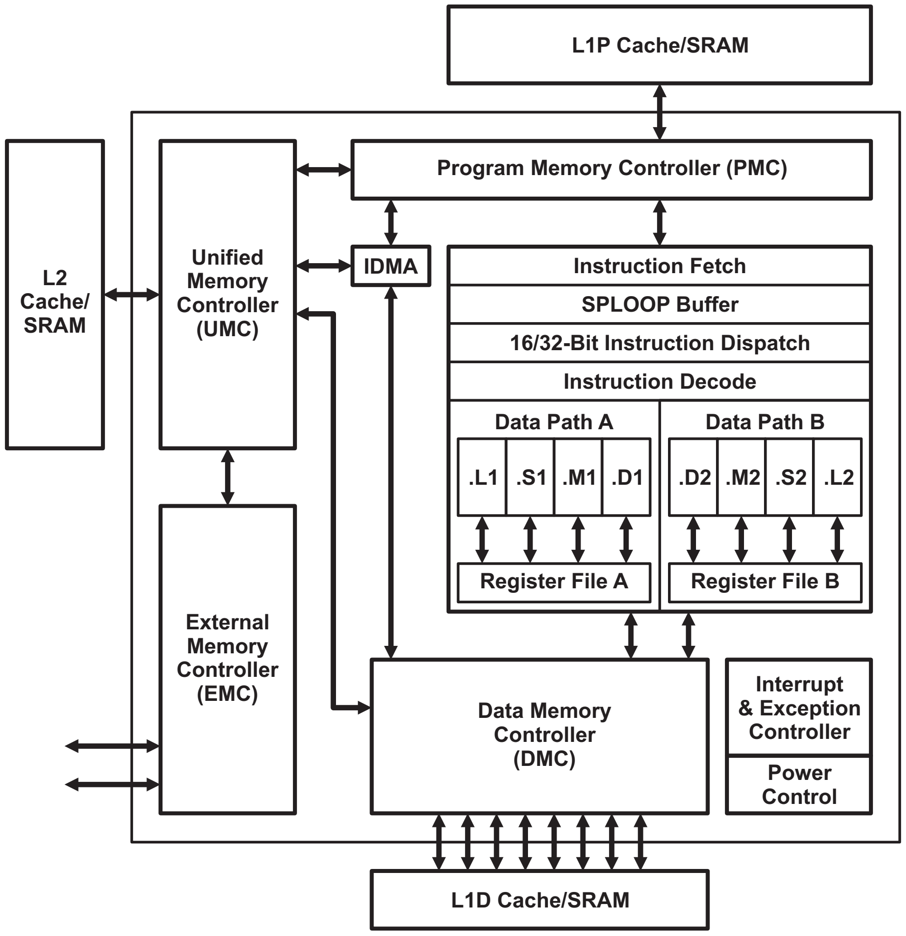
Figure 1-1 is the block diagram for the C64x DSP. Figure 1-2 is the block diagram for the C64x+ DSP. The C6000 devices come with program memory, which, on some devices, can be used as a program cache. The devices also have varying sizes of data memory. Peripherals such as a direct memory access (DMA) 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. TMS320C64x DSP Block Diagram



Note: The instruction dispatch unit has advanced instruction packing.

Figure 1-2. TMS320C64x+ DSP Block Diagram


L1P Cache/SRAM
Program Memory Controller (PMC)
L2 Cache/ SRAM
Unified Memory Controller (UMC)
IDMA
Instruction Fetch
SPLOOP Buffer
16/32-Bit Instruction Dispatch
Instruction Decode
Data Path A
Data Path B
.L1
.S1
.M1
.D1
.D2
.M2
.S2
.L2
Register File A
Register File B
External Memory Controller (EMC)
Data Memory Controller (DMC)
Interrupt & Exception Controller
Power Control
L1D Cache/SRAM

1.4.1 Central Processing Unit (CPU)

The C64x CPU, in Figure 1-1, contains:

- Program fetch unit
- Instruction dispatch unit, advanced instruction packing
- Instruction decode unit
- Two data paths, each with four functional units
- 64 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 32 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.

The 64x+ CPU, in Figure 1-2 , contains:

- Program fetch unit
- 16/32 bit instruction dispatch unit, advanced instruction packing
- Instruction decode unit
- Two data paths, each with four functional units
- 64 32-bit registers
- Control registers
- Control logic
- Test, emulation, and interrupt logic
- Internal DMA (IDMA) for transfers between internal memories

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 32 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 C64x and C64x+ DSP have a 32-bit, byte-addressable address space. Internal (on-chip) memory is organized in separate data and program spaces. When off-chip memory is used, these spaces are unified on most devices to a single memory space via the external memory interface (EMIF).

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

The 64x+ DSP has a 256-bit read-only port to access internal program memory and two 256-bit ports (read and write) to access internal data memory.

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 (EMIF) supports SDRAM, SBSRAM, SRAM, and other asynchronous memories for a broad range of external memory requirements and maximum system performance.
- The enhanced direct memory access (EDMA) controller transfers data between address ranges in the memory map without intervention by the CPU. The EDMA has 16 programmable channels, as well as a RAM space to hold multiple configurations for future transfers.
- The Ethernet Media Access Controller (EMAC) and Physical layer (PHY) device Management Data Input/Output (MDIO) module interfaces to the DSP through a custom interface that allows efficient data transmission and reception.
- The host port interface (HPI) is a parallel port through which a host processor can directly access the CPU memory space. The host device functions as a master to the interface, which increases ease of access. The host and CPU can exchange information via internal or external memory. The host also has direct access to memory-mapped peripherals. Connectivity to the CPU memory space is provided through the EDMA controller.
- The inter-integrated circuit (I2C) module provides an interface between a C64x/C64x+ DSP and I^{2}C compatible devices connected by way of the I^{2}C serial bus.
- The multichannel audio serial port (McASP) functions as a general-purpose audio serial port optimized for the needs of multichannel audio applications. The McASP is intended to be flexible so that it may connect gluelessly to audio analog-to-digital converters (ADC), digital-to-analog converters (DAC), codec, digital audio interface receiver (DIR) and S/PDIF transmit physical layer components.
- The multichannel buffered serial port (McBSP) 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 EDMA controller. It also has multichannel capability compatible with the T1, E1, SCSA, and MVIP networking standards.
- The peripheral component interconnect (PCI) port supports connection fo the C64x/C64x+ DSP to a PCI host via the integrated PCI master/slave bus interface.
- 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.
- Channel decoding of high bit-rate data channels found in third generation (3G) cellular standards requires decoding of turbo-encoded data. The turbo-decoder coprocessor (TCP) in the C6000 DSP is designed to perform this operation for IS2000 and 3GPP wireless standards.
- Channel decoding of voice and low bit-rate data channels found in third generation (3G) cellular standards requires decoding of convolutional encoded data. The Viterbi-decoder coprocessor (VCP) in the C6000 DSP is designed to perform this operation for IS2000 and 3GPP wireless standards.
- The universal test and operations PHY interface for asynchronous transfer mode [ATM] (UTOPIA) is an ATM controller (ATMC) slave device that interfaces to a master ATM controller. The UTOPIA port conforms to the ATM Forum standard specification af-phy-0039.000. Specifically, this interface supports the UTOPIA Level 2 interface that allows 8-bit slave operation up to 50 MHz for both transmit and receive operations.

For an overview of the peripherals available on the C6000 DSP, refer to the *TM320C6000 DSP Peripherals Overview Reference Guide* (SPRU190) or to your device-specific data manual.

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.

Topic **Page**

2.1 Introduction

The components of the data path for the 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

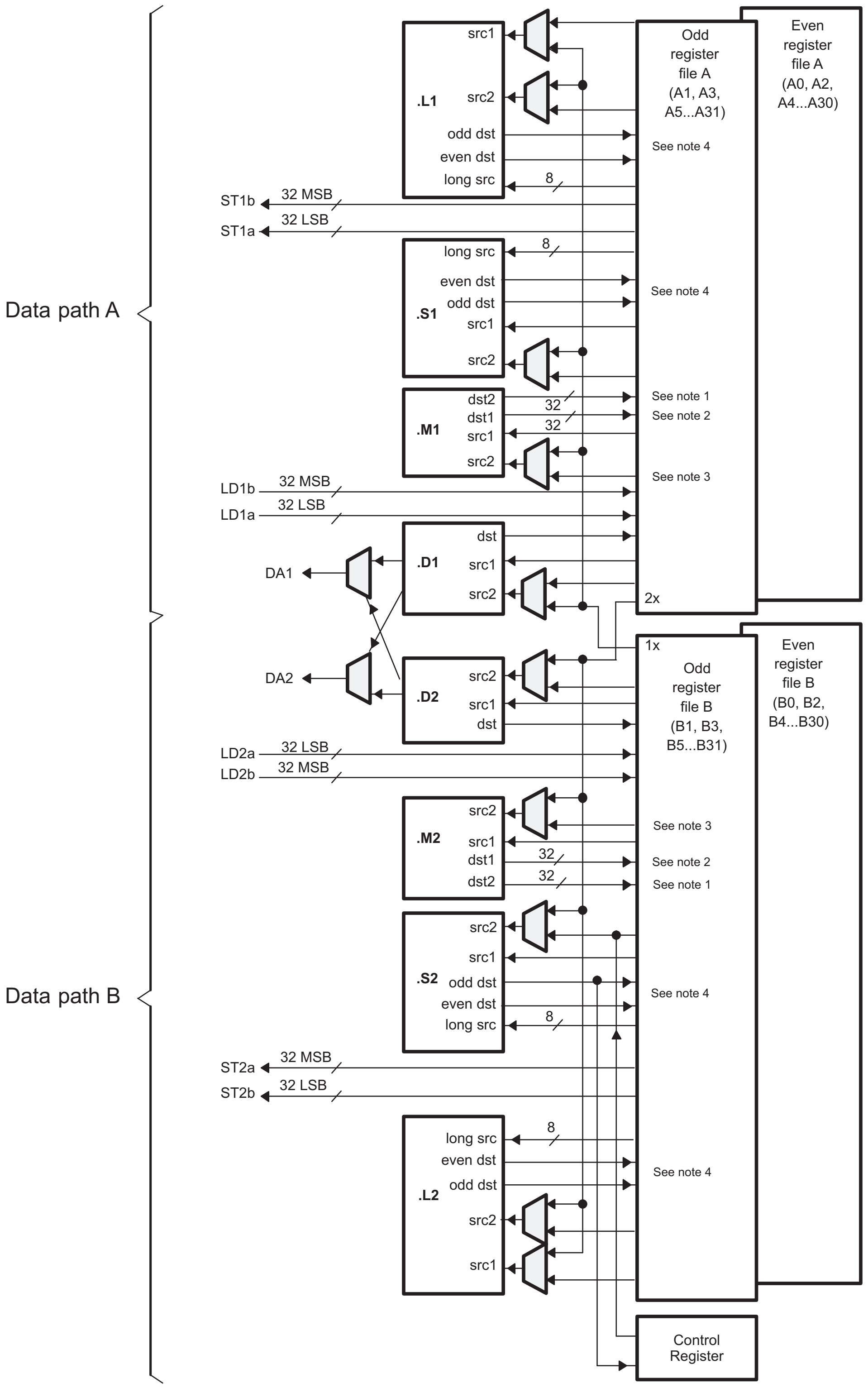
There are two general-purpose register files (A and B) in the CPU data paths. Each of these files contains 32 32-bit registers (A0–A31 for file A and B0–B31 for file B), as shown in Table 2-1. The general-purpose registers can be used for data, data address pointers, or condition registers.

The DSP general-purpose register files support data ranging in size from packed 8-bit through 64-bit fixed-point data. Values larger than 32 bits, such as 40-bit and 64-bit quantities, are stored in register pairs. 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 32 valid register pairs for 40-bit and 64-bit data in the DSP cores. In assembly language syntax, a colon between the register names denotes the register pair, and the odd-numbered register is specified first.

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. CPU Data Paths

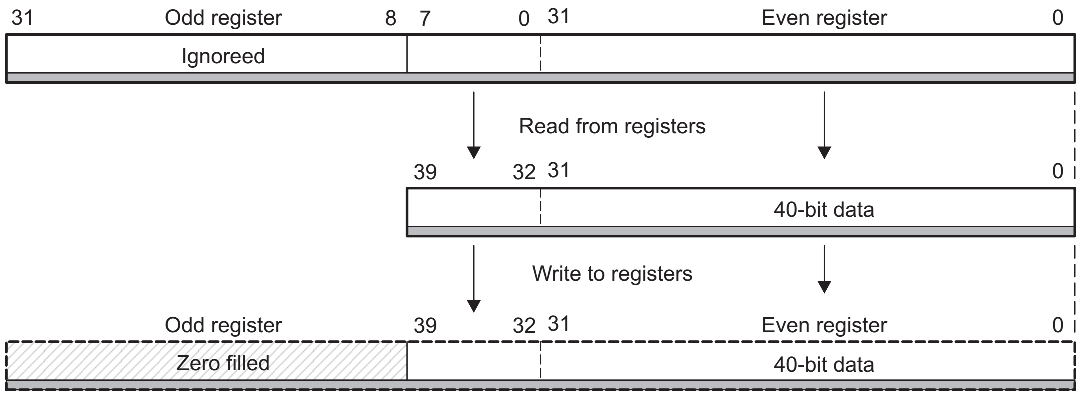


1. On .M unit, dst2 is 32 MSB.
2. On .M unit, dst1 is 32 MSB.
3. On C64x CPU .M unit, src2 is 32 bits; on C64x+ CPU .M unit, src2 is 64 bits.
4. On .L and .S units, odd dst connects to odd register files and even dst connects to even register files.

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

Register Files	
A	**B**
A1:A0	B1:B0
A3:A2	B3:B2
A5:A4	B5:B4
A7:A6	B7:B6
A9:A8	B9:B8
A11:A10	B11:B10
A13:A12	B13:B12
A15:A14	B15:B14
A17:A16	B17:B16
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.

In addition to performing all of the TMS320C62x DSP instructions, the C64x and C64x+ DSP also contains many 8-bit to 16-bit extensions to the instruction set. For example, the **MPYU4** instruction performs four 8 × 8 unsigned multiplies with a single instruction on an .M unit. The **ADD4** instruction performs four 8-bit additions with a single instruction on an .L unit.

Most data lines in the CPU support 32-bit operands, and some support long (40-bit) and doubleword (64-bit) operands. Each functional unit has its own 32-bit write port, so all eight units can be used in parallel every cycle, 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. Since each DSP multiplier can return up to a 64-bit result, an extra write port has been added from the multipliers to the register file.

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
.L unit (.L1, .L2)	32/40-bit arithmetic and compare operations
	32-bit logical operations
	Leftmost 1 or 0 counting for 32 bits
	Normalization count for 32 and 40 bits
	Byte shifts
	Data packing/unpacking
	5-bit constant generation
	Dual 16-bit arithmetic operations
	Quad 8-bit arithmetic operations
	Dual 16-bit minimum/maximum operations
	Quad 8-bit minimum/maximum operations
.S unit (.S1, .S2)	32-bit arithmetic operations
	32/40-bit shifts and 32-bit bit-field operations
	32-bit logical operations
	Branches
	Constant generation
	Register transfers to/from control register file (.S2 only)
	Byte shifts
	Data packing/unpacking
	Dual 16-bit compare operations
	Quad 8-bit compare operations
	Dual 16-bit shift operations
	Dual 16-bit saturated arithmetic operations
	Quad 8-bit saturated arithmetic operations

Table 2-2. Functional Units and Operations Performed (continued)

Functional Unit	Fixed-Point Operations
.M unit (.M1, .M2)	32 × 32-bit multiply operations
	16 × 16-bit multiply operations
	16 × 32-bit multiply operations
	Quad 8 × 8-bit multiply operations
	Dual 16 × 16-bit multiply operations
	Dual 16 × 16-bit multiply with add/subtract operations
	Quad 8 × 8-bit multiply with add operation
	Bit expansion
	Bit interleaving/de-interleaving
	Variable shift operations
	Rotation
	Galois Field Multiply
.D unit (.D1, .D2)	32-bit add, subtract, linear and circular address calculation
	Loads and stores with 5-bit constant offset
	Loads and stores with 15-bit constant offset (.D2 only)
	Load and store doublewords with 5-bit constant
	Load and store nonaligned words and doublewords
	5-bit constant generation
	32-bit logical operations

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

On the DSP, a delay clock cycle is introduced whenever an instruction attempts to read a register via a cross path that was updated in the previous cycle. This is known as a cross path stall. This stall is inserted automatically by the hardware, no **NOP** instruction is needed. It should be noted that no stall is introduced if the register being read is the destination for data placed by an **LDx** instruction. For more information see Section 3.7.4. Techniques for avoiding this stall are discussed in the *TMS320C6000 Programmers Guide* (SPRU198).

2.5 Memory, Load, and Store Paths

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

On the C6000 architecture, some of the ports for long and 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 3.7.6.

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 DSP, LD1 is comprised of LD1a and LD1b to support 64-bit loads; ST1 is comprised of ST1a and ST1b to support 64-bit stores. Similarly, T2 consists of the DA2 address path and the LD2 and ST2 data paths. For the DSP, LD2 is comprised of LD2a and LD2b to support 64-bit loads; ST2 is comprised of ST2a and ST2b to support 64-bit stores.

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 Galois Field

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

The DSP contains Galois field multiply hardware that is used for Reed-Solomon encode and decode functions. To understand the relevance of the Galois field multiply hardware, it is necessary to first define some mathematical terms.

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

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

An example is doing arithmetic using integers modulo 2. Perform the operations using normal integer arithmetic and then take the result modulo 2. Table 2-3 illustrates addition, subtraction, and multiplication modulo 2.

Table 2-3. Modulo 2 Arithmetic

Addition

+	0	1
0	0	1
1	1	0

Subtraction

-	0	1
0	0	1
1	1	0

Multiplication

×	0	1
0	0	0
1	0	1

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

$x \times x^{-1} = 1$

Another example, arithmetic modulo 5, illustrates this concept more clearly. The addition, subtraction, and multiplication tables are given in Table 2-4.

Table 2-4. Modulo 5 Arithmetic

Addition

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

Subtraction

-	0	1	2	3	4
0	0	4	3	2	1
1	1	0	4	3	2
2	2	1	0	4	3
3	3	2	1	0	4
4	4	3	2	1	0

Multiplication

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

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

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

A final example considers the field $GF(2^3)$, which has 8 elements. This can be generated by arithmetic modulo the (irreducible) polynomial $P(x) = x^3 + x + 1$. Elements of this field look like vectors of three bits. Table 2-5 shows the addition and multiplication tables for field $GF(2^3)$.

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

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

The Galois field multiply hardware on the DSP is named GMPY4. The **GMPY4** instruction performs four parallel operations on 8-bit packed data on the .M unit. The Galois field multiplier can be programmed to perform all Galois multiplies for fields of the form $GF(2^m)$, where m can range between 1 and 8 using any generator polynomial. The field size and the polynomial generator are controlled by the Galois field polynomial generator function register (GFPGFR).

In addition to the **GMPY4** instruction available on the C64x DSP, the C64x+ DSP has the **GMPY** instruction that uses either the GPLYA or GPLYB control register as a source for the polynomial (depending on whether the A or B side functional unit is used) and produces a 32-bit result.

The GFPGFR, shown in Figure 2-6 and described in Table 2-10, contains the Galois field polynomial generator and the field size control bits. These bits control the operation of the **GMPY4** instruction. GFPGFR can only be set via the **MVC** instruction. The default function after reset for the **GMPY4** instruction is field size = 7h and polynomial = 1Dh.

2.7.1 Special Timing Considerations

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

Table 2-5. Modulo Arithmetic for Field GF(2^3)

Addition

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

Multiplication

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

2.8 Control Register File

Table 2-6 lists the control registers contained in the control register file.

Table 2-6. Control Registers

Acronym	Register Name	Section
AMR	Addressing mode register	Section 2.8.3
CSR	Control status register	Section 2.8.4
GFPGFR	Galois field multiply control register	Section 2.8.5
ICR	Interrupt clear register	Section 2.8.6
IER	Interrupt enable register	Section 2.8.7
IFR	Interrupt flag register	Section 2.8.8
IRP	Interrupt return pointer register	Section 2.8.9
ISR	Interrupt set register	Section 2.8.10
ISTP	Interrupt service table pointer register	Section 2.8.11
NRP	Nonmaskable interrupt return pointer register	Section 2.8.12
PCE1	Program counter, E1 phase	Section 2.8.13
Control Register File Extensions (C64x+ DSP)		
DIER	Debug interrupt enable register	Section 2.9.1
DNUM	DSP core number register	Section 2.9.2
ECR	Exception clear register	Section 2.9.3
EFR	Exception flag register	Section 2.9.4
GPLYA	GMPY A-side polynomial register	Section 2.9.5
GPLYB	GMPY B-side polynomial register	Section 2.9.6
IERR	Internal exception report register	Section 2.9.7
ILC	Inner loop count register	Section 2.9.8
ITSR	Interrupt task state register	Section 2.9.9
NTSR	NMI/Exception task state register	Section 2.9.10
REP	Restricted entry point address register	Section 2.9.11
RILC	Reload inner loop count register	Section 2.9.12
SSR	Saturation status register	Section 2.9.13
TSCH	Time-stamp counter (high 32) register	Section 2.9.14
TSCL	Time-stamp counter (low 32) register	Section 2.9.14
TSR	Task state register	Section 2.9.15

2.8.1 Register Addresses for Accessing the Control Registers

Table 3-22 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 (see MVC) 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).

On the C64x+ CPU, access to some of the registers is restricted when in User mode. See Chapter 8 for more information.

2.8.2 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 2.8.10, Section 2.8.6, and Section 2.8.8 .

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 bits, 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.

2.8.3 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 2-7.

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 2-7. 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 2-8 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 2-8 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
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

Table 2-7. Addressing Mode Register (AMR) Field Descriptions (continued)

Bit	Field	Value	Description
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
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 2-8. 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

2.8.4 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 2-9. For the PWRD, EN, PCC, and DCC fields, see the device-specific datasheet to see if it supports the options that these fields control. The PCC and DCC fields are ignored on the C64x+ CPU.

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.

The PWRD, PCC, DCC, and PGIE fields cannot be written in User mode. The PCC and DCC fields can only be modified in Supervisor mode. See Chapter 8 for more information.

Figure 2-4. Control Status Register (CSR)

31 – 24	23 – 16
CPU ID	REVISION ID
R-x[1]	R-x[1]

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

LEGEND: R = Readable by the **MVC** instruction; W = Writeable by the **MVC** instruction; SW = Writeable by the **MVC** instruction only in supervisor mode; WC = Bit is cleared on write; -n = value after reset; -x = value is indeterminate after reset

(1) See the device-specific datasheet 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/SW-0	R/SW-0	R/SW-0	R/SW-0	R/SW-0	R/SW-0

LEGEND: R = Readable by the **MVC** instruction; W = Writeable by the **MVC** instruction; *-n* = value after reset; SW = Writeable by the **MVC** instruction only in supervisor mode; -n = value after reset

Table 2-9. 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-Bh	Reserved
		Ch	C64x CPU
		Dh-Fh	Reserved
		10h	C64x+ CPU
		11h-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 datasheet. Not writable by the **MVC** instruction.

Table 2-9. Control Status Register (CSR) Field Descriptions (continued)

Bit	Field	Value	Description
15-10	PWRD	0-3Fh	Power-down mode field. See Figure 2-5. Writable by the **MVC** instruction only in Supervisor mode.
		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	No functional units generated saturated results.
		1	One or more functional units performed an arithmetic operation which resulted in saturation.
8	EN		Endian mode. Not writable by the **MVC** instruction.
		0	Big endian
		1	Little endian
7-5	PCC	0-7h	Program cache control mode. This field is ignored on the C64x+ CPU. Writable by the **MVC** instruction only in Supervisor mode; not writable in User mode. See the *TMS320C64x DSP Two-Level Internal Memory Reference Guide* (SPRU610).
		0	Direct-mapped cache enabled
		1h	Reserved
		2h	Direct-mapped cache enabled
		3h-7h	Reserved
4-2	DCC	0-7h	Data cache control mode. This field is ignored on the C64x+ CPU. Writable by the **MVC** instruction only in Supervisor mode; not writable in User mode. See the *TMS320C64x DSP Two-Level Internal Memory Reference Guide* (SPRU610).
		0	2-way cache enabled
		1h	Reserved
		2h	2-way cache enabled
		3h-7h	Reserved
1	PGIE		Previous GIE (global interrupt enable). This bit contains a copy of the GIE bit at the point when interrupt is taken. It is physically the same bit as GIE bit in the interrupt task state register (ITSR). Writeable by the **MVC** instruction only in Supervisor mode; not writable in User mode.
		0	Interrupts will be disabled after return from interrupt.
		1	Interrupts will be enabled after return from interrupt.
0	GIE		Global interrupt enable. Physically the same bit as GIE bit in the task state register (TSR). Writable by the **MVC** instruction in Supervisor and User mode. See Section 5.2 for details on how the GIE bit affects interruptibility.
		0	Disables all interrupts, except the reset interrupt and NMI (nonmaskable interrupt).
		1	Enables all interrupts.

2.8.5 Galois Field Polynomial Generator Function Register (GFPGFR)

The Galois field polynomial generator function register (GFPGFR) controls the field size and the Galois field polynomial generator of the Galois field multiply hardware. The GFPGFR is shown in Figure 2-6 and described in Table 2-10. The Galois field is described in Section 2.7.

Figure 2-6. Galois Field Polynomial Generator Function Register (GFPGFR)

31–27	26–24	23–16
Reserved	SIZE	Reserved
R-0	R/W-7h	R-0

15–8	7–0
Reserved	POLY
R-0	R/W-1Dh

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

Table 2-10. Galois Field Polynomial Generator Function Register (GFPGFR) 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.
26-24	SIZE	0-7h	Field size.
23-8	Reserved	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.
7-0	POLY	0-FFh	Polynomial generator.

2.8.6 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 2-7 and described in Table 2-11. See Chapter 5 for more information on interrupts.

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 2-7. 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	W-0	W-0	W-0	W-0	W-0	W-0	W-0	W-0	W-0	W-0	W-0	R-0

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

Table 2-11. 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.

2.8.7 Interrupt Enable Register (IER)

The interrupt enable register (IER) enables and disables individual interrupts. The IER is shown in Figure 2-8 and described in Table 2-12.

The IER is not accessible in User mode. See Section 8.2.4.1 for more information. See Chapter 5 for more information on interrupts.

Figure 2-8. 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/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	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 2-12. 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.

2.8.8 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 (see MVC) for information on how to use this instruction.) The IFR is shown in Figure 2-9 and described in Table 2-13. See Chapter 5 for more information on interrupts.

Figure 2-9. 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	R-0	R-0	R-0	R-0	R-0	R-0	R-0	R-0	R-0	R-0	R-0

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

Table 2-13. 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.

2.8.9 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 2-10.

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.

See Chapter 5 for more information on interrupts.

Figure 2-10. 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

2.8.10 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 2-11 and described in Table 2-14. See Chapter 5 for more information on interrupts.

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 2-11. 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	W-0	W-0	W-0	W-0	W-0	W-0	W-0	W-0	W-0	W-0	W-0	R-0

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

Table 2-14. 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.

2.8.11 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 2-12 and described in Table 2-15. See Section 5.1.2.2 for a discussion of the use of the ISTP.

The ISTP is not accessible in User mode. See Section 8.2.4.1 for more information. See Chapter 5 for more information on interrupts.

Figure 2-12. Interrupt Service Table Pointer Register (ISTP)

31–16
ISTB
R/W-S

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

LEGEND: R = Readable by the **MVC** instruction; W = Writeable by the **MVC** instruction; -*n* = value after reset; S = See the device-specific data manual for the default value of this field after reset

Table 2-15. 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 a device-specific default value on reset; therefore, upon startup the IST must reside at this specific address. See the device-specific data manual for more information. 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 its default value. 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) 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	0	Cleared to 0 (fetch packets must be aligned on 8-word (32-byte) boundaries).

2.8.12 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 2-13.

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.

See Chapter 5 for more information on interrupts. See Chapter 6 for more information on exceptions.

Figure 2-13. 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

2.8.13 E1 Phase Program Counter (PCE1)

The E1 phase program counter (PCE1), shown in Figure 2-14, contains the 32-bit address of the fetch packet in the E1 pipeline phase.

Figure 2-14. E1 Phase Program Counter (PCE1)

31 … 0
PCE1
R-x

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

2.9 Control Register File Extensions

Table 2-16 lists the additional control registers in the C64x+ DSP.

Table 2-16. Control Register File Extensions (C64x+ DSP)

Acronym	Register Name	Section
DIER	Debug interrupt enable register	Section 2.9.1
DNUM	DSP core number register	Section 2.9.2
ECR	Exception clear register	Section 2.9.3
EFR	Exception flag register	Section 2.9.4
GPLYA	GMPY polynomial for A side register	Section 2.9.5
GPLYB	GMPY polynomial for B side register	Section 2.9.6
IERR	Internal exception report register	Section 2.9.7
ILC	Inner loop count register	Section 2.9.8
ITSR	Interrupt task state register	Section 2.9.9
NTSR	NMI/Exception task state register	Section 2.9.10
REP	Restricted entry point register	Section 2.9.11
RILC	Reload inner loop count register	Section 2.9.12
SSR	Saturation status register	Section 2.9.13
TSCH	Time stamp counter register—high half of 64 bit	Section 2.9.14
TSCL	Time stamp counter register—low half of 64 bit	Section 2.9.14
TSR	Task state register	Section 2.9.15

2.9.1 Debug Interrupt Enable Register (DIER)

The debug interrupt enable register (DIER) is used to designate which interrupts and exceptions are treated as high-priority interrupts when operating in real-time emulation mode. The DIER is shown in Figure 2-15 and described in Table 2-17.

Figure 2-15. Debug Interrupt Enable Register (DIER)

31	30	29	16
NMI	EXCEP	Reserved	
R/W-0	R/W-0	R-0	

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
INT15	INT14	INT13	INT12	INT11	INT10	INT9	INT8	INT7	INT6	INT5	INT4	Reserved		WSEL	Rsvd
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	R/W-0	R-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 2-17. Debug Interrupt Enable Register (DIER) Field Descriptions

Bit	Field	Value	Description
31	NMI		Nonmaskable interrupt (NMI).
		1	Designate NMI as high-priority interrupt.
30	EXCEP		Maskable external exception (EXCEP).
		1	Designate EXCEP as high-priority interrupt.
29-16	Reserved	0	Reserved
15	INT15		Maskable interrupt 15 (INT15).
		1	Designate INT15 as high-priority interrupt.
14	INT14		Maskable interrupt 14 (INT14).
		1	Designate INT14 as high-priority interrupt.
13	INT13		Maskable interrupt 13 (INT13).
		1	Designate INT13 as high-priority interrupt.
12	INT12		Maskable interrupt 12 (INT12).
		1	Designate INT12 as high-priority interrupt.
11	INT11		Maskable interrupt 11 (INT11).
		1	Designate INT11 as high-priority interrupt.
10	INT10		Maskable interrupt 10 (INT10).
		1	Designate INT10 as high-priority interrupt.
9	INT9		Maskable interrupt 9 (INT9).
		1	Designate INT9 as high-priority interrupt.
8	INT8		Maskable interrupt 8 (INT8).
		1	Designate INT8 as high-priority interrupt.
7	INT7		Maskable interrupt 7 (INT7).
		1	Designate INT7 as high-priority interrupt.
6	INT6		Maskable interrupt 6 (INT6).
		1	Designate INT6 as high-priority interrupt.
5	INT5		Maskable interrupt 5 (INT5).
		1	Designate INT5 as high-priority interrupt.
4	INT4		Maskable interrupt 4 (INT4).
		1	Designate INT4 as high-priority interrupt.
3-2	Reserved	0	Reserved
1	WSEL		Write control select. This bit must be cleared to 0 to modify bits 31-2.
		0	Bits 31-2 can be modified.
0	Reserved	0	Reserved

2.9.2 DSP Core Number Register (DNUM)

Multiple C64x+ CPUs may be used in a system. The DSP core number register (DNUM), provides an identifier to shared resources in the system which identifies which C64x+ CPU is accessing those resources. The contents of this register are set to a specific value (depending on the device) at reset. See your device-specific data manual for the reset value of this register. The DNUM is shown in Figure 2-16.

Figure 2-16. DSP Core Number Register (DNUM)

31–16
Reserved
R-0

15–8	7–0
Reserved	DSP number
R-0	R-S

LEGEND: R = Readable by the **MVC** instruction; -*n* = value after reset; S = See the device-specific data manual for the default value of this field after reset

2.9.3 Exception Clear Register (ECR)

The exception clear register (ECR) is used to clear individual bits in the exception flag register (EFR). Writing a 1 to any bit in ECR clears the corresponding bit in EFR.

The ECR is not accessible in User mode. See Section 8.2.4.1 for more information. See Chapter 6 for more information on exceptions.

2.9.4 Exception Flag Register (EFR)

The exception flag register (EFR) contains bits that indicate which exceptions have been detected. Clearing the EFR bits is done by writing a 1 to the corresponding bit position in the exception clear register (ECR). Writing a 0 to the bits in this register has no effect. The EFR is shown in Figure 2-17 and described in Table 2-18.

The EFR is not accessible in User mode. See Section 8.2.4.1 for more information. See Chapter 6 for more information on exceptions.

Figure 2-17. Exception Flag Register (EFR)

31	30	29–16
NXF	EXF	Reserved
R/W-0	R/W-0	R-0

15–2	1	0
Reserved	IXF	SXF
R-0	R/W-0	R/W-0

LEGEND: R = Readable by the **MVC** EFR instruction only in Supervisor mode; W = Clearable by the **MVC** ECR instruction only in Supervisor mode; -*n* = value after reset

Table 2-18. Exception Flag Register (EFR) Field Descriptions

Bit	Field	Value	Description
31	NXF		NMI exception flag.
		0	NMI exception has not been detected.
		1	NMI exception has been detected.
30	EXF		EXCEP flag.
		0	Exception has not been detected.
		1	Exception has been detected.
29-2	Reserved	0	Reserved. Read as 0.
1	IXF		Internal exception flag.
		0	Internal exception has not been detected.
		1	Internal exception has been detected.
0	SXF		Software exception flag (set by **SWE** or **SWENR** instructions).
		0	Software exception has not been detected.
		1	Software exception has been detected.

2.9.5 GMPY Polynomial—A Side Register (GPLYA)

The **GMPY** instruction (see GMPY) uses the 32-bit polynomial in the GMPY polynomial—A side register (GPLYA), Figure 2-18, when the instruction is executed on the M1 unit.

Figure 2-18. GMPY Polynomial A-Side Register (GPLYA)

31 … 0
32-bit polynomial
R/W-0

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

2.9.6 GMPY Polynomial—B Side Register (GPLYB)

The **GMPY** instruction (see GMPY) uses the 32-bit polynomial in the GMPY polynomial—B side register (GPLYB), Figure 2-19, when the instruction is executed on the M2 unit.

Figure 2-19. GMPY Polynomial B-Side (GPLYB)

31 … 0
32-bit polynomial
R/W-0

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

2.9.7 Internal Exception Report Register (IERR)

The internal exception report register (IERR) contains flags that indicate the cause of the internal exception. In the case of simultaneous internal exceptions, the same flag may be set by different exception sources. In this case, it may not be possible to determine the exact causes of the individual exceptions. The IERR is shown in Figure 2-20 and described in Table 2-19.

The IERR is not accessible in User mode. See Section 8.2.4.1 for more information. See Chapter 6 for more information on exceptions.

Figure 2-20. Internal Exception Report Register (IERR)

31–16
Reserved
R-0

15–9	8	7	6	5	4	3	2	1	0
Reserved	MSX	LBX	PRX	RAX	RCX	OPX	EPX	FPX	IFX
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

LEGEND: R = Readable by the **MVC** instruction only in Supervisor mode; W = Writeable by the **MVC** instruction only in Supervisor mode; -*n* = value after reset

Table 2-19. Internal Exception Report Register (IERR) Field Descriptions

Bit	Field	Value	Description
31-9	Reserved	0	Reserved. Read as 0.
8	MSX		Missed stall exception
		0	Missed stall exception is not the cause.
		1	Missed stall exception is the cause.
7	LBX		SPLOOP buffer exception
		0	SPLOOP buffer exception is not the cause.
		1	SPLOOP buffer exception is the cause.
6	PRX		Privilege exception
		0	Privilege exception is not the cause.
		1	Privilege exception is the cause.
5	RAX		Resource access exception
		0	Resource access exception is not the cause.
		1	Resource access exception is the cause.
4	RCX		Resource conflict exception
		0	Resource conflict exception is not the cause.
		1	Resource conflict exception is the cause.
3	OPX		Opcode exception
		0	Opcode exception is not the cause.
		1	Opcode exception is the cause.
2	EPX		Execute packet exception
		0	Execute packet exception is not the cause.
		1	Execute packet exception is the cause.
1	FPX		Fetch packet exception
		0	Fetch packet exception is not the cause.
		1	Fetch packer exception is the cause.
0	IFX		Instruction fetch exception
		0	Instruction fetch exception is not the cause.
		1	Instruction fetch exception is the cause.

2.9.8 SPLOOP Inner Loop Count Register (ILC)

The **SPLOOP** or **SPLOOPD** instructions use the SPLOOP inner loop count register (ILC), Figure 2-21, as the count of the number of iterations left to perform. The ILC content is decremented at each stage boundary until the ILC content reaches 0.

Figure 2-21. Inner Loop Count Register (ILC)

31 … 0
32-bit inner loop count
R/W-0

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

2.9.9 Interrupt Task State Register (ITSR)

The interrupt task state register (ITSR) is used to store the contents of the task state register (TSR) in the event of an interrupt. The ITSR is shown in Figure 2-22 and described in Table 2-20. For detailed bit descriptions, see Section 2.9.15.

The GIE bit in ITSR is physically the same bit as the PGIE bit in CSR.

The ITSR is not accessible in User mode. See Section 8.2.4.1 for more information.

Figure 2-22. Interrupt Task State Register (ITSR)

31 … 16
Reserved
R-0

15	14	13–11	10	9	8	7–6	5	4	3	2	1	0
IB	SPLX	Reserved	EXC	INT	Rsvd	CXM	Rsvd	DBGM	XEN	GEE	SGIE	GIE
R/W-0	R/W-0	R-0	R/W-0	R/W-0	R-0	R/W-0	R-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0

LEGEND: R = Readable by the **MVC** instruction only in Supervisor mode; W = Writeable by the **MVC** instruction only in Supervisor mode; -*n* = value after reset

Table 2-20. Interrupt Task State Register (ITSR) Field Descriptions

Bit	Field	Description
31-16	Reserved	Reserved. Read as 0.
15	IB	Interrupt occurred while interrupts were blocked.
14	SPLX	Interrupt occurred during an SPLOOP.
13-11	Reserved	Reserved. Read as 0.
10	EXC	Contains EXC bit value in TSR at point of interrupt.
9	INT	Contains INT bit value in TSR at point of interrupt.
8	Reserved	Reserved. Read as 0.
7-6	CXM	Contains CXM bit value in TSR at point of interrupt.
5	Reserved	Reserved. Read as 0.
4	DBGM	Contains DBGM bit value in TSR at point of interrupt.
3	XEN	Contains XEN bit value in TSR at point of interrupt.
2	GEE	Contains GEE bit value in TSR at point of interrupt.
1	SGIE	Contains SGIE bit value in TSR at point of interrupt.
0	GIE	Contains GIE bit value in TSR at point of interrupt.

2.9.10 NMI/Exception Task State Register (NTSR)

The NMI/exception task state register (NTSR) is used to store the contents of the task state register (TSR) and the conditions under which an exception occurred in the event of a nonmaskable interrupt (NMI) or an exception. The NTSR is shown in Figure 2-23 and described in Table 2-21. For detailed bit descriptions (except for the HWE bit), see Section 2.9.15. The HWE bit is set by taking a hardware exception (NMI, EXCEP, or internal) and is cleared by either **SWE** or **SWENR** instructions.

The NTSR is not accessible in User mode. See Section 8.2.4.1 for more information.

Figure 2-23. NMI/Exception Task State Register (NTSR)

31–17	16
Reserved	HWE
R-0	R/W-0

15	14	13–11	10	9	8	7–6	5	4	3	2	1	0
IB	SPLX	Reserved	EXC	INT	Rsvd	CXM	Rsvd	DBGM	XEN	GEE	SGIE	GIE
R/W-0	R/W-0	R-0	R/W-0	R/W-0	R-0	R/W-0	R-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0

LEGEND: R = Readable by the **MVC** instruction only in Supervisor mode; W = Writeable by the **MVC** instruction only in Supervisor mode; -*n* = value after reset

Table 2-21. NMI/Exception Task State Register (NTSR) Field Descriptions

Bit	Field	Description
31-17	Reserved	Reserved. Read as 0.
16	HWE	Hardware exception taken (NMI, EXCEP, or internal).
15	IB	Exception occurred while interrupts were blocked.
14	SPLX	Exception occurred during an SPLOOP.
13-11	Reserved	Reserved. Read as 0.
10	EXC	Contains EXC bit value in TSR at point exception taken.
9	INT	Contains INT bit value in TSR at point exception taken.
8	Reserved	Reserved. Read as 0.
7-6	CXM	Contains CXM bit value in TSR at point exception taken.
5	Reserved	Reserved. Read as 0.
4	DBGM	Contains DBGM bit value in TSR at point exception taken.
3	XEN	Contains XEN bit value in TSR at point exception taken.
2	GEE	Contains GEE bit value in TSR at point exception taken.
1	SGIE	Contains SGIE bit value in TSR at point exception taken.
0	GIE	Contains GIE bit value in TSR at point exception taken.

2.9.11 Restricted Entry Point Register (REP)

The restricted entry point register (REP) is used by the **SWENR** instruction as the target of the change of control when an **SWENR** instruction is issued. The contents of REP should be preinitialized by the processor in Supervisor mode before any **SWENR** instruction is issued. See Section 8.2.4.1 for more information. REP cannot be modified in User mode.

2.9.12 SPLOOP Reload Inner Loop Count Register (RILC)

Predicated **SPLOOP** or **SPLOOPD** instructions used in conjunction with a **SPMASKR** or **SPKERNELR** instruction use the SPLOOP reload inner loop count register (RILC), Figure 2-24, as the iteration count value to be written to the SPLOOP inner loop count register (ILC) in the cycle before the reload operation begins. See Chapter 7 for more information.

Figure 2-24. Reload Inner Loop Count Register (RILC)

31 … 0
32-bit inner loop count reload
R/W-0

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

2.9.13 Saturation Status Register (SSR)

The saturation status register (SSR) provides saturation flags for each functional unit, making it possible for the program to distinguish between saturations caused by different instructions in the same execute packet. There is no direct connection to the SAT bit in the control status register (CSR); writes to the SAT bit have no effect on SSR and writes to SSR have no effect on the SAT bit. Care must be taken when restoring SSR and the SAT bit when returning from a context switch. Since the SAT bit cannot be written to a value of 1 using the **MVC** instruction, restoring the SAT bit to a 1 must be done by executing an instruction that results in saturation. The saturating instruction would affect SSR; therefore, SSR must be restored after the SAT bit has been restored. The SSR is shown in Figure 2-25 and described in Table 2-22.

Instructions resulting in saturation set the appropriate unit flag in SSR in the cycle following the writing of the result to the register file. The setting of the flag from a functional unit takes precedence over a write to the bit from an **MVC** instruction. If no functional unit saturation has occurred, the flags may be set to 0 or 1 by the **MVC** instruction, unlike the SAT bit in CSR.

The bits in SSR can be set by the **MVC** instruction or by a saturation in the associated functional unit. The bits are cleared only by a reset or by the **MVC** instruction. The bits are not cleared by the occurrence of a nonsaturating instruction.

Figure 2-25. Saturation Status Register (SSR)

31 … 16
Reserved
R-0

15 … 6	5	4	3	2	1	0
Reserved	M2	M1	S2	S1	L2	L1
R-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 2-22. Saturation Status Register Field Descriptions

Bit	Field	Value	Description
31-6	Reserved	0	Reserved. Read as 0.
5	M2		M2 unit.
		0	Saturation did not occur on M2 unit.
		1	Saturation occurred on M2 unit.
4	M1		M1 unit.
		0	Saturation did not occur on M1 unit.
		1	Saturation occurred on M1 unit.

Table 2-22. Saturation Status Register Field Descriptions (continued)

Bit	Field	Value	Description
3	S2		S2 unit.
		0	Saturation did not occur on S2 unit.
		1	Saturation occurred on S2 unit.
2	S1		S1 unit.
		0	Saturation did not occur on S1 unit.
		1	Saturation occurred on S1 unit.
1	L2		L2 unit.
		0	Saturation did not occur on L2 unit.
		1	Saturation occurred on L2 unit.
0	L1		L1 unit.
		0	Saturation did not occur on L1 unit.
		1	Saturation occurred on L1 unit.

2.9.14 Time Stamp Counter Registers (TSCL and TSCH)

The C64x+ CPU contains a free running 64-bit counter that advances each CPU clock under normal operation. The counter is accessed as two 32-bit read-only control registers, TSCL (Figure 2-26) and TSCH (Figure 2-27).

Figure 2-26. Time Stamp Counter Register - Low Half (TSCL)

31 … 0
CPU clock count (32 LSBs of 64-bit value)
R-0

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

Figure 2-27. Time Stamp Counter Register - High Half (TSCH)

31 … 0
CPU clock count (32 MSBs of 64-bit value)
R-0

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

2.9.14.1 Initialization

The counter is cleared to 0 after reset, and counting is disabled.

2.9.14.2 Enabling Counting

The counter is enabled by writing to TSCL. The value written is ignored. Counting begins in the cycle after the **MVC** instruction executes. If executed with the count disabled, the following code sequence shows the timing of the count starting (assuming no stalls occur in the three cycles shown).

```
MVC B0,TSCL ; Start TSC
MVC TSCL,B0 ; B0 = 0
MVC TSCL,B1 ; B1 = 1
```

2.9.14.3 Disabling Counting

Once enabled, counting cannot be disabled under program control. Counting is disabled in the following cases:

- After exiting the reset state.
- When the CPU is fully powered down.

2.9.14.4 Reading the Counter

Reading the full 64-bit count takes two sequential **MVC** instructions. A read from TSCL causes the upper 32 bits of the count to be copied into TSCH. In normal operation, only this snapshot of the upper half of the 64-bit count is available to the programmer. The value read will always be the value copied at the cycle of the last MVC TSCL, *reg* instruction. If it is read with no TSCL reads having taken place since reset, then the reset value of 0 is read.

CAUTION

Reading TSCL in the cycle before a cross path stall may give an inaccurate value in TSCH.

When reading the full 64-bit value, it must be ensured that no interrupts are serviced between the two **MVC** instructions if an ISR is allowed to make use of the time stamp counter. There is no way for an ISR to restore the previous value of TSCH (snapshot) if it reads TSCL, since a new snapshot is performed.

Two methods for reading the 64-bit count value in an uninterruptible manner are shown in Example 2-1 and Example 2-2. Example 2-1 uses the fact that interrupts are automatically disabled in the delay slots of a branch to prevent an interrupt from happening between the TSCL read and the TSCH read. Example 2-2 accomplishes the same task by explicitly disabling interrupts.

Example 2-1. Code to Read the 64-Bit TSC Value in Branch Delay Slot

```
        BNOP       TSC_Read_Done, 3
        MVC        TSCL,B0         ; Read the low half first; high half copied to TSCH
        MVC        TSCH,B1         ; Read the snapshot of the high half
TSC_Read_Done:
```

Example 2-2. Code to Read the 64-Bit TSC Value Using DINT/RINT

```
      DINT
||       MVC          TSCL,B0         ; Read the low half first; high half copied to TSCH
      RINT
||       MVC          TSCH,B1         ; Read the snapshot of the high half
TSC_Read_Done:
```

2.9.15 Task State Register (TSR)

The task state register (TSR) contains all of the status bits that determine or indicate the current execution environment. TSR is saved in the event of an interrupt or exception to the ITSR or NTSR, respectively. All bits are readable by the **MVC** instruction. The TSR is shown in Figure 2-28 and described in Table 2-23. The SGIE bit in TSR is used by the **DINT** and **RINT** instructions to globally disable and reenable interrupts.

The GIE and SGIE bits may be written in both User mode and Supervisor mode. The remaining bits all have restrictions on how they are written. See Section 8.2.4.2 for more information.

The GIE bit in TSR is physically the same bit as the GIE bit in CSR. It is retained in CSR for compatibility reasons, but placed in TSR so that it will be copied in the event of either an exception or an interrupt.

Figure 2-28. Task State Register (TSR)

31–16
Reserved
R-0

15	14	13–11	10	9	8	7–6	5	4	3	2	1	0
IB	SPLX	Reserved	EXC	INT	Rsvd	CXM	Rsvd	DBGM	XEN	GEE	SGIE	GIE
R-0	R-0	R-0	R/C-0	R-0	R-0	R/W-0	R-0	R/W-0	R/W-0	R/S-0	R/W-0	R/W-0

LEGEND: R = Readable by the **MVC** instruction; W = Writeable in Supervisor mode; C = Clearable in Supervisor mode; S = Can be set in Supervisor mode; -*n* = value after reset

Table 2-23. Task State Register (TSR) Field Descriptions

Bit	Field	Value	Description
31-16	Reserved	0	Reserved. Read as 0.
15	IB		Interrupts blocked. Not writable by the **MVC** instruction; set only by hardware.
		0	Interrupts not blocked in previous cycle (interruptible point).
		1	Interrupts were blocked in previous cycle.
14	SPLX		SPLOOP executing. Not writable by the **MVC** instruction; set only by hardware.
		0	Not currently executing SPLOOP
		1	Currently executing SPLOOP
13-11	Reserved	0	Reserved. Read as 0.
10	EXC		Exception processing. Clearable by the **MVC** instruction in Supervisor mode. Not clearable by the **MVC** instruction in User mode.
		0	Not currently processing an exception.
		1	Currently processing an exception.
9	INT		Interrupt processing. Not writable by the **MVC** instruction.
		0	Not currently processing an interrupt.
		1	Currently processing an interrupt.
8	Reserved	0	Reserved. Read as 0.
7-6	CXM	0-3h	Current execution mode. Not writable by the **MVC** instruction; these bits reflect the current execution mode of the execute pipeline. CXM is set to 1 when you begin executing the first instruction in User mode. See Chapter 8 for more information.
		0	Supervisor mode
		1h	User mode
		2h-3h	Reserved (an attempt to set these values is ignored)
5	Reserved	0	Reserved. Read as 0.
4	DBGM		Emulator debug mask. Writable in Supervisor and User mode. Writable by emulator.
		0	Enables emulator capabilities.
		1	Disables emulator capabilities.

Table 2-23. Task State Register (TSR) Field Descriptions (continued)

Bit	Field	Value	Description
3	XEN		Maskable exception enable. Writable only in Supervisor mode.
		0	Disables all maskable exceptions.
		1	Enables all maskable exceptions.
2	GEE		Global exception enable. Can be set to 1 only in Supervisor mode. Once set, cannot be cleared except by reset.
		0	Disables all exceptions except the reset interrupt.
		1	Enables all exceptions.
1	SGIE		Saved global interrupt enable. Contains previous state of GIE bit after execution of a **DINT** instruction. Writable in Supervisor and User mode.
		0	Global interrupts remain disabled by the **RINT** instruction.
		1	Global interrupts are enabled by the **RINT** instruction.
0	GIE		Global interrupt enable. Same physical bit as the GIE bit in the control status register (CSR). Writable in Supervisor and User mode. See Section 5.2 for details on how the GIE bit affects interruptibility.
		0	Disables all interrupts except the reset interrupt and NMI (nonmaskable interrupt).
		1	Enables all interrupts.

Instruction Set

This chapter describes the assembly language instructions of the TMS320C64x DSP and TMS320C64x+ DSP . Also described are parallel operations, conditional operations, resource constraints, and addressing modes.

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

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 3.5
*cst*n	n-bit constant field (for example, cst5)
dint	64-bit integer value (two registers)
dst_e	lsb32 of 64-bit *dst* (placed in even-numbered register of a 64-bit register pair)
dst_h	msb8 of 40-bit *dst* (placed in odd-numbered register of 64-bit register pair)
dst_l	lsb32 of 40-bit *dst* (placed in even-numbered register of a 64-bit register pair)
dst_o	msb32 of 64-bit *dst* (placed in odd-numbered register of 64-bit register pair)
dws4	Four packed signed 16-bit integers in a 64-bit register pair
dwu4	Four packed unsigned 16-bit integers in a 64-bit register pair
gmpy	Galois Field Multiply
i2	Two packed 16-bit integers in a single 32-bit register
i4	Four packed 8-bit integers in a single 32-bit register
int	32-bit integer value
lmb0(x)	Leftmost 0 bit search of x
lmb1(x)	Leftmost 1 bit search of x
long	40-bit integer value
lsb*n* or LSB*n*	n least-significant bits (for example, lsb16)
msb*n* or MSB*n*	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
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)

Table 3-1. Instruction Operation and Execution Notations (continued)

Symbol	Meaning
scst*n*	n-bit signed constant field
se	Sign-extend
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
src1_e or *src2_e*	lsb32 of 64-bit *src* (placed in even-numbered register of a 64-bit register pair)
src1_h or *src2_h*	msb8 of 40-bit *src* (placed in odd-numbered register of 64-bit register pair)
src1_l or *src2_l*	lsb32 of 40-bit *src* (placed in even-numbered register of a 64-bit register pair)
src1_o or *src2_o*	msb32 of 64-bit *src* (placed in odd-numbered register of 64-bit register pair)
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)
ucst*n*	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
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-ORs
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
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
→	Assignment
+	Addition
++	Increment by 1
×	Multiplication

Table 3-1. Instruction Operation and Execution Notations (continued)

Symbol	Meaning
-	Subtraction
==	Equal to
>	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.

Table 3-2. Instruction Syntax and Opcode Notations

Symbol	Meaning
baseR	base address register
creg	3-bit field specifying a conditional register, see Section 3.5
cst	constant
csta	constant a
cstb	constant b
cstn	n-bit constant field
dst	destination
dw	doubleword; 0 = word, 1 = doubleword
fcyc	SPLOOP fetch cycle
fstg	SPLOOP fetch stage
h	MVK or MVKH instruction
ii_n	bit n of the constant *ii*
ld/st	load or store; 0 = store, 1 = load
mode	addressing mode, see Section 3.8
na	nonaligned; 0 = aligned, 1 = nonaligned
N3	3-bit field
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
ptr	offset from either A4-A7 or B4-B7 depending on the value of the *s* bit. The *ptr* field is the 2 least-significant bits of the *src2* (*baseR*) field—bit 2 of register address is forced to 1.
r	LDDW/LDNDW/LDNW 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
stg_n	bit n of the constant *stg*
sz	data size select; 0 = primary size, 1 = secondary size (see Section 3.9.2.2)
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

3.2.1 32-Bit Opcode Maps

The C64x CPU and C64x+ CPU 32-bit opcodes are mapped in Appendix C through Appendix H.

3.2.2 16-Bit Opcode Maps

The C64x+ CPU 16-bit opcodes used for compact instructions are mapped in Appendix C through Appendix H. See Section 3.9 for more information about compact instructions.

3.3 Delay Slots

The execution of the additional instructions can be defined in terms of delay slots. The number of delay slots is equivalent to the number of additional cycles required after the source operands are read for the result to be available for reading. For a single-cycle type instruction (such as **CMPGT2**), source operands read in cycle *i* produce a result that can be read in cycle *i* + 1. For a 2-cycle instruction (such as **AVGU4**), source operands read in cycle *i* produce a result that can be read in cycle *i* + 2. For a four-cycle instruction (such as **DOTP2**), source operands read in cycle *i* produce a result that can be read in cycle *i* + 4. Table 3-3 shows the number of delay slots associated with each type of instruction.

Delay slots are equivalent to an execution or result latency. All of the instructions in the C64x and C64x+ DSP have a functional unit latency of 1. This means that a new instruction can be started on the functional unit each cycle. Single-cycle throughput is another term for single-cycle functional unit latency.

Table 3-3. Delay Slot and Functional Unit Latency

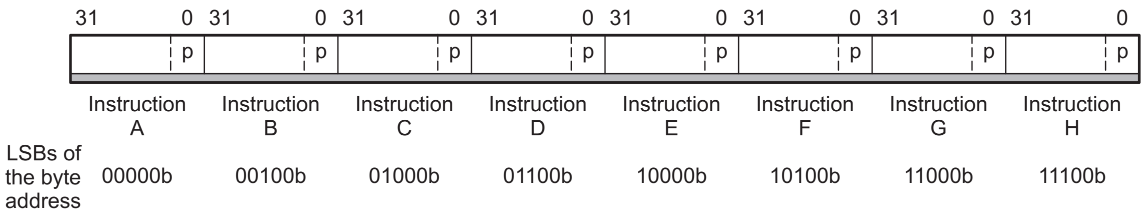
Instruction Type	Delay Slots	Functional Unit Latency	Read Cycles [1]	Write Cycles [1]	Branch Taken [1]
NOP (no operation)	0	1			
Store	0	1	i	i	
Single cycle	0	1	i	i	
Two cycle	1	1	i	i + 1	
Multiply (16 × 16)	1	1	i	i + 1	
Four cycle	3	1	i	i + 3	
Load	4	1	i	i, i + 4[2]	
Branch	5	1	i[3]		i + 5

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

3.4 Parallel Operations

Instructions are always fetched eight words at a time. This constitutes a *fetch packet*. On the C64x CPU this will always be eight instructions. On the C64x+ CPU, this may be as many as 14 instructions due to the existence of compact instructions in a header based fetch packet. The basic format of a fetch packet is shown in Figure 3-1. Fetch packets are aligned on 256-bit (8-word) boundaries.

Figure 3-1. Basic Format of a Fetch Packet



The C64x+ CPU supports compact 16-bit instructions. Unlike the normal 32-bit instructions, the *p*-bit information for compact instructions is not contained within the instruction opcode. Instead, the *p*-bit is contained within the *p*-bits field within the fetch packet header. See Section 3.9 for more information.

The execution of the individual noncompact 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 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 CPU, the execute packet can cross fetch packet boundaries, but will be limited to no more than eight instructions in a fetch packet. The last instruction in an execute packet will be marked with its *p*-bit cleared to zero. 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:

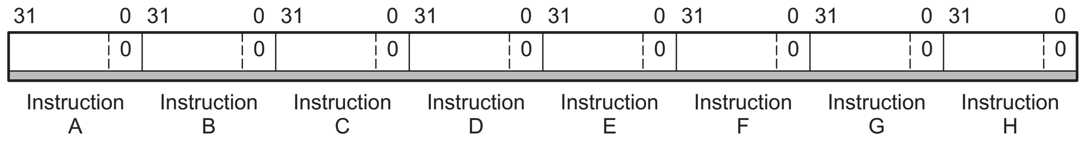
- Fully serial
- Fully parallel
- Partially serial

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

The eight instructions are executed sequentially.

This *p*-bit pattern:



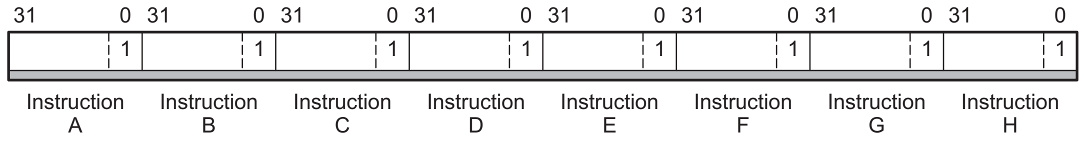
results in this execution sequence:

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

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

All eight instructions are executed in parallel.

This *p*-bit pattern:

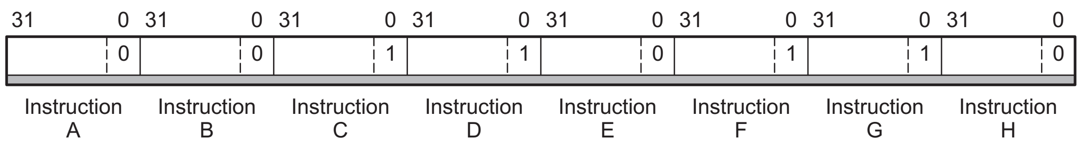


results in this execution sequence:

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

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

3.4.1 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
```

3.4.2 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.

3.5 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 3-4.

Compact (16-bit) instructions on the C64x+ DSP do not contain a *creg* field and always execute unconditionally. See Section 3.9 for more information.

Table 3-4. Registers That Can Be Tested by Conditional Operations

Specified Conditional Register		*creg*			*z*
	Bit:	**31**	**30**	**29**	**28**
Unconditional		0	0	0	0
Reserved		0	0	0	1
B0		0	0	1	z
B1		0	1	0	z
B2		0	1	1	z
A1		1	0	0	z
A2		1	0	1	z
A0		1	1	0	z
Reserved		1	1	1	x[1]

[1] 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.

The act of making an instruction conditional is often called predication and the conditional register is often called the predication register.

3.6 SPMASKed Operations

On the C64x+ CPU, the **SPMASK** and **SPMASKR** instructions can be used to inhibit the execution of instructions from the SPLOOP buffer. The selection of which instruction to inhibit can be specified by the **SPMASK** or **SPMASKR** instruction argument or can be marked by the addition of a caret (^) next to the parallel code marker as shown below:

```
     SPMASK
||^  LDW    .D1   *A0,A1            ;This instruction is SPMASKed
||^  LDW    .D2   *B0,B1            ;This instruction is SPMASKed
||   MPY    .M1   A3,A4,A5          ;This instruction is Not SPMASKed
```

See Chapter 7 for more information.

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

The .M unit has two 32-bit write ports; so the results of a 4-cycle 32-bit instruction and a 2-cycle 32-bit instruction operating on the same .M unit can write their results on the same instruction cycle. Any other combination of parallel writes on the .M unit will result in a conflict. On the C64x+ DSP this will result in an exception.

On the C64x DSP and C64x+ DSP , this will result in erroneous values being written to the destination registers.

For example, the following sequence is valid and results in both A2 and A5 being written by the .M1 unit on the same cycle.

```
DOTP2  .M1   A0,A1,A2            ;This instruction has 3 delay slots
NOP
AVG2   .M1   A4,A5               ;This instruction has 1 delay slot
NOP                              ;Both A2 and A5 get written on this cycle
```

The following sequence is invalid. The attempt to write 96 bits of output through 64-bits of write port will fail.

```
SMPY2  .M1   A5,A6,A9:A8         ;This instruction has 3 delay slots; but generates a 64 bit
result
NOP
MPY    .M1   A1,A2,A3            ;This instruction has 1 delay slot
NOP
```

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

Up to two units (.S, .L, .D, 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) provided that each unit is reading the same operand.

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

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

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

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

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

The following execute packet is invalid because more than two functional units use the same cross path operand:

```
   MV .L2X A0, B0 ; 1st cross path move
|| MV .S2X A0, B1 ; 2nd cross path move
|| MV .D2X A0, B2 ; 3rd cross path move
```

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

3.7.4 Cross Path Stalls

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

Here are some examples:

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


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


LDW  .D1 *++A0[1], A1  ; / No stall is introduced; A1 is the load
                       ;   destination
NOP  4                 ;   NOP 4 represents 4 instructions to
ADD  .S2X  A1, B0, B1  ; \ be executed between the load and add.
LDW  .D1  *++A0[1], A1 ; / Stall is introduced; A0 is updated
ADD  .S2X   A0, B0, B1 ;   1 cycle before it is used as a
                       ; \ cross path source
```

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

3.7.5 Constraints on Loads and Stores

The data address paths named DA1 and DA2 are each connected to the .D units in both data paths. Load 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 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. LD1 is comprised of LD1a and LD1b to support 64-bit loads; ST1 is comprised of ST1a and ST1b to support 64-bit stores. Similarly, T2 consists of the DA2 address path and the LD2 and ST2 data paths. LD2 is comprised of LD2a and LD2b to support 64-bit loads; ST2 is comprised of ST2a and ST2b to support 64-bit stores. The T1 and T2 designations appear in the functional unit fields for load and store instructions.

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

The following execute packet is invalid:

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

The following execute packet is valid:

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

3.7.6 Constraints on Long (40-Bit) Data

Both the C62x and C67x device families had constraints on the number of simultaneous reads and writes of 40-bit data due to shared data paths.

The C64x and C64x+ CPU maintain separate datapaths to each functional unit, so these constraints are removed.

The following, for example, is valid:

```
    DDOTPL2   .M1    A1:A0,A2,A5:A4
||  DDOTPL2   .M2    B1:B0,B2,B5:B4
||  STDW      .D1    A9:A8,*A6
||  STDW      .D2    B9:B8,*B6
||  SUB       .L1    A25:A24,A20,A31:A30
||  SUB       .L2    B25:B24,B20,B31:B30
||  SHL       .S1    A11:A10,5,A13:A12
||  SHL       .S2    B11:B10,8,B13:B12
```

3.7.7 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
```

3.7.8 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 3-2 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 3-2. 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     ; /
```

3.7.9 Constraints on AMR Writes

A write to the addressing mode register (AMR) using the **MVC** instruction that is immediately followed by a **LD, ST, ADDA,** or **SUBA** instruction causes a 1 cycle stall, if the **LD, ST, ADDA,** or **SUBA** instruction uses the A4-A7 or B4-B7 registers for addressing.

3.7.10 Constraints on Multicycle NOPs

Two instructions that generate multicycle NOPs cannot share the same execute packet. Instructions that generate a multicycle **NOP** are:

- NOP *n* (where *n* > 1)
- IDLE
- BNOP target, *n* (for all values of *n*, regardless of predication)
- ADDKPC label, reg, *n* (for all values of *n*, regardless of predication)

3.7.11 Constraints on Unitless Instructions

3.7.11.1 SPLOOP Restrictions

The **NOP**, **NOP *n***, and **BNOP** instructions are the only unitless instructions allowed to be used in an SPLOOP(D/W) body. The assembler disallows the use of any other unitless instruction in the loop body.

See Chapter 7 for more information.

3.7.11.2 BNOP <disp>,n

A **BNOP** instruction cannot be placed in parallel with the following instructions if the **BNOP** has a non-zero NOP count:

- ADDKPC
- CALLP
- NOP *n*

3.7.11.3 DINT

A **DINT** instruction cannot be placed in parallel with the following instructions:

- MVC reg, TSR
- MVC reg, CSR
- B IRP
- B NRP
- IDLE
- NOP *n* (if *n* > 1)
- RINT
- SPKERNEL(R)
- SPLOOP(D/W)
- SPMASK(R)
- SWE
- SWENR

A **DINT** instruction can be placed in parallel with the **NOP** instruction.

3.7.11.4 IDLE

An **IDLE** instruction cannot be placed in parallel with the following instructions:

- DINT
- NOP *n* (if *n* > 1)
- RINT
- SPKERNEL(R)
- SPLOOP(D/W)
- SPMASK(R)
- SWE
- SWENR

An **IDLE** instruction can be placed in parallel with the **NOP** instruction.

3.7.11.5 NOP n

A **NOP *n*** (with *n* > 1) instruction cannot be placed in parallel with other multicycle **NOP** counts (**ADDKPC**, **BNOP**, **CALLP**) with the exception of another **NOP *n*** where the NOP count is the same. A **NOP *n*** (with *n* > 1) instruction cannot be placed in parallel with the following instructions:

- DINT
- IDLE
- RINT
- SPKERNEL(R)
- SPLOOP(D/W)
- SPMASK(R)
- SWE
- SWENR

3.7.11.6 RINT

A **RINT** instruction cannot be placed in parallel with the following instructions:

- MVC reg, TSR
- MVC reg, CSR
- B IRP
- B NRP
- DINT
- IDLE
- NOP *n* (if *n* > 1)
- SPKERNEL(R)
- SPLOOP(D/W)
- SPMASK(R)
- SWE
- SWENR

A **RINT** instruction can be placed in parallel with the **NOP** instruction.

3.7.11.7 SPKERNEL(R)

An **SPKERNEL(R)** instruction cannot be placed in parallel with the following instructions:

- DINT
- IDLE
- NOP *n* (if *n* > 1)
- RINT
- SPLOOP(D/W)

- SPMASK(R)
- SWE
- SWENR

An **SPKERNEL(R)** instruction can be placed in parallel with the **NOP** instruction.

3.7.11.8 SPLOOP(D/W)

An **SPLOOP(D/W)** instruction cannot be placed in parallel with the following instructions:

- DINT
- IDLE
- NOP *n* (if *n* > 1)
- RINT
- SPKERNEL(R)
- SPMASK(R)
- SWE
- SWENR

An **SPLOOP(D/W)** instruction can be placed in parallel with the **NOP** instruction:

3.7.11.9 SPMASK(R)

An **SPMASK(R)** instruction cannot be placed in parallel with the following instructions:

- DINT
- IDLE
- NOP *n* (if *n* > 1)
- RINT
- SPLOOP(D/W)
- SPKERNEL(R)
- SWE
- SWENR

An **SPMASK(R)** instruction can be placed in parallel with the **NOP** instruction.

3.7.11.10 SWE

An **SWE** instruction cannot be placed in parallel with the following instructions:

- DINT
- IDLE
- NOP *n* (if *n* > 1)
- RINT
- SPLOOP(D/W)
- SPKERNEL(R)
- SWENR

An **SWE** instruction can be placed in parallel with the **NOP** instruction.

3.7.11.11 SWENR

An **SWENR** instruction cannot be placed in parallel with the following instructions:

- DINT
- IDLE
- NOP *n* (if *n* > 1)
- RINT
- SPLOOP(D/W)

- SPKERNEL(R)
- SWE

An **SWENR** instruction can be placed in parallel with the **NOP** instruction.

3.8 Addressing Modes

The addressing modes on the DSP are linear, circular using BK0, and circular using BK1. The addressing mode is specified by the addressing mode register (AMR), described in Section 2.8.3.

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, LDNDW, LDNW, STNDW, STNW, LDDW, STDW, ADDAB/ADDAH/ADDAW/ADDAD,** and **SUBAB/SUBAH/SUBAW** instructions all use AMR to determine what type of address calculations are performed for these registers. There is no **SUBAD** instruction.

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). The **LDNDW** and **STNDW** instructions also support nonscaled offsets. In nonscaled mode, the *offsetR/cst* is not shifted before adding or subtracting from the *baseR*.

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 2.8.3.

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 [(1)]	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

(1) **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 [(1)]
A4	0000 0100h	0000 0106h
A1	0000 0013h	0000 0013h

(1) **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).

3.8.2.3 Circular Addressing Considerations with Nonaligned Memory

Circular addressing may be used with nonaligned accesses. When circular addressing is enabled, address updates and memory accesses occur in the same manner as for the equivalent sequence of byte accesses.

On the C64x CPU, the only restriction is that the circular buffer size be at least as large as the data size being accessed. Nonaligned access to circular buffers that are smaller than the data being read will cause undefined results.

On the C64x+ CPU, the circular buffer size must be at least 32 bytes. Nonaligned access to circular buffers that are smaller than 32 bytes will cause undefined results.

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

Consider, for example, a circular buffer size of 16 bytes. A circular buffer of this size at location 20h, would look like this in physical memory:

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

The effect of circular buffering is to make it so that memory accesses and address updates in the 20h-2Fh range stay completely inside this range. Effectively, the memory map behaves in this manner:

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

Example 3-6 shows an **LDNW** 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. The buffer starts at address 0020h and ends at 0040h. The register A4 is initialized to the address 003Ah.

Example 3-6. LDNW in Circular Mode

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

	Before LDNW		1 cycle after LDNW [1]		5 cycles after LDNW
A4	0000 003Ah	A4	0000 0022h	A4	0000 0022h
A1	xxxx xxxxh	A1	xxxx xxxxh	A1	5678 9ABCh
mem 0022h	5678 9ABCh	mem 0022h	5678 9ABCh	mem 0022h	5678 9ABCh

[1] **Note:** 2h words is 8h bytes. 8h bytes is 2 bytes beyond the 32-byte (20h) boundary starting at address 003Ah; thus, it is wrapped around to 0022h (003Ah + 8h = 0022h).

3.8.3 Syntax for Load/Store Address Generation

The 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 3-5 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 3-6 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 3-5. 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 3-6. 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 Compact Instructions on the C64x+ CPU

The C64x+ CPU supports a header based set of 16-bit-wide compact instructions in addition to the normal 32-bit wide instructions. The C64x CPU does not support compact instructions.

3.9.1 Compact Instruction Overview

The availability of compact instructions is enabled by the replacement of the eighth word of a fetch packet with a 32-bit header word. The header word describes which of the other seven words of the fetch packet contain compact instructions, which of the compact instructions in the fetch packet operate in parallel, and also contains some decoding information which supplements the information contained in the 16-bit compact opcode. Table 3-7 compares the standard fetch packet with a header-based fetch packet containing compact instructions.

Table 3-7. C64x+ CPU Fetch Packet Types

Standard C6000 Fetch Packet

Word	
0	32-bit opcode
1	32-bit opcode
2	32-bit opcode
3	32-bit opcode
4	32-bit opcode
5	32-bit opcode
6	32-bit opcode
7	32-bit opcode

Header-Based Fetch Packet

Word		
0	16-bit opcode	16-bit opcode
1	32-bit opcode	
2	16-bit opcode	16-bit opcode
3	32-bit opcode	
4	16-bit opcode	16-bit opcode
5	32-bit opcode	
6	16-bit opcode	16-bit opcode
7	Header	

Within the other seven words of the fetch packet, each word may be composed of a single 32-bit opcode or two 16-bit opcodes. The header word specifies which words contain compact opcodes and which contain 32-bit opcodes.

The compiler will automatically code instructions as 16-bit compact instructions when possible.

There are a number of restrictions to the use of compact instructions:

- No dedicated predication field
- 3-bit register address field
- Very limited 3 operand instructions
- Subset of 32-bit instructions

3.9.2 Header Word Format

Figure 3-3 describes the format of the compact instruction header word.

Figure 3-3. Compact Instruction Header Format

31	30	29	28	27 – 21	20 – 14	13 – 0
1	1	1	0	Layout	Expansion	p-bits
				7	7	14

Bits 27-21 (Layout field) indicate which words in the fetch packet contain 32-bit opcodes and which words contain two 16-bit opcodes.

Bits 20-14 (Expansion field) contain information that contributes to the decoding of all compact instructions in the fetch packet.

Bits 13-0 (p-bits field) specify which compact instructions are run in parallel.

3.9.2.1 Layout Field in Compact Header Word

Bits 27-21 of the compact instruction header contains the layout field. This field specifies which of the other seven words in the current fetch packet contain 32-bit full-sized instructions and which words contain two 16-bit compact instructions.

Figure 3-4 shows the layout field in the compact header word and Table 3-8 describes the bits.

Figure 3-4. Layout Field in Compact Header Word

27	26	25	24	23	22	21
L7	L6	L5	L4	L3	L2	L1

Table 3-8. Layout Field Description in Compact Instruction Packet Header

Bit	Field	Value	Description
27	L7	0	Seventh word of fetch packet contains a single 32-bit opcode.
		1	Seventh word of fetch packet contains two 16-bit compact instructions.
26	L6	0	Sixth word of fetch packet contains a single 32-bit opcode.
		1	Sixth word of fetch packet contains two 16-bit compact instructions.
25	L5	0	Fifth word of fetch packet contains a single 32-bit opcode.
		1	Fifth word of fetch packet contains two 16-bit compact instructions.
24	L4	0	Fourth word of fetch packet contains a single 32-bit opcode.
		1	Fourth word of fetch packet contains two 16-bit compact instructions.
23	L3	0	Third word of fetch packet contains a single 32-bit opcode.
		1	Third word of fetch packet contains two 16-bit compact instructions.
22	L2	0	Second word of fetch packet contains a single 32-bit opcode.
		1	Second word of fetch packet contains two 16-bit compact instructions.
21	L1	0	First word of fetch packet contains a single 32-bit opcode.
		1	First word of fetch packet contains two 16-bit compact instructions.

3.9.2.2 Expansion Field in Compact Header Word

Bits 20-14 of the compact instruction header contains the opcode expansion field. This field specifies properties that apply to all compact instructions contained in the current fetch packet.

Figure 3-5 shows the expansion field in the compact header word and Table 3-9 describes the bits.

Figure 3-5. Expansion Field in Compact Header Word

20	19	18 16	15	14
PROT	RS	DSZ	BR	SAT

Table 3-9. Expansion Field Description in Compact Instruction Packet Header

Bit	Field	Value	Description
20	PROT	0	Loads are nonprotected (NOPs must be explicit).
		1	Loads are protected (4 NOP cycles added after every LD instruction).
19	RS	0	Instructions use low register set for data source and destination.
		1	Instructions use high register set for data source and destination.
18-16	DSZ	0-7h	Defines primary and secondary data size (see Table 3-10)
15	BR	0	Compact instructions in the S unit are not decoded as branches
		1	Compact Instructions in the S unit are decoded as branches.
14	SAT	0	Compact instructions do not saturate.
		1	Compact instructions saturate.

Bit 20 (PROT) selects between protected and nonprotected mode for all **LD** instructions within the fetch packet. When PROT is 1, four cycles of NOP are added after each **LD** instruction within the fetch packet whether the **LD** is in 16-bit compact format or 32-bit format.

Bit 19 (RS) specifies which register set is used by compact instructions within the fetch packet. The register set defines which subset of 8 registers on each side are data registers. The 3-bit register field in the compact opcode indicates which one of eight registers is used. When RS is 1, the high register set (A16-A23 and B16-B23) is used; when RS is 0, the low register set (A0-A7 and B0-B7) is used.

Bits 18-16 (DSZ) determine the two data sizes available to the compact versions of the **LD** and **ST** instructions in a fetch packet. Bit 18 determines the primary data size that is either word (W) or doubleword (DW). In the case of DW, an opcode bit selects between aligned (DW) and nonaligned (NDW) accesses. Bits 17 and 16 determine the secondary data size: byte unsigned (BU), byte (B), halfword unsigned (HU), halfword (H), word (W), or nonaligned word (NW). Table 3-10 describes how the bits map to data size.

Bit 15 (BR). When BR is 1, instructions in the S unit are decoded as branches.

Bit 14 (SAT). When SAT is 1, the **ADD, SUB, SHL, MPY, MPYH, MPYLH,** and **MPYHL** instructions are decoded as **SADD, SUBS, SSHL, SMPY, SMPYH, SMPYLH,** and **SMPYHL**, respectively.

Table 3-10. LD/ST Data Size Selection

DSZ Bits			Primary Data Size [1]	Secondary Data Size [2]
18	17	16		
0	0	0	W	BU
0	0	1	W	B
0	1	0	W	HU
0	1	1	W	H
1	0	0	DW/NDW	W
1	0	1	DW/NDW	B
1	1	0	DW/NDW	NW
1	1	1	DW/NDW	H

[1] Primary data size is word W) or doubleword (DW). In the case of DW, aligned (DW) or nonaligned (NDW).
[2] Secondary data size is byte unsigned (BU), byte (B), halfword unsigned (HU), halfword (H), word (W), or nonaligned word (NW).

3.9.2.3 P-bit Field in Compact Header Word

Unlike normal 32-bit instructions in which the *p*-bit filed in each opcode determines whether the instruction executes in parallel with other instructions; the parallel/nonparallel execution information for compact instructions is contained in the compact instruction header word.

Bits 13-0 of the compact instruction header contain the *p*-bit field. This field specifies which of the compact instructions within the current fetch packet are executed in parallel. If the corresponding bit in the layout field is 0 (indicating that the word is a noncompact instruction), then the bit in the *p*-bit field must be zero; that is, 32-bit instructions within compact fetch packets use their own *p*-bit field internal to the 32-bit opcode; therefore, the associated *p*-bit field in the header should always be zero.

Figure 3-6 shows the *p*-bits field in the compact header word and Table 3-11 describes the bits.

Figure 3-6. P-bits Field in Compact Header Word

13	12	11	10	9	8	7	6	5	4	3	2	1	0
P13	P12	P11	P10	P9	P8	P7	P6	P5	P4	P3	P2	P1	P0

Table 3-11. P-bits Field Description in Compact Instruction Packet Header

Bit	Field	Value	Description
13	P13	0	Word 6 (16 most-significant bits) of fetch packet has parallel bit cleared.
		1	Word 6 (16 most-significant bits) of fetch packet has parallel bit set.
12	P12	0	Word 6 (16 least-significant bits) of fetch packet has parallel bit cleared.
		1	Word 6 (16 least-significant bits) of fetch packet has parallel bit set.
11	P11	0	Word 5 (16 most-significant bits) of fetch packet has parallel bit cleared.
		1	Word 5 (16 most-significant bits) of fetch packet has parallel bit set.
10	P10	0	Word 5 (16 least-significant bits) of fetch packet has parallel bit cleared.
		1	Word 5 (16 least-significant bits) of fetch packet has parallel bit set.
9	P9	0	Word 4 (16 most-significant bits) of fetch packet has parallel bit cleared.
		1	Word 4 (16 most-significant bits) of fetch packet has parallel bit set.
8	P8	0	Word 4 (16 least-significant bits) of fetch packet has parallel bit cleared.
		1	Word 4 (16 least-significant bits) of fetch packet has parallel bit set.
7	P7	0	Word 3 (16 most-significant bits) of fetch packet has parallel bit cleared.
		1	Word 3 (16 most-significant bits) of fetch packet has parallel bit set.
6	P6	0	Word 3 (16 least-significant bits) of fetch packet has parallel bit cleared.
		1	Word 3 (16 least-significant bits) of fetch packet has parallel bit set.
5	P5	0	Word 2 (16 most-significant bits) of fetch packet has parallel bit cleared.
		1	Word 2 (16 most-significant bits) of fetch packet has parallel bit set.
4	P4	0	Word 2 (16 least-significant bits) of fetch packet has parallel bit cleared.
		1	Word 2 (16 least-significant bits) of fetch packet has parallel bit set.
3	P3	0	Word 1 (16 most-significant bits) of fetch packet has parallel bit cleared.
		1	Word 1 (16 most-significant bits) of fetch packet has parallel bit set.
2	P2	0	Word 1 (16 least-significant bits) of fetch packet has parallel bit cleared.
		1	Word 1 (16 least-significant bits) of fetch packet has parallel bit set.
1	P1	0	Word 0 (16 most-significant bits) of fetch packet has parallel bit cleared.
		1	Word 0 (16 most-significant bits) of fetch packet has parallel bit set.
0	P0	0	Word 0 (16 least-significant bits) of fetch packet has parallel bit cleared.
		1	Word 0 (16 least-significant bits) of fetch packet has parallel bit set.

3.9.3 Processing of Fetch Packets

The header information is used to fully define the 32-bit version of the 16-bit instructions. In the case where an execute packet crosses fetch packet boundaries, there are two headers in use simultaneously. Each instruction uses the header information from its fetch packet header.

3.9.4 Execute Packet Restrictions

Execute packets that span fetch packet boundaries may not be the target of branches in the case where one of the two fetch packets involved are header-based. The only exception to this is where an interrupt is taken in the cycle before a spanning execute packet reaches E1. The target of the return may be a normally disallowed target.

If the execute packet contains eight instructions, then neither of the two fetch packets may be header-based.

3.9.5 Available Compact Instructions

Table 3-12 lists the available compact instructions and their functional unit.

Table 3-12. Available Compact Instructions

Instruction	L Unit	M Unit	S Unit	D Unit
ADD	✓		✓	✓
ADDAW				✓
ADDK			✓	
AND	✓			
BNOP displacement			✓	
CALLP			✓	
CLR			✓	
CMPEQ	✓			
CMPGT	✓			
CMPGTU	✓			
CMPLT	✓			
CMPLTU	✓			
EXT			✓	
EXTU			✓	
LDB				✓
LDBU				✓
LDDW				✓
LDH				✓
LDHU				✓
LDNDW				✓
LDNW				✓
LDW				✓
LDW (15-bit offset)				✓
MPY		✓		
MPYH		✓		
MPYHL		✓		
MPYLH		✓		
MV	✓		✓	✓
MVC			✓	
MVK	✓		✓	✓
NEG	✓			

Table 3-12. Available Compact Instructions (continued)

Instruction	L Unit	M Unit	S Unit	D Unit
NOP		No unit		
OR	✓			
SADD	✓		✓	
SET			✓	
SHL			✓	
SHR			✓	
SHRU			✓	
SMPY		✓		
SMPYH		✓		
SMPYHL		✓		
SMPYLH		✓		
SPKERNEL		No unit		
SPLOOP		No unit		
SPLOOPD		No unit		
SPMASK		No unit		
SPMASKR		No unit		
SSHL			✓	
SSUB	✓			
STB				✓
STDW				✓
STH				✓
STNDW				✓
STNW				✓
STW				✓
STW (15-bit offset)				✓
SUB	✓		✓	✓
SUBAW				✓
XOR	✓			

3.10 Instruction Compatibility

The C62x, C64x, and C64x+ DSPs share an instruction set. All of the instructions valid for the C62x DSP are also valid for the C64x and C64x+ DSPs. The C64x/C64x+ DSP adds functionality to the C62x DSP with some unique instructions. See Appendix A for a list of the instructions that are common to the C62x, C64x, and C64x+ DSPs.

3.11 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.

There are instructions that can be executed on more than one functional unit. Table 3-13 shows how this is documented for the **ADD** instruction. This instruction has three opcode map fields: *src1*, *src2*, and *dst*. In the fifth 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 ninth group, *src1*, *src2*, and *dst* are *int*, *cst5*, and *int*, respectively. The *u* in front of the *cst5* operand signifies that *src1* (*ucst5*) is an unsigned value. 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).

Compatibility

The C62x, C64x, and C64x+ DSPs share an instruction set. All of the instructions valid for the C62x DSP are also valid for the C64x and C64x+ DSPs. 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

The execution describes the processing that takes place when the instruction is executed. The symbols are defined in Table 3-1. For example:

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

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

Execution for .D1, .D2 Opcodes

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

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 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 3.3 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.

Table 3-13. Relationships Between Operands, Operand Size, Functional Units, and Opfields for Example Instruction (ADD)

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

ABS *Absolute Value With Saturation*

Syntax

ABS (.unit) *src2, dst*

or

ABS (.unit) *src2_h:src2_l,dst_h:dst_l*

unit = .L1 or .L2

Compatibility C62x, C64x, and C64x+ CPU

Opcode

31 29	28	27 23	22 18	17	16	15	14	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*.

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

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

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

1. If *src2* > 0, then *src2* → *dst_h:dst_l*
2. If *src2* < 0 and *src2*≠ -2^{39}, then -*src2* → *dst_h:dst_l*
3. If *src2* = -2^{39}, then $2^{39} - 1$ → *dst_h:dst_l*

Execution

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

Pipeline

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

Instruction Type Single-cycle

Delay Slots 0

See Also ABS2

Examples

Example 1

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

	Before instruction			1 cycle after instruction	
A1	8000 4E3Dh	-2,147,463,619	A1	8000 4E3Dh	
A5	xxxx xxxxh		A5	7FFF B1C3h	2,147,463,619

Example 2

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

	Before instruction			1 cycle after instruction	
A1	3FF6 0010h	1,073,086,480	A1	3FF6 0010h	
A5	xxxx xxxxh		A5	3FF6 0010h	1,073,086,480

Example 3

```
ABS .L1 A1:A0,A5:A4
```

	Before instruction			1 cycle after instruction	
A0	FFFF FFFFh	1,073,086,480	A0	FFFF FFFFh	1,073,086,480
A1	0000 00FFh		A1	0000 00FFh	
A4	xxxx xxxxh		A4	0000 0001h	
A5	xxxx xxxxh		A5	0000 0000h	

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

Syntax

ABS2 (.unit) *src2, dst*

unit = .L1 or .L2

Compatibility

C64x and C64x+ CPU

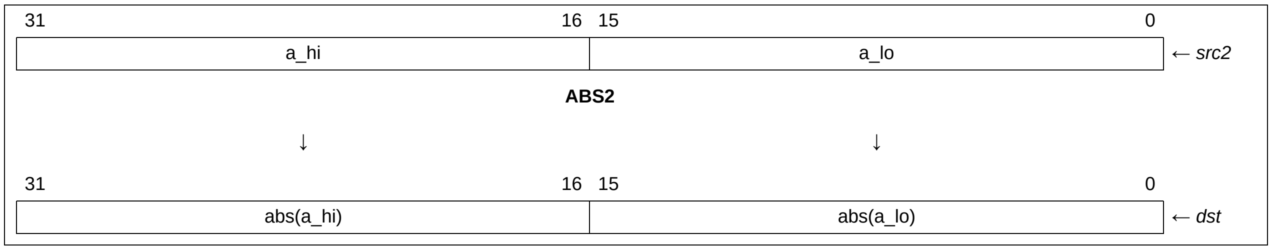
Opcode

31 29	28	27 23	22 18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	0	0	1	0	0	x	0	0	1	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*	xs2 s2	.L1, .L2

Description

The absolute values of the upper and lower halves of the *src2* operand are placed in the upper and lower halves of the *dst.*



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

1. If the value is between 0 and 2^{15}, then value → *dst*
2. If the value is less than 0 and not equal to -2^{15}, then -value → *dst*
3. If the value is equal to -2^{15}, then 2^{15} -1 → *dst*

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

Execution

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

Pipeline

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

Instruction Type Single-cycle

Delay Slots 0

See Also ABS

Examples

Example 1

```
ABS2 .L1 A0,A2
```

	Before instruction			1 cycle after instruction	
A0	FF68 4E3Dh	-152 20029	A0	FF68 4E3Dh	
A2	xxxx xxxxh		A2	0098 4E3Dh	152 20029

Example 2

```
ABS2 .L1 A0,A2
```

	Before instruction			1 cycle after instruction	
A0	3FF6 F105h	16374 -3835	A0	3FF6 F105h	
A2	xxxx xxxxh		A2	3FF6 0EFBh	16374 3835

ADD *Add Two Signed Integers Without Saturation*

Syntax

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

or

ADD (.L1 or .L2) *src1, src2_h:src2_l, dst_h:dst_l*

or

ADD (.D1 or .D2) *src2, src1, dst* (if the cross path form is not used)

or

ADD (.D1 or .D2) *src1, src2, dst* (if the cross path form is used)

or

ADD (.D1 or .D2) *src2, src1, dst* (if the cross path form is used with a constant)

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

Compatibility C62x, C64x, and C64x+ CPU

Compact Instruction Format

Unit	Opcode Format	Figure
.L	L3	Figure D-4
	L3i	Figure D-5
	Lx1	Figure D-11
.S	S3	Figure F-21
	Sx2op	Figure F-28
	Sx1	Figure F-30
.D	Dx2op	Figure C-18
.L, .S, .D	LSDx1	Figure G-4

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 (if the cross path form is not used)

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

Opcode .D unit (if the cross path form is used)

31 29	28	27 23	22 18	17 13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	1	0	1	0	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 *src2* *dst*	sint xsint sint	.D1, .D2

Opcode .D unit (if the cross path form is used with a constant)

31 29	28	27 23	22 18	17 13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	1	0	1	0	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 *src2* *dst*	scst5 xsint sint	.D1, .D2

Description for .D1, .D2 Opcodes *src1* is added to *src2*. The result is placed in *dst*.

Execution for .D1, .D2 Opcodes

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

Pipeline

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

Instruction Type Single-cycle

Delay Slots 0

See Also ADDU, ADD2, SADD

Examples

Example 1

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

	Before instruction			1 cycle after instruction	
A1	0000 325Ah	12,890	A1	0000 325Ah	
B1	FFFF FF12h	-238	B1	FFFF FF12h	
B2	xxxx xxxxh		B2	0000 316Ch	12,652

Example 2

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

	Before instruction				1 cycle after instruction		
A1	0000 325Ah		12,890	A1	0000 325Ah		
A3:A2	0000 00FFh	FFFF FF12h	-228[1]	A3:A2	0000 00FFh	FFFF FF12h	
A5:A4	0000 0000h	0000 0000h		A5:A4	0000 0000h	0000 316Ch	12,652[1]

[1] Signed 40-bit (long) integer

Example 3

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

	Before instruction			1 cycle after instruction	
A1	0000 325Ah	12,890	A1	0000 325Ah	
A6	xxxx xxxxh		A6	0000 324Dh	12,877

Example 4

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

	Before instruction			1 cycle after instruction	
A1	0000 325Ah	12,890	A1	0000 325Ah	
A6	xxxx xxxxh		A6	0000 3274h	12,916

Example 5

```
ADD .D1 B0,5,A2
```

	Before instruction		1 cycle after instruction	
B0	0000 0007h	B0	0000 0007h	
A2	xxxx xxxxh	A2	0000 000Ch	12

ADDAB *Add Using Byte Addressing Mode*

Syntax

ADDAB (.unit) *src2, src1, dst* (C64x and C64x+ CPU)

or

ADDAB (.unit) B14/B15, *ucst15, dst* (C64x+ CPU)

unit = .D1 or .D2

Compatibility C62x, C64x, and C64x+ CPU

Opcode C64x and C64x+ CPU

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 For the C64x and C64x+ CPU, *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 2.8.3).The result is placed in *dst*.

Execution

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

Pipeline

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

Opcode C64x+ CPU only

31	30	29	28	27 23	22 8	7	6	5	4	3	2	1	0
0	0	0	1	*dst*	*ucst15*	*y*	0	1	1	1	1	*s*	*p*
				5	15	1						1	1

Description For the C64x+ CPU, this instruction reads a register (*baseR*), B14 (y = 0) or B15 (y = 1), and adds a 15-bit unsigned constant (*ucst15*) to it, writing the result to a register (*dst*). This instruction is executed unconditionally, it cannot be predicated.

The offset, *ucst15*, is added to *baseR*. The result of the calculation is written into *dst*. The addressing arithmetic is always performed in linear mode.

The *s* bit determines the unit used (D1 or D2) and the file the destination is written to: *s* = 0 indicates the unit is D1 and *dst* is in the A register file; and *s* = 1 indicates the unit is D2 and *dst* is in the B register file.

Execution B14/B15 + *ucst15* → *dst*

Pipeline

Pipeline Stage	E1
Read	*B14/B15*
Written	*dst*
Unit in use	.D

Instruction Type Single-cycle

Delay Slots 0

See Also ADDAD, ADDAH, ADDAW

Examples

Example 1

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

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

(1) BK0 = 2: block size = 8
A4 in circular addressing mode using BK0

Example 2

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

	Before instruction [(1)]		1 cycle after instruction
B14	0020 1000h	A4	0020 1042h

(1) Using linear addressing.

Example 3

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

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

[1] Using linear addressing.

ADDAD *Add Using Doubleword Addressing Mode*

Syntax

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

unit = . D1 or .D2

Compatibility

C64x and C64x+ 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 2.8.3). *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 + src1 <<3 → dst
else nop
```

Pipeline

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

Instruction Type Single-cycle

Delay Slots 0

See Also ADDAB, ADDAH, ADDAW

Example

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

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

ADDAH *Add Using Halfword Addressing Mode*

Syntax

ADDAH (.unit) *src2, src1, dst* (C64x and C64x+ CPU)

or

ADDAH (.unit) B14/B15, *ucst15, dst* (C64x+ CPU)

unit = .D1 or .D2

Compatibility C62x, C64x, and C64x+ CPU

Opcode C64x and C64x+ CPU

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 For the C64x and C64x+ CPU, *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 2.8.3). *src1* is left shifted by 1. The result is placed in *dst*.

Execution

```
if (cond)     src2 + src1 <<1 → dst
else nop
```

Pipeline

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

Opcode C64x+ CPU only

31	30	29	28	27 23	22 8	7	6	5	4	3	2	1	0
0	0	0	1	*dst*	*ucst15*	*y*	1	0	1	1	1	*s*	*p*
				5	15	1						1	1

Description For the C64x+ CPU, this instruction reads a register (*baseR*), B14 (y = 0) or B15 (y = 1), and adds a scaled 15-bit unsigned constant (*ucst15*) to it, writing the result to a register (*dst)*. This instruction is executed unconditionally, it cannot be predicated.

The offset, *ucst15*, is scaled by a left-shift of 1 and added to *baseR*. The result of the calculation is written into *dst*. The addressing arithmetic is always performed in linear mode.

The *s* bit determines the unit used (D1 or D2) and the file the destination is written to: *s* = 0 indicates the unit is D1 and *dst* is in the A register file; and *s* = 1 indicates the unit is D2 and *dst* is in the B register file.

Execution B14/B15 + (*ucst15* << 1) → *dst*

Pipeline

Pipeline Stage	E1
Read	*B14/B15*
Written	*dst*
Unit in use	.D

Instruction Type Single-cycle

Delay Slots 0

See Also ADDAB, ADDAD, ADDAW

Examples

Example 1

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

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

(1) BK0 = 2: block size = 8
A4 in circular addressing mode using BK0

Example 2

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

	Before instruction [(1)]		1 cycle after instruction
B14	0020 1000h	A4	0020 1084h

(1) Using linear addressing.

Example 3

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

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

[1] Using linear addressing.

ADDAW *Add Using Word Addressing Mode*

Syntax

ADDAW (.unit) *src2, src1, dst* (C64x and C64x+ CPU)

or

ADDAW (.unit) B14/B15, *ucst15, dst* (C64x+ CPU)

unit = .D1 or .D2

Compatibility C62x, C64x, and C64x+ CPU

Compact Instruction Format

Unit	Opcode Format	Figure
.D	Dx5	Figure C-19
	Dx5p	Figure C-20

Opcode C64x and C64x+ CPU

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 For the C64x and C64x+ CPU, *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 2.8.3). *src1* is left shifted by 2. The result is placed in *dst*.

Execution

if (cond)	*src2* + *src1* <<2 → *dst*
else nop	

Pipeline

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

Opcode C64x+ CPU only

31	30	29	28	27 … 23	22 … 8	7	6	5	4	3	2	1	0
0	0	0	1	*dst*	*ucst15*	*y*	1	1	1	1	1	*s*	*p*
				5	15	1						1	1

Description For the C64x+ CPU, this instruction reads a register (*baseR*), B14 (y = 0) or B15 (y = 1), and adds a scaled 15-bit unsigned constant (*ucst15*) to it, writing the result to a register (*dst)*. This instruction is executed unconditionally, it cannot be predicated.

The offset, *ucst15*, is scaled by a left-shift of 2 and added to *baseR*. The result of the calculation is written into *dst*. The addressing arithmetic is always performed in linear mode.

The *s* bit determines the unit used (D1 or D2) and the file the destination is written to: *s* = 0 indicates the unit is D1 and *dst* is in the A register file; and *s* = 1 indicates the unit is D2 and *dst* is in the B register file.

Execution B14/B15 + (*ucst15* << 2) → *dst*

Pipeline

Pipeline Stage	E1
Read	*B14/B15*
Written	*dst*
Unit in use	.D

Instruction Type Single-cycle

Delay Slots 0

See Also ADDAB, ADDAD, ADDAH

Examples

Example 1

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

	Before instruction [(1)]		1 cycle after instruction
A4	0002 0000h	A4	0002 0000h
AMR	0002 0001h	AMR	0002 0001h

(1) BK0 = 2: block size = 8
A4 in circular addressing mode using BK0

Example 2

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

	Before instruction [(1)]		1 cycle after instruction
B14	0020 1000h	A4	0020 1108h

(1) Using linear addressing.

Example 3

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

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

[1] Using linear addressing.

ADDK *Add Signed 16-Bit Constant to Register*

Syntax

ADDK (.unit) *cst, dst*

unit = .S1 or .S2

Compatibility C62x, C64x, and C64x+ CPU

Compact Instruction Format

Unit	Opcode Format	Figure
.S	Sx5	Figure F-29

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)     cst16 + dst → 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	2,176,993	A1	0021 740Ah	2,192,394

ADDKPC *Add Signed 7-Bit Constant to Program Counter*

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

unit = .S2

Compatibility C64x and C64x+ CPU

Opcode

31–29	28	27–23	22–16	15–13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src1*	*src2*	0	0	0	0	1	0	1	1	0	0	0	*s*	*p*
3	1	5	7	3												1	1

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

Description A 7-bit signed constant, *src1*, is shifted 2 bits to the left, then added to the address of the first instruction of the fetch packet that contains the **ADDKPC** instruction (PCE1). The result is placed in *dst*. The 3-bit unsigned constant, *src2*, specifies the number of NOP cycles to insert after the current instruction. This instruction helps reduce the number of instructions needed to set up the return address for a function call.

The following code:

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

could be replaced by:

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

The 7-bit value coded as *src1* is the difference between LABEL and PCE1 shifted right by 2 bits. The address of LABEL must be within 9 bits of PCE1.

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

The **ADDKPC** instruction cannot be paired with any other multicycle **NOP** instruction in the same execute packet. Instructions that generate a multicycle **NOP** are: **IDLE**, **BNOP**, and the multicycle **NOP**.

Execution

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

Pipeline

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

Instruction Type Single-cycle

Delay Slots 0

See Also B, BNOP

Example

```
ADDKPC .S2   LABEL,B3,4
LABEL:
```

	Before instruction [1]		1 cycle after instruction
PCE1	0040 13DCh		
B3	xxxx xxxxh	B3	0040 13E0h

[1] LABEL is equal to 0040 13DCh.

ADDSUB *Parallel ADD and SUB Operations On Common Inputs*

Syntax **ADDSUB** (.unit) *src1, src2, dst_o:dst_e*

unit = .L1 or .L2

Compatibility C64x+ CPU only

Opcode

31	30	29	28	27–24	23	22–18	17–13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	*dst*	0	*src2*	*src1*	x	0	0	0	1	1	0	0	1	1	0	*s*	*p*
				4		5	5	1											1	1

Opcode map field used...	For operand type...	Unit
src1	sint	.L1, .L2
src2	xsint	
dst	dint	

Description The following is performed in parallel:

1. *src2* is added to *src1*. The result is placed in *dst_o*.
2. *src2* is subtracted from *src1*. The result is placed in *dst_e*.

Execution

src1 + *src2* → *dst_o*
src1 - *src2* → *dst_e*

Instruction Type Single-cycle

Delay Slots 0

See Also ADDSUB2, SADDSUB

Examples

Example 1

```
ADDSUB .L1 A0,A1,A3:A2
```

	Before instruction		1 cycle after instruction
A0	0700 C005h	A2	0700 C006h
A1	FFFF FFFFh	A3	0700 C004h

Example 2

```
ADDSUB .L2X B0,A1,B3:B2
```

	Before instruction		1 cycle after instruction
B0	7FFF FFFFh	B2	7FFF FFFEh
A1	0000 0001h	B3	8000 0000h

ADDSUB2 *Parallel ADD2 and SUB2 Operations On Common Inputs*

Syntax **ADDSUB2** (.unit) *src1, src2, dst_o:dst_e*

unit = .L1 or .L2

Compatibility C64x+ CPU only

Opcode

31	30	29	28	27–24	23	22–18	17–13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	*dst*	0	*src2*	*src1*	x	0	0	0	1	1	0	1	1	1	0	*s*	*p*
				4		5	5	1											1	1

Opcode map field used...	For operand type...	Unit
src1 *src2* *dst*	sint xsint dint	.L1, .L2

Description For the **ADD2** operation, the upper and lower halves of the *src2* operand are added to the upper and lower halves of the *src1* 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_o*.

For the **SUB2** operation, the upper and lower halves of the *src2* operand are subtracted from the upper and lower halves of the *src1* 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_e*.

Execution

lsb16(*src1*) + lsb16(*src2*) → lsb16(*dst_o*)
msb16(*src1*) + msb16(*src2*) → msb16(*dst_o*)
lsb16(*src1*) - lsb16(*src2*) → lsb16(*dst_e*)
msb16(*src1*) - msb16(*src2*) → msb16(*dst_e*)

Instruction Type Single-cycle

Delay Slots 0

See Also ADDSUB, SADDSUB2

Examples **Example 1**

```
ADDSUB2 .L1 A0,A1,A3:A2
```

	Before instruction		1 cycle after instruction
A0	0700 C005h	A2	0701 C004h
A1	FFFF 0001h	A3	06FF C006h

Example 2

```
ADDSUB2 .L2X B0,A1,B3:B2
```

	Before instruction		1 cycle after instruction
B0	7FFF 8000h	B2	8000 8001h
A1	FFFF FFFFh	B3	7FFE 7FFFh

Example 3

```
ADDSUB2 .L1 A0,A1,A3:A2
```

	Before instruction		1 cycle after instruction
A0	9000 9000h	A2	1000 1000h
A1	8000 8000h	A3	1000 1000h

Example 4

```
ADDSUB2 .L1 A0,A1,A3:A2
```

	Before instruction		1 cycle after instruction
A0	9000 8000h	A2	1000 F000h
A1	8000 9000h	A3	1000 1000h

ADDU *Add Two Unsigned Integers Without Saturation*

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

unit = .L1 or .L2

Compatibility C62x, C64x, and C64x+ 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 → 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

Examples

Example 1

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

	Before instruction				1 cycle after instruction		
A1	0000 325Ah	12,890[1]		A1	0000 325Ah		
A2	FFFF FF12h	4,294,967,058[1]		A2	FFFF FF12h		
A5:A4	xxxx xxxxh			A5:A4	0000 0001h	0000 316Ch	4,294,979,948[2]

(1) Unsigned 32-bit integer
(2) Unsigned 40-bit (long) integer

Example 2

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

	Before instruction				1 cycle after instruction		
A1	0000 325Ah		12,890[1]	A1	0000 325Ah		
A3:A2	0000 00FFh	FFFF FF12h	1,099,511,627,538[2]	A3:A2	0000 00FFh	FFFF FF12h	
A5:A4	0000 0000h	0000 0000h	0	A5:A4	0000 0000h	0000 316Ch	12,652[2]

(1) Unsigned 32-bit integer
(2) 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, .S2, .L1, .L2, .D1, .D2

Compatibility C62x, C64x, and C64x+ CPU

Opcode .S unit

31 29	28	27 23	22 18	17 13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*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	i2	.S1, .S2
src2	xi2	
dst	i2	

Opcode .L Unit

31 29	28	27 23	22 18	17 13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	0	0	0	0	1	0	1	1	1	0	*s*	*p*
3	1	5	5	5	1											1	1

Opcode map field used...	For operand type...	Unit
src1	i2	.L1, .L2
src2	xi2	
dst	i2	

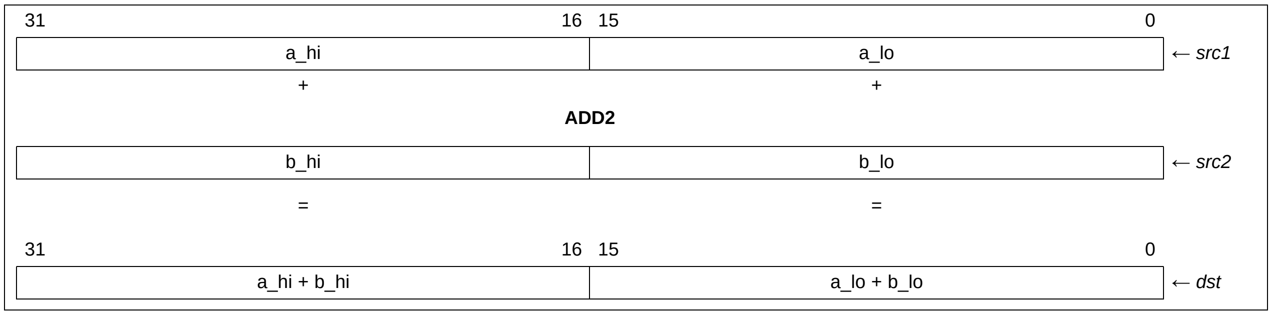
Opcode .D unit

31 29	28	27 23	22 18	17 13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	1	0	0	1	0	0	1	1	0	0	*s*	*p*
3	1	5	5	5	1											1	1

Opcode map field used...	For operand type...	Unit
src1	i2	.D1, .D2
src2	xi2	
dst	i2	

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 a 16-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, .L, .D

Instruction Type Single-cycle

Delay Slots 0

See Also ADD, ADD4, SADD2, SUB2

Examples **Example 1**

`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	

Example 2

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

	Before instruction			1 cycle after instruction	
A0	0021 37E1h	33 14305 signed	A0	0021 37E1h	
A1	039A E4B8h	922 -6984 signed	A1	039A E4B8h	
A2	xxxx xxxxh		A2	03BB 1C99h	955 7321 signed

ADD4 *Add Without Saturation, Four 8-Bit Pairs for Four 8-Bit Results*

Syntax **ADD4** (.unit) *src1, src2, dst*

unit = .L1 or .L2

Compatibility C64x and C64x+ CPU

Opcode

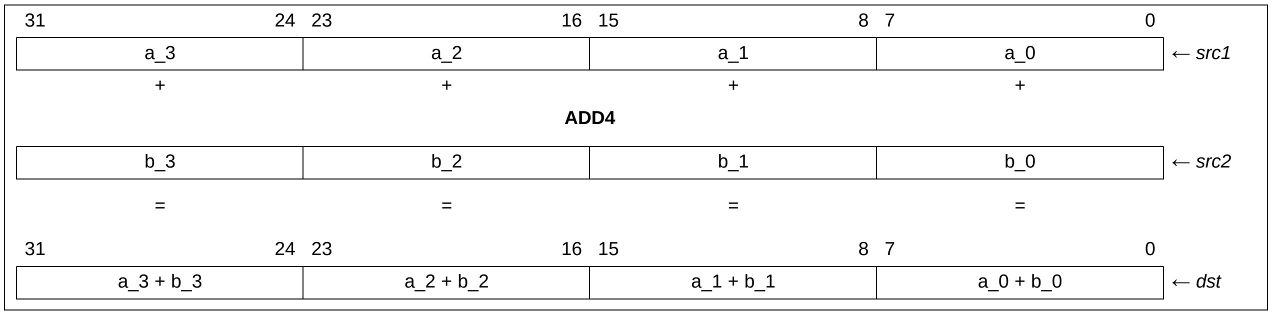
31–29	28	27–23	22–18	17–13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	1	1	0	0	1	0	1	1	1	0	*s*	*p*
3	1	5	5	5	1											1	1

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

Description Performs 2s-complement addition between packed 8-bit quantities. The values in *src1* and *src2* are treated as packed 8-bit data and the results are written into *dst* in a packed 8-bit format.

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

- The sum of *src1* byte0 and *src2* byte0 is placed in byte0 of *dst*.
- The sum of *src1* byte1 and *src2* byte1 is placed in byte1 of *dst*.
- The sum of *src1* byte2 and *src2* byte2 is placed in byte2 of *dst*.
- The sum of *src1* byte3 and *src2* byte3 is placed in byte3 of *dst*.



Execution

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

Pipeline

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

Instruction Type Single-cycle

Delay Slots 0

See Also ADD, ADD2, SADDU4, SUB4

Examples

Example 1

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

	Before instruction			1 cycle after instruction	
A0	FF 68 4E 3Dh	-1 104 78 61	A0	FF 68 4E 3Dh	
A1	3F F6 F1 05h	63 -10 -15 5	A1	3F F6 F1 05h	
A2	xxxx xxxxh		A2	3E 5E 3F 42h	62 94 63 66

Example 2

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

	Before instruction			1 cycle after instruction	
A0	4A E2 D3 1Fh	74 226 211 31	A0	4A E2 D3 1Fh	
A1	32 1A C1 28h	50 26 -63 40	A1	32 1A C1 28h	
A2	xxxx xxxxh		A2	7C FC 94 47h	124 252 148 71

AND *Bitwise AND*

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

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

Compatibility C62x, C64x, and C64x+ CPU

Compact Instruction Format

Unit	Opcode Format	Figure
.L	L2c	Figure D-7

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	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 1111
src1 *src2* *dst*	scst5 xuint uint	.S1, .S2	01 1110

Opcode .D unit

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

Opcode map field used...	For operand type...	Unit	Opfield
src1 *src2* *dst*	uint xuint uint	.D1, .D2	0110
src1 *src2* *dst*	scst5 xuint uint	.D1, .D2	0111

Description Performs 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 → dst
else nop
```

Pipeline

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

Instruction Type Single-cycle

Delay Slots 0

See Also ANDN, OR, XOR

Examples

Example 1

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

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

Example 2

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

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

ANDN *Bitwise AND Invert*

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

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

Compatibility C64x and C64x+ CPU

Opcode .L unit

31 29	28	27 23	22 18	17 13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	1	1	1	1	1	0	0	1	1	0	*s*	*p*
3	1	5	5	5	1											1	1

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

Opcode .S unit

31 29	28	27 23	22 18	17 13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	1	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	uint	.S1, .S2
src2	xuint	
dst	uint	

Opcode .D unit

31 29	28	27 23	22 18	17 13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	1	0	0	0	0	0	1	1	0	0	*s*	*p*
3	1	5	5	5	1											1	1

Opcode map field used...	For operand type...	Unit
src1	uint	.D1, .D2
src2	xuint	
dst	uint	

Description Performs a bitwise logical **AND** operation between *src1* and the bitwise logical inverse of *src2.* The result is placed in *dst.*

Execution

if (cond)	*src1* AND ~*src2* → *dst*
else nop	

Pipeline

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

Instruction Type Single-cycle

Delay Slots 0

See Also AND, OR, XOR

Example `ANDN .L1 A0,A1,A2`

	Before instruction			1 cycle after instruction
A0	1957 21ABh		A0	1957 21ABh
A1	081C 17E6h	F7E3 E819h	A1	081C 17E6h
A2	xxxx xxxxh		A2	1143 2009h

AVG2 *Average, Signed, Packed 16-Bit*

Syntax **AVG2** (.unit) *src1, src2, dst*

unit = .M1 or .M2

Compatibility C64x and C64x+ CPU

Opcode

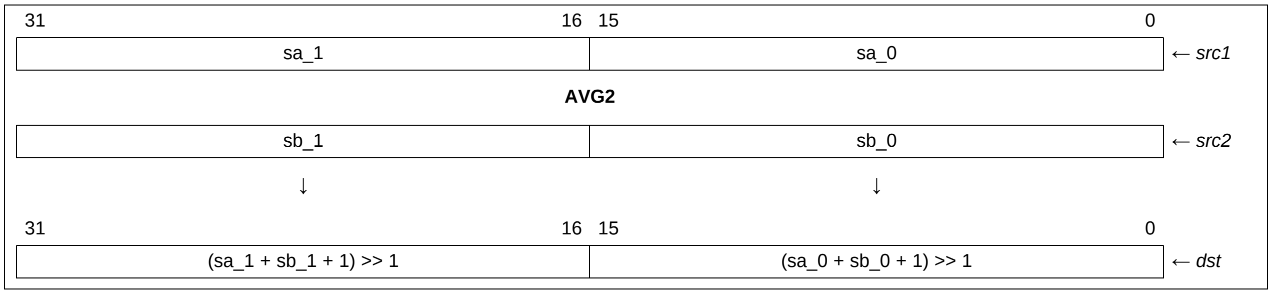
31 29	28	27 23	22 18	17 13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	0	1	0	0	1	1	1	1	0	0	*s*	*p*
3	1	5	5	5	1											1	1

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

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

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

No overflow conditions exist.



Execution

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

Pipeline

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

Instruction Type Two-cycle

Delay Slots 1

See Also AVGU4

Example `AVG2 .M1 A0,A1,A2`

	Before instruction			2 cycles after instruction	
A0	6198 4357h	24984 17239	A0	6198 4357h	
A1	7582 AE15	30082 -20971	A1	7582 AE15h	
A2	xxxx xxxxh		A2	6B8D F8B6h	27533 -1866

AVGU4　*Average, Unsigned, Packed 8-Bit*

Syntax

AVGU4 (.unit) *src1, src2, dst*

unit = .M1 or .M2

Compatibility

C64x and C64x+ CPU

Opcode

31　29	28	27　23	22　18	17　13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	0	1	0	0	1	0	1	1	0	0	*s*	*p*
3	1	5	5	5	1											1	1

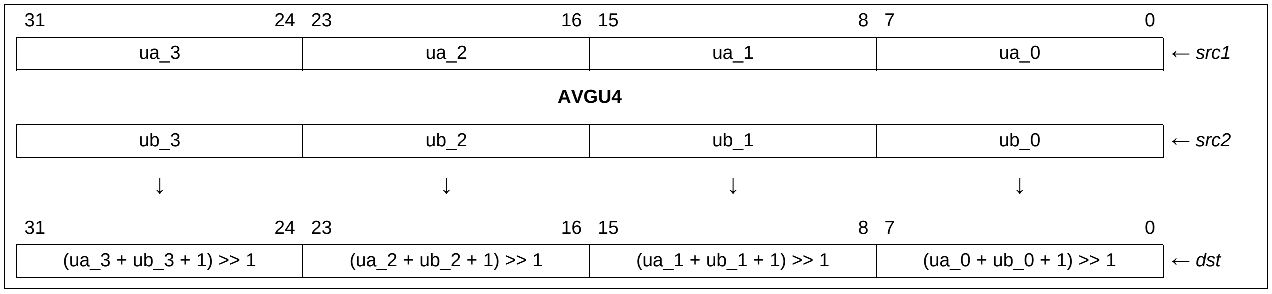
Opcode map field used...	For operand type...	Unit
src1 *src2* *dst*	u4 xu4 u4	.M1, .M2

Description

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

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

No overflow conditions exist.



Execution

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

Pipeline

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

Instruction Type Two-cycle

Delay Slots 1

See Also AVG2

Example `AVGU4 .M1 A0,A1,A2`

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

B *Branch Using a Displacement*

Syntax

B (.unit) label

unit = .S1 or .S2

Compatibility

C62x, C64x, and C64x+ 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.

NOTE:

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 3.4.2 for information on branching into the middle of an execute packet.
4. On the C64x+ CPU, a branch to an execute packet that spans two fetch packets will cause a stall while the second fetch packet is fetched.
5. A relative branch instruction cannot be in the same execute packet as an **ADDKPC** instruction.

Execution

```
if (cond)      (cst21 << 2) + PCE1 → PFC
else nop
```

Pipeline

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

Instruction Type Branch

Delay Slots 5

Example Table 3-14 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 3-14. Program Counter Values for Branch Using a Displacement Example

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

B *Branch Using a Register*

Syntax

B (.unit) *src2*

unit = .S2

Compatibility C62x, C64x, and C64x+ CPU

Opcode

31–29	28	27	26	25	24	23	22–18	17	16	15	14	13	12	11	10	9	8	7	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	1	*p*
3	1						5						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.

NOTE:

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 3.4.2 for information on branching into the middle of an execute packet.
4. On the C64x+ CPU, a branch to an execute packet that spans two fetch packets will cause a stall while the second fetch packet is fetched.

Execution

if (cond)	*src2* → PFC
else nop	

Pipeline

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

Instruction Type Branch

Delay Slots 5

Example Table 3-15 gives the program counter values and actions for the following code example. In this example, the B10 register holds the value 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 3-15. Program Counter Values for Branch Using a Register Example

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, and C64x+ CPU

Opcode

31–29	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
creg	*z*	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0	0	0	1	*p*
3	1																												1

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.

NOTE:

1. This instruction executes on .S2 only. PFC is the program fetch counter.
2. Refer to Chapter 5 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 3.4.2 for information on branching into the middle of an execute packet.
5. On the C64x+ CPU, a branch to an execute packet that spans two fetch packets will cause a stall while the second fetch packet is fetched.

Execution

if (cond)	IRP → PFC
else nop	

Pipeline

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

Instruction Type Branch

Delay Slots 5

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 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 3-16. Program Counter Values for B IRP Instruction Example

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, and C64x+ CPU

Opcode

31–29	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
creg	*z*	0	0	0	0	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	1	1	1	0	0	0	1	*p*
3	1																												1

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.

NOTE:

1. This instruction executes on .S2 only. PFC is program fetch counter.
2. Refer to Chapter 5 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 3.4.2 for information on branching into the middle of an execute packet.
5. On the C64x+ CPU, a branch to an execute packet that spans two fetch packets will cause a stall while the second fetch packet is fetched.

Execution

```
if (cond)       NRP → PFC
else nop
```

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 3-17 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 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 3-17. Program Counter Values for B NRP Instruction Example

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

BDEC *Branch and Decrement*

Syntax

BDEC (.unit) *src, dst*

unit = .S1 or .S2

Compatibility C64x and C64x+ CPU

Opcode

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

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

Description

If the predication and decrement register (*dst*) is positive (greater than or equal to 0), the **BDEC** instruction performs a relative branch and decrements *dst* by 1. The instruction performs the relative branch using a 10-bit signed constant, *scst10*, in *src*. The constant is shifted 2 bits to the left, then added to the address of the first instruction of the fetch packet that contains the **BDEC** instruction (PCE1). The result is placed in the program fetch counter (PFC).

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

The following code:

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

could be replaced by:

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

NOTE:

1. Only one **BDEC** instruction can be executed per cycle. The **BDEC** instruction can be predicated by using any conventional condition register. The conditions are effectively ANDed together. If two branches are in the same execute packet, and if both are taken, behavior is undefined.
2. See Section 3.4.2 for information on branching into the middle of an execute packet.
3. On the C64x+ CPU, a branch to an execute packet that spans two fetch packets will cause a stall while the second fetch packet is fetched.
4. The **BDEC** instruction cannot be in the same execute packet as an **ADDKPC** instruction.

Execution

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

Pipeline

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

Instruction Type Branch

Delay Slots 5

Examples

Example 1

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

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

Example 2

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

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

BITC4 *Bit Count, Packed 8-Bit*

Syntax **BITC4** (.unit) *src2, dst*

unit = .M1 or .M2

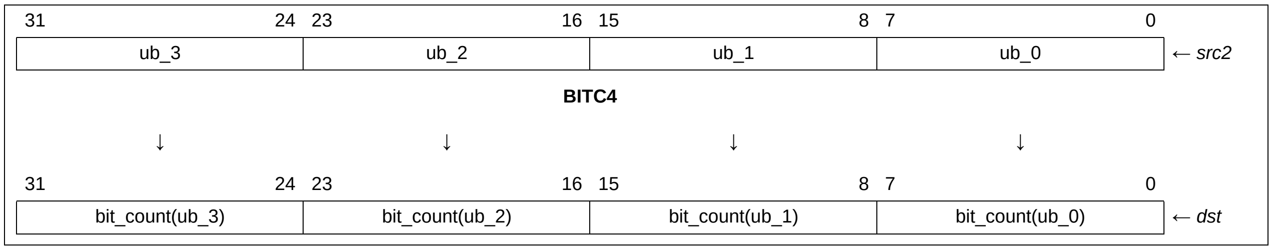
Compatibility C64x and C64x+ CPU

Opcode

31 29	28	27 23	22 18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	1	1	1	1	0	x	0	0	0	0	1	1	1	1	0	0	*s*	*p*
3	1	5	5						1											1	1

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

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



Execution

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

Pipeline

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

Instruction Type Two-cycle

Delay Slots 1

Example `BITC4 .M1 A1,A2`

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

BITR *Bit Reverse*

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

unit = .M1 or .M2

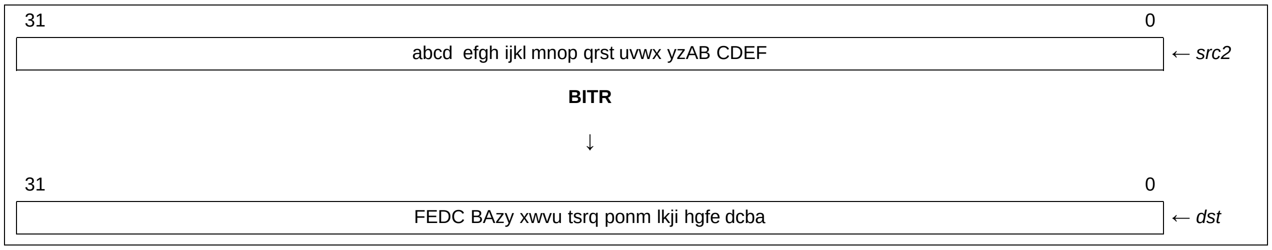
Compatibility C64x and C64x+ CPU

Opcode

31–29	28	27–23	22–18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	1	1	1	1	1	x	0	0	0	0	1	1	1	1	0	0	*s*	*p*
3	1	5	5						1											1	1

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

Description Implements a bit-reversal function that reverses the order of bits in a 32-bit word. This means that bit 0 of the source becomes bit 31 of the result, bit 1 of the source becomes bit 30 of the result, bit 2 becomes bit 29, and so on.


31
0
abcd efgh ijkl mnop qrst uvwx yzAB CDEF
← src2
BITR
↓
31
0
FEDC BAzy xwvu tsrq ponm lkji hgfe dcba
← dst

Execution

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

Pipeline

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

Instruction Type Two-cycle

Delay Slots 1

Example `BITR .M2 B4,B5`

	Before instruction		2 cycles after instruction
B4	A6E2 C179h	B4	A6E2 C179h
B5	xxxx xxxxh	B5	9E83 4765h

BNOP *Branch Using a Displacement With NOP*

Syntax

BNOP (.unit) *src2, src1*

unit = .S1 or .S2 (C64x and C64x+ CPU)

unit = .S1, .S2, or none (C64x+ CPU only)

Compatibility C64x and C64x+ CPU

Compact Instruction Format

Unit	Opcode Format	Figure
.S	Sbs7	Figure F-16
	Sbu8	Figure F-17
	Sbs7c	Figure F-19
	Sbu8c	Figure F-20
	Sx1b	Figure F-31

Opcode

31–29	28	27–16	15–13	12	11	10	9	8	7	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

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

Description

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

The 3-bit unsigned constant specified in *src1* gives the number of delay slot **NOP** instructions to be inserted, from 0 to 7. With *src1* = 0, no **NOP** cycles are inserted.

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

The following code:

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

could be replaced by:

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

NOTE:

1. **BNOP** instructions may be predicated. The predication condition controls whether or not the branch is taken, but does not affect the insertion of **NOP**s. **BNOP** always inserts the number of **NOP**s specified by N, regardless of the predication condition.
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 3.4.2 for information on branching into the middle of an execute packet.
4. On the C64x+ CPU, a branch to an execute packet that spans two fetch packets will cause a stall while the second fetch packet is fetched.

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

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

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

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

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

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

The **BNOP** instruction cannot be paired with any other multicycle **NOP** instruction in the same execute packet. Instructions that generate a multicycle **NOP** are: **IDLE**, **ADDKPC**, **CALLP**, and the multicycle **NOP**.

For the C64x+ CPU: The **BNOP** instruction does not require the use of the .S unit. If no unit is specified, then it may be scheduled in parallel with instructions executing on both the .S1 and .S2 units. If either the .S1 or .S2 unit is specified for **BNOP**, then the .S unit specified is not available for another instruction in the same execute packet. This is enforced by the assembler.

For the C64x+ CPU: It is possible to branch into the middle of a 32-bit instruction. The only case that will be detected and result in an exception is when the 32-bit instruction is contained in a compact header-based fetch packet. The header cannot be the target of a branch instruction. In the event that the header is the target of a branch, an exception will be raised.

Execution (if instruction is within compact instruction fetch packet, C64x+ CPU only)

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

Execution (if instruction is not within compact instruction fetch packet)

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

Pipeline

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

Instruction Type Branch

Delay Slots 5

See Also ADDKPC, B, NOP

Example `BNOP .S1 30h,2`

	Before instruction		After branch has been taken
PCE1	0100 0500h		
PC	xxxx xxxxh	PC	0100 1100h

BNOP *Branch Using a Register With NOP*

Syntax **BNOP** (.unit) *src2, src1*

unit = .S2

Compatibility C64x and C64x+ CPU

Opcode

31 29	28	27	26	25	24	23	22 18	17	16	15 13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	0	0	0	0	1	*src2*	0	0	*src1*	x	0	0	1	1	0	1	1	0	0	0	1	*p*
3	1						5			3	1												1

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

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

For branch targets residing in compact header-based fetch packets(C64x+ CPU only, see Section 3.9 for more information), the 31 most-significant bits of the register are used to determine the branch target. For branch targets not residing in compact header-based fetch packets, the 30 most-significant bits of the register are used to determine the branch target.

The 3-bit unsigned constant specified in *src1* gives the number of delay slots **NOP** instructions to be inserted, from 0 to 7. With *src1* = 0, no NOP cycles are inserted.

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

The following code:

```
B     .S2 B3
NOP   N
```

could be replaced by:

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

NOTE:

1. **BNOP** instructions may be predicated. The predication condition controls whether or not the branch is taken, but does not affect the insertion of **NOP**s. **BNOP** always inserts the number of **NOP**s specified by N, regardless of the predication condition.
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 3.4.2 for information on branching into the middle of an execute packet.
4. On the C64x+ CPU, a branch to an execute packet that spans two fetch packets will cause a stall while the second fetch packet is fetched.

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

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

```
     ZERO  .L1  A0
[A0] BNOP  .S2 B3,7    ; branch is not taken and 7 cycles of NOPs are inserted
```

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

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

```
     MVK  .D1  1,A0
[A0] BNOP .S2  B3,7    ; branch is taken and 5 cycles of NOPs are inserted
```

The **BNOP** instruction cannot be paired with any other multicycle **NOP** instruction in the same execute packet. Instructions that generate a multicycle **NOP** are: **IDLE**, **ADDKPC**, **CALLP**, and the multicycle **NOP**.

Execution

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

Pipeline

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

Instruction Type Branch

Delay Slots 5

See Also ADDKPC, B, NOP

Example `BNOP .S2 A5,2`

	Before instruction		After branch has been taken
PCE1	0010 0000h		
PC	xxxx xxxxh	PC	0100 F000h
A5	0100 F000h	A5	0100 F000h

BPOS *Branch Positive*

Syntax

BPOS (.unit) *src, dst*

unit = .S1 or .S2

Compatibility C64x and C64x+ CPU

Opcode

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

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

Description

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

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

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

NOTE:

1. Only one **BPOS** instruction can be executed per cycle. The **BPOS** instruction can be predicated by using any conventional condition register. The conditions are effectively ANDed together. If two branches are in the same execute packet, and if both are taken, behavior is undefined.
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 3.4.2 for information on branching into the middle of an execute packet.
4. On the C64x+ CPU, a branch to an execute packet that spans two fetch packets will cause a stall while the second fetch packet is fetched.
5. The **BPOS** instruction cannot be in the same execute packet as an **ADDKPC** instruction.

Execution

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

Pipeline

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

Instruction Type Branch

Delay Slots 5

Example `BPOS .S1 200h,A10`

	Before instruction		After branch has been taken
PCE1	0010 0000h		
PC	xxxx xxxxh	PC	0100 0800h
A10	0000 000Ah	A10	0000 000Ah

CALLP *Call Using a Displacement*

Syntax **CALLP** (.unit) label, A3/B3

unit = .S1 or .S2

Compatibility C64x+ CPU

Compact Instruction Format

Unit	Opcode Format	Figure
.S	Scs10	Figure F-18

Opcode

31	30	29	28	27 … 7	6	5	4	3	2	1	0
0	0	0	1	*cst21*	0	0	1	0	0	*s*	*p*
				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

The address of the execute packet immediately following the execute packet containing the **CALLP** instruction is placed in A3, if the S1 unit is used; or in B3, if the S2 unit is used. This write occurs in E1. An implied **NOP 5** is inserted into the instruction pipeline occupying E2-E6.

Since this branch is taken unconditionally, it cannot be placed in the same execute packet as another branch. Additionally, no other branches should be pending when the **CALLP** instruction is executed.

CALLP, like other relative branch instructions, cannot have an **ADDKPC** instruction in the same execute packet with it.

NOTE:

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. retPC represents the address of the first instruction of the execute packet in the DC stage of the pipeline.
2. The execute packets in the delay slots of a branch cannot be interrupted. This is true regardless of whether the branch is taken.

Execution

```
(cst21 << 2) + PCE1 → PFC
if (unit = S2), retPC → B3
else if (unit = S1), retPC → A3
nop 5
```

Pipeline

		Target Instruction					
Pipeline Stage	**E1**	**PS**	**PW**	**PR**	**DP**	**DC**	**E1**
Read							
Written	A3/B3						
Branch taken							✓
Unit in use	.S						

Instruction Type Branch

Delay Slots 5

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, and C64x+ CPU

Compact Instruction Format

Unit	Opcode Format	Figure
.S	Sc5	Figure F-26

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	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*	uint ucst5 ucst5 uint	.S1, .S2

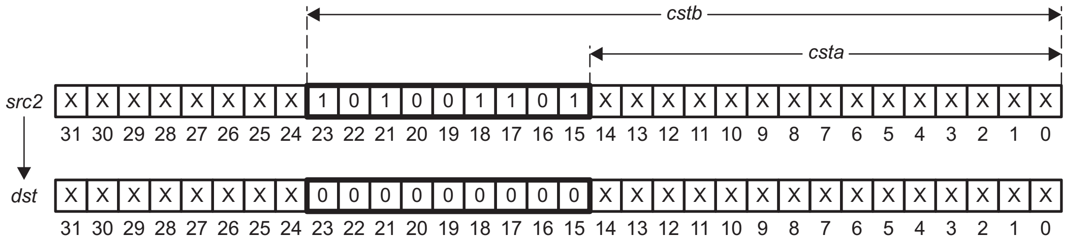
Opcode Register form

31 29	28	27 23	22 18	17 13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	1	1	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*	xuint uint uint	.S1, .S2

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* → *dst*
else nop	

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

if (cond)	*src2* clear *src1* $_{9..5}$, *src1* $_{4..0}$ → *dst*
else nop	

Pipeline

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

Instruction Type Single-cycle

Delay Slots 0

See Also SET

Examples

Example 1

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

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

Example 2

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

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

CMPEQ *Compare for Equality, Signed Integer*

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

unit = .L1 or .L2

Compatibility C62x, C64x, and C64x+ CPU

Compact Instruction Format

Unit	Opcode Format	Figure
.L	L2c	Figure D-7
	Lx3c	Figure D-9

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 CMPEQ2, CMPEQ4

Examples

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

CMPEQ2 *Compare for Equality, Packed 16-Bit*

Syntax **CMPEQ2** (.unit) *src1, src2, dst*

unit = .S1 or .S2

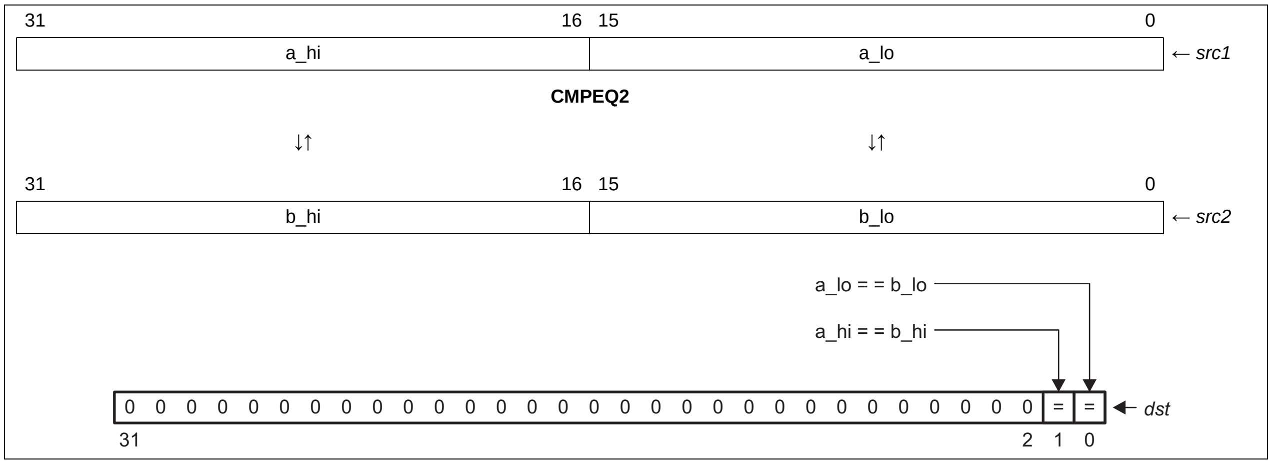
Compatibility C64x and C64x+ CPU

Opcode

31–29	28	27–23	22–18	17–13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	0	1	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
src1	s2	.S1, .S2
src2	xs2	
dst	bv2	

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



Execution

```
if (cond)      {
               if (lsb16(src1) == lsb16(src2)), 1 → dst 0
                    else 0 → dst 0;
               if (msb16(src1) == msb16(src2)), 1 → dst 1
                    else 0 → dst 1
               }
else nop
```

Pipeline

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

Instruction Type Single-cycle

Delay Slots 0

See Also CMPEQ, CMPEQ4, CMPGT2, XPND2

Examples

Example 1

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

	Before instruction		1 cycle after instruction	
A3	1105 6E30h	A3	1105 6E30h	
A4	1105 6980h	A4	1105 6980h	
A5	xxxx xxxxh	A5	0000 0002h	true, false

Example 2

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

	Before instruction		1 cycle after instruction	
B2	F23A 3789h	B2	F23A 3789h	
B8	04B8 3789h	B8	04B8 3789h	
B15	xxxx xxxxh	B15	0000 0001h	false, true

Example 3

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

	Before instruction		1 cycle after instruction	
B2	01B6 2451h	B2	01B6 2451h	
B8	01B6 2451h	B8	01B6 2451h	
B15	xxxx xxxxh	B15	0000 0003h	true, true

CMPEQ4 *Compare for Equality, Packed 8-Bit*

Syntax **CMPEQ4** (.unit) *src1, src2, dst*

unit = .S1 or .S2

Compatibility C64x and C64x+ CPU

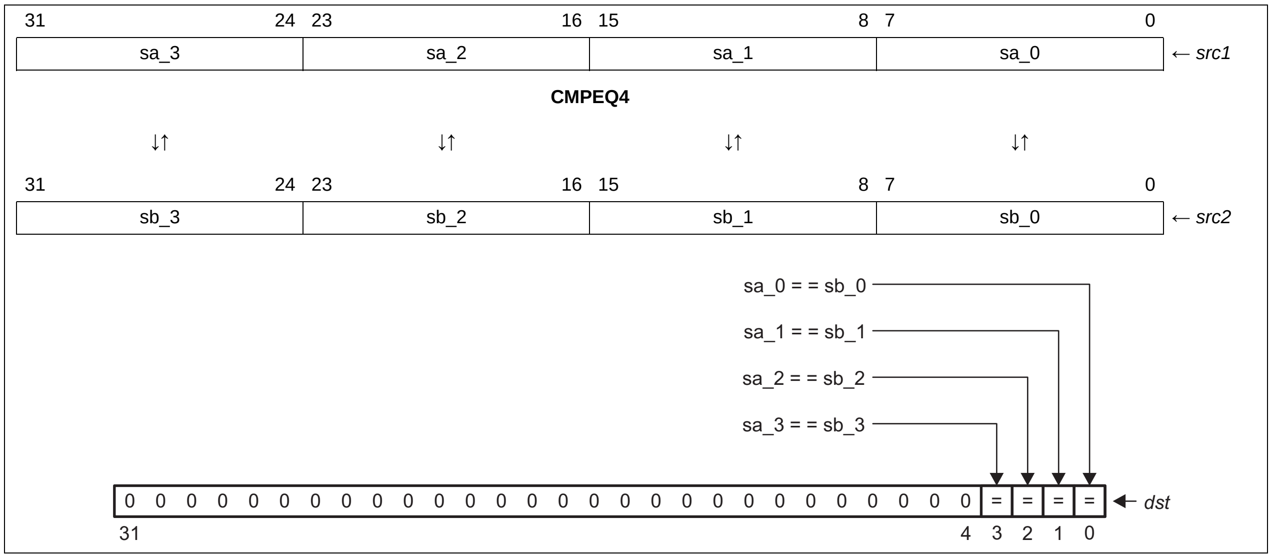
Opcode

31–29	28	27–23	22–18	17–13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	0	1	1	1	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	s4	.S1, .S2
src2	xs4	
dst	bv4	

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

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


31 24 23 16 15 8 7 0
sa_3 sa_2 sa_1 sa_0 ← src1
CMPEQ4
31 24 23 16 15 8 7 0
sb_3 sb_2 sb_1 sb_0 ← src2
sa_0 = = sb_0
sa_1 = = sb_1
sa_2 = = sb_2
sa_3 = = sb_3
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 = = = = ← dst
31 4 3 2 1 0

Execution

```
if (cond)     {
              if (sbyte0(src1) == sbyte0(src2)), 1 → dst 0
                  else 0 → dst 0;
              if (sbyte1(src1) == sbyte1(src2)), 1 → dst 1
                  else 0 → dst 1;
              if (sbyte2(src1) == sbyte2(src2)), 1 → dst 2
                  else 0 → dst 2;
              if (sbyte3(src1) == sbyte3(src2)), 1 → dst 3
                  else 0 → dst 3
              }
else nop
```

Pipeline

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

Instruction Type Single-cycle

Delay Slots 0

See Also CMPEQ, CMPEQ2, CMPGTU4, XPND4

Examples

Example 1

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

	Before instruction		1 cycle after instruction	
A3	02 3A 4E 1Ch	A3	02 3A 4E 1Ch	
A4	02 B8 4E 76h	A4	02 B8 4E 76h	
A5	xxxx xxxxh	A5	0000 000Ah	true, false, false, false

Example 2

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

	Before instruction		1 cycle after instruction	
B2	F2 3A 37 89h	B2	F2 3A 37 89h	
B8	04 B8 37 89h	B8	04 B8 37 89h	
B13	xxxx xxxxh	B13	0000 0003h	false, false, true, true

Example 3

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

	Before instruction		1 cycle after instruction	
B2	01 B6 24 51h	B2	01 B6 24 51h	
B8	05 B6 24 51h	B8	05 B6 24 51h	
B13	xxxx xxxxh	B13	0000 0007h	false, true, true, true

CMPGT *Compare for Greater Than, Signed Integers*

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

unit = .L1 or .L2

Compatibility C62x, C64x, and C64x+ CPU

Compact Instruction Format

Unit	Opcode Format	Figure
.L	L2c	Figure D-7
	Lx1c	Figure D-10

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 CMPGT2, CMPGTU, CMPGTU4

Examples

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	

CMPGT2 *Compare for Greater Than, Packed 16-Bit*

Syntax **CMPGT2** (.unit) *src1, src2, dst*

unit = .S1 or .S2

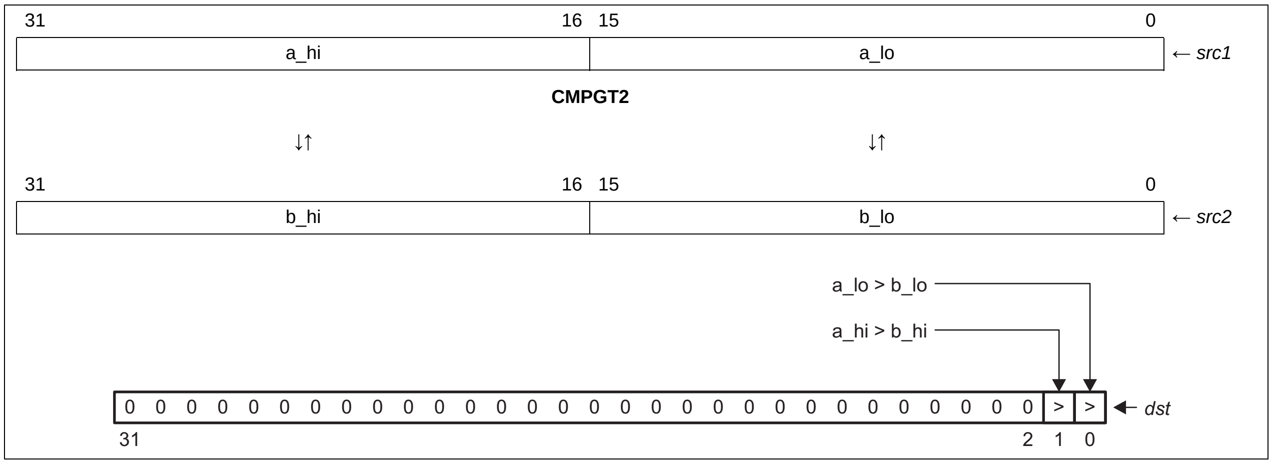
Compatibility C64x and C64x+ CPU

Opcode

31–29	28	27–23	22–18	17–13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	0	1	0	1	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	s2	.S1, .S2
src2	xs2	
dst	bv2	

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



Execution

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

Pipeline

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

Instruction Type Single-cycle

Delay Slots 0

See Also CMPEQ2, CMPGT, CMPGTU, CMPGTU4, CMPLT2, XPND2

Examples

Example 1

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

	Before instruction			1 cycle after instruction	
A3	1105 6E30h	4357 28208	A3	1105 6E30h	
A4	1105 6980h	4357 27008	A4	1105 6980h	
A5	xxxx xxxxh		A5	0000 0001h	false, true

Example 2

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

	Before instruction			1 cycle after instruction	
B2	F348 3789h	-3526 14217	B2	F348 3789h	
B8	04B8 4975h	1208 18805	B8	04B8 4975h	
B15	xxxx xxxxh		B15	0000 0000h	false, false

Example 3

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

	Before instruction			1 cycle after instruction	
B2	01A6 2451h	422 9297	B2	01A6 2451h	
B8	0124 A051h	292 -24495	B8	0124 A051h	
B15	xxxx xxxxh		B15	0000 0003h	true, true

CMPGTU *Compare for Greater Than, Unsigned Integers*

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

unit = .L1 or .L2

Compatibility C62x, C64x, and C64x+ CPU

Compact Instruction Format

Unit	Opcode Format	Figure
.L	L2c	Figure D-7
	Lx1c	Figure D-10

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 or ucst5[1] ulong uint	.L1, .L2	100 1100

(1) On the C62x CPU, only the four LSBs (*ucst4*) are valid in the 5-bit *src1* field. On the C64x and C64x+ CPU, all five bits (*ucst5*) are valid in the 5-bit *src1* field.

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*.

On the C62x CPU: When the *ucst4* operand is used, only the four LSBs are valid in the 5-bit *src1* field; if the MSB of the *src1* field is nonzero, the result is invalid.

On the C64x and C64x+ CPU: When the *ucst5* operand is used, all five bits are valid in the 5-bit *src1* field.

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, CMPGT2, CMPGTU4

Examples

Example 1

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

	Before instruction			1 cycle after instruction	
A1	0000 0128h	296[1]	A1	0000 0128h	
A2	FFFF FFDEh	4,294,967,262[1]	A2	FFFF FFDEh	
A3	xxxx xxxxh		A3	0000 0000h	false

[1] Unsigned 32-bit integer

Example 2

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

	Before instruction			1 cycle after instruction	
A1	0000 0005h	5[1]	A1	0000 0005h	
A2	xxxx xxxxh		A2	0000 0001h	true

[1] 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[1]	A3:A2	0000 0000h	0000 000Ah	
A4	xxxx xxxxh			A4	0000 0001h	true	

[1] Unsigned 40-bit (long) integer

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

Syntax

CMPGTU4 (.unit) *src1, src2, dst*

unit = .S1 or .S2

Compatibility

C64x and C64x+ CPU

Opcode

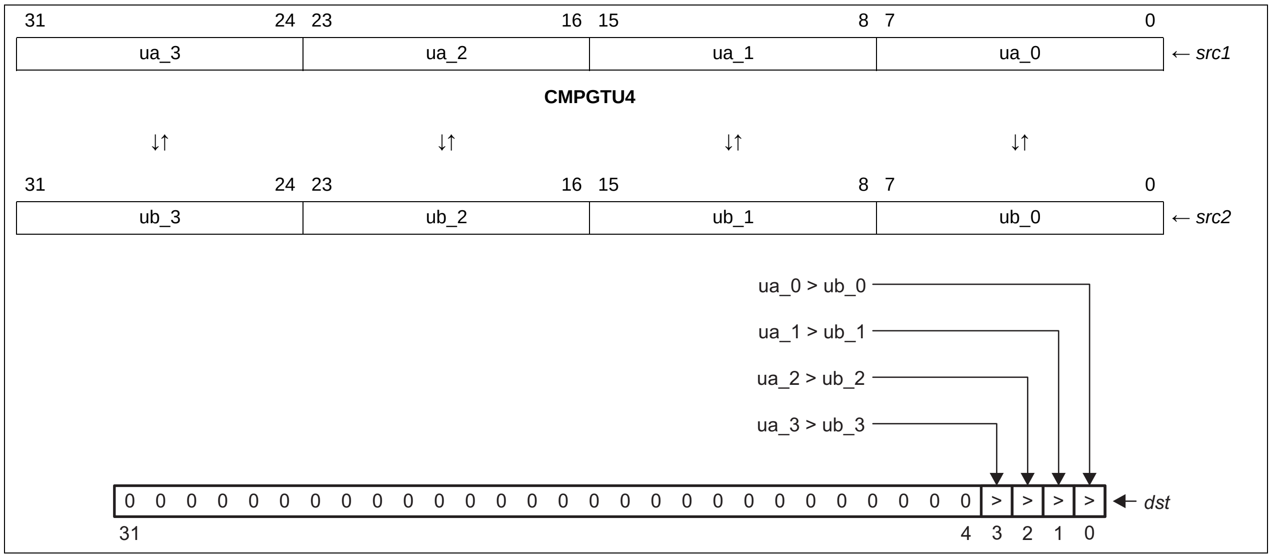
31–29	28	27–23	22–18	17–13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	0	1	0	1	0	1	1	0	0	0	*s*	*p*
3	1	5	5	5	1											1	1

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

Description

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

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



Execution

```
if (cond)    {
             if (ubyte0(src1) > ubyte0(src2)), 1 → dst 0
                 else 0 → dst 0;
             if (ubyte1(src1) > ubyte1(src2)), 1 → dst 1
                 else 0 → dst 1;
             if (ubyte2(src1) > ubyte2(src2)), 1 → dst 2
                 else 0 → dst 2;
             if (ubyte3(src1) > ubyte3(src2)), 1 → dst 3
                 else 0 → dst 3
             }
else nop
```

Pipeline

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

Instruction Type Single-cycle

Delay Slots 0

See Also CMPEQ4, CMPGT, CMPGT2, CMPGTU, CMPLT, XPND4

Examples

Example 1

`CMPGTU4 .S1 A3,A4,A5`

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

Example 2

`CMPGTU4 .S2 B2,B8,B13`

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

Example 3

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

	Before instruction			**1 cycle after instruction**	
B2	12 33 9D 51h	18 51 157 81	B2	12 33 9D 51h	
B8	75 67 24 C5h	117 103 36 197	B8	75 67 24 C5h	
B13	xxxx xxxxh		B13	0000 0002h	false, false, true, false

CMPLT *Compare for Less Than, Signed Integers*

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

unit = .L1 or .L2

Compatibility C62x, C64x, and C64x+ CPU

Compact Instruction Format

Unit	Opcode Format	Figure
.L	L2c	Figure D-7
	Lx1c	Figure D-10

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 → 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 CMPLT2, CMPLTU, CMPLTU4

Examples **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

CMPLT2 *Compare for Less Than, Packed 16-Bit*

Syntax

CMPLT2 (.unit) *src2, src1, dst*

unit = .S1 or .S2

Compatibility C64x and C64x+ CPU

Opcode

31–29	28	27–23	22–18	17–13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	0	1	0	1	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*	s2 xs2 bv2	.S1, .S2

Description

The **CMPLT2** instruction is a pseudo-operation used to perform less-than comparisons on signed, packed 16-bit data. Each signed 16-bit value in *src2* is compared against the corresponding signed 16-bit value in *src1*, returning a 1 if *src2* is less than *src1* or returning a 0 if it is not less than. The comparison results are packed into the two least-significant bits of *dst*. The result for the lower pair of values is placed in bit 0, and the results for the upper pair of values are placed in bit 1. The remaining bits of *dst* are cleared to 0.

The assembler uses the operation **CMPGT2** (.unit) *src1, src2, dst* to perform this task (see CMPGT2).

Execution

```
if (cond)    {
             if (lsb16(src2) < lsb16(src1)), 1 → dst 0
                  else 0 → dst 0;
             if (msb16(src2) < msb16(src1)), 1 → dst 1
                  else 0 → dst 1
             }
else nop
```

Pipeline

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

Instruction Type Single-cycle

Delay Slots 0

See Also CMPEQ2, CMPGT2, CMPLT, CMPLTU, CMPLTU4, XPND2

Examples

Example 1

```
CMPLT2 .S1 A4,A3,A5; assembler treats as CMPGT2 A3,A4,A5
```

	Before instruction			1 cycle after instruction	
A3	1105 6E30h	4357 28208	A3	1105 6E30h	
A4	1105 6980h	4357 27008	A4	1105 6980h	
A5	xxxx xxxxh		A5	0000 0001h	false, true

Example 2

```
CMPLT2 .S2 B8,B2,B15; assembler treats as CMPGT2 B2,B8,B15
```

	Before instruction			1 cycle after instruction	
B2	F23A 3789h	-3526 14217	B2	F23A 3789h	
B8	04B8 4975h	1208 18805	B8	04B8 4975h	
B15	xxxx xxxxh		B15	0000 0000h	false, false

Example 3

```
CMPLT2 .S2 B8,B2,B12; assembler treats as CMPGT2 B2,B8,B15
```

	Before instruction			1 cycle after instruction	
B2	01A6 2451h	422 9297	B2	01A6 2451h	
B8	0124 A051h	292 -24495	B8	0124 A051h	
B12	xxxx xxxxh		B12	0000 0003h	true, true

CMPLTU *Compare for Less Than, Unsigned Integers*

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

unit = .L1 or .L2

Compatibility C62x, C64x, and C64x+ CPU

Compact Instruction Format

Unit	Opcode Format	Figure
.L	L2c	Figure D-7
	Lx1c	Figure D-10

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 or ucst5[1] ulong uint	.L1, .L2	101 1100

[1] On the C62x CPU, only the four LSBs (*ucst4*) are valid in the 5-bit *src1* field. On the C64x and C64x+ CPU, all five bits (*ucst5*) are valid in the 5-bit *src1* field.

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*.

On the C62x CPU: When the *ucst4* operand is used, only the four LSBs are valid in the 5-bit *src1* field; if the MSB of the *src1* field is nonzero, the result is invalid.

On the C64x and C64x+ CPU: When the *ucst5* operand is used, all five bits are valid in the 5-bit *src1* field.

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 CMPLT, CMPLT2, CMPLTU4

Examples

Example 1

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

	Before instruction			1 cycle after instruction	
A1	0000 289Ah	10,394[1]	A1	0000 289Ah	
A2	FFFF F35Eh	4,294,964,062[1]	A2	FFFF F35Eh	
A3	xxxx xxxxh		A3	0000 0001h	true

[1] Unsigned 32-bit integer

Example 2

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

	Before instruction			1 cycle after instruction	
A1	0000 000Fh	15[1]	A1	0000 000Fh	
A2	xxxx xxxxh		A2	0000 0001h	true

[1] Unsigned 32-bit integer

Example 3

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

	Before instruction				1 cycle after instruction	
A1	003B 8260h		3,900,000[1]	A1	003B 8260h	
A2	xxxx xxxxh			A2	0000 0000h	false
A5:A4	0000 0000h	003A 0002h	3,801,090[2]	A5:A4	0000 0000h	003A 0002h

[1] Unsigned 32-bit integer
[2] Unsigned 40-bit (long) integer

CMPLTU4 *Compare for Less Than, Unsigned, Packed 8-Bit*

Syntax **CMPLTU4** (.unit) *src2, src1, dst*

unit = .S1 or .S2

Compatibility C64x and C64x+ CPU

Opcode

31 29	28	27 23	22 18	17 13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	0	1	0	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
src1	u4	.S1, .S2
src2	xu4	
dst	bv4	

Description The **CMPLTU4** instruction is a pseudo-operation that performs less-than comparisons on packed 8-bit data. Each unsigned 8-bit value in *src2* is compared against the corresponding unsigned 8-bit value in *src1*, returning a 1 if the byte in *src2* is less than the corresponding byte in *src1* or a 0 it if is not less than. The comparison results are packed into the four least-significant bits of *dst*.

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

The assembler uses the operation **CMPGTU4** (.unit) *src1, src2, dst* to perform this task (see CMPGTU4).

Execution

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

Pipeline

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

Instruction Type Single-cycle

Delay Slots 0

See Also CMPEQ4, CMPGT, CMPLT, CMPLT2, CMPLTU, XPND4

Examples

Example 1

```
CMPLTU4 .S1 A4,A3,A5; assembler treats as CMPGTU4 A3,A4,A5
```

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

Example 2

```
CMPLTU4 .S2 B8,B2,B13; assembler treats as CMPGTU4 B2,B8,B13
```

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

Example 3

```
CMPLTU4 .S2 B8,B2,B13; assembler treats as CMPGTU4 B2,B8,B13
```

	Before instruction			1 cycle after instruction	
B2	12 33 9D 51h	18 51 157 81	B2	12 33 9D 51h	
B8	75 67 24 C5h	117 103 36 197	B8	75 67 24 C5h	
B13	xx xx xx xxh		B13	0000 0002h	false, false, true, false

CMPY *Complex Multiply Two Pairs, Signed, Packed 16-Bit*

Syntax **CMPY** (.unit) *src1, src2, dst_o:dst_e*

unit = .M1 or .M2

Compatibility C64x+ CPU

Opcode

31	30	29	28	27–23	22–18	17–13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	*dst*	*src2*	*src1*	x	0	0	1	0	1	0	1	1	0	0	*s*	*p*
				5	5	5	1											1	1

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

Description Returns two dot-products between two pairs of signed, packed 16-bit values. The values in *src1* and *src2* are treated as signed, packed 16-bit quantities. The signed results are written to a 64-bit register pair.

The product of the lower halfwords of *src1* and *src2* is subtracted from the product of the upper halfwords of *src1* and *src2*. The result is written to *dst_o*.

The product of the upper halfword of *src1* and the lower halfword of *src2* is added to the product of the lower halfword of *src1* and the upper halfword of *src2*. The result is written to *dst_e*.

If the result saturates, the M1 or M2 bit in SSR and the SAT bit in CSR are written one cycle after the result is written to *dst_e.*

This instruction executes unconditionally.

NOTE: In the overflow case, where all four halfwords in *src1* and *src2* are 8000h, the saturation value 7FFF FFFFh is written into the 32-bit *dst_e* register.

Execution

sat((lsb16(*src1*) × msb16(*src2*)) + (msb16(*src1*) × lsb16(*src2*))) → *dst_e*
(msb16(*src1*) × msb16(*src2*)) - (lsb16(*src1*) × lsb16(*src2*)) → *dst_o*

Instruction Type Four-cycle

Delay Slots 3

See Also CMPYR, CMPYR1, DOTP2, DOTPN2

Examples

Example 1

```
CMPY .M1 A0,A1,A3:A2
```

	Before instruction		4 cycles after instruction [(1)]
A0	0008 0004h	A2	0000 0034h
A1	0009 0002h	A3	0000 0040h

(1) CSR.SAT and SSR.M1 unchanged by operation

Example 2

```
CMPY .M2X B0,A1,B3:B2
```

	Before instruction		4 cycles after instruction [(1)]
B0	7FFF 7FFFh	B2	FFFF 8001h
A1	7FFF 8000h	B3	7FFE 8001h

(1) CSR.SAT and SSR.M2 unchanged by operation

Example 3

```
CMPY .M1 A0,A1,A3:A2
```

	Before instruction		4 cycles after instruction [(1)]
A0	8000 8000h	A2	7FFF FFFFh
A1	8000 8000h	A3	0000 0000h

(1) CSR.SAT and SSR.M1 unchanged by operation

CMPYR *Complex Multiply Two Pairs, Signed, Packed 16-Bit With Rounding*

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

unit = .M1 or .M2

Compatibility C64x+ CPU

Opcode

31	30	29	28	27–23	22–18	17–13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	*dst*	*src2*	*src1*	x	0	0	1	0	1	1	1	1	0	0	*s*	*p*
				5	5	5	1											1	1

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

Description Performs two dot-products between two pairs of signed, packed 16-bit values. The values in *src1* and *src2* are treated as signed, packed 16-bit quantities. The signed results are rounded with saturation, shifted, packed and written to a 32-bit register.

The product of the lower halfwords of *src1* and *src2* is subtracted from the product of the upper halfwords of *src1* and *src2*. The result is rounded by adding 2^{15} to it. The 16 most-significant bits of the rounded value are written to the upper half of *dst*.

The product of the upper halfword of *src1* and the lower halfword of *src2* is added to the product of the lower halfword of *src1* and the upper halfword of *src2*. The result is rounded by adding 2^{15} to it. The 16 most-significant bits of the rounded value are written to the lower half of *dst.*

If either result saturates, the M1 or M2 bit in SSR and the SAT bit in CSR are written one cycle after the result is written to *dst*.

This instruction executes unconditionally.

Execution

```
sat((lsb16(src1) × msb16(src2)) + (msb16(src1) × lsb16(src2))) → tmp_e
msb16(sat(tmp_e + 0000 8000h)) → lsb16(dst)
sat((msb16(src1) × msb16(src2)) - (lsb16(src1) × lsb16(src2))) → tmp_o
msb16(sat(tmp_o + 0000 8000h)) → msb16(dst)
```

Instruction Type Four-cycle

Delay Slots 3

See Also CMPY, CMPYR1, DOTP2, DOTPN2

Examples

Example 1

```
CMPYR .M1 A0,A1,A2
```

	Before instruction		4 cycles after instruction [(1)]
A0	0800 0400h	A2	0040 0034h
A1	0900 0200h		

(1) CSR.SAT and SSR.M1 unchanged by operation

Example 2

```
CMPYR .M2X B0,A1,B2
```

	Before instruction		4 cycles after instruction [(1)]
B0	7FFF 7FFFh	B2	7FFF 0000h
A1	7FFF 8000h		

(1) CSR.SAT and SSR.M2 unchanged by operation

Example 3

```
CMPYR .M1 A0,A1,A2
```

	Before instruction		4 cycles after instruction
A0	8000 8000h	A2	0000 7FFFh
A1	8000 8000h		
CSR	0001 0100h	CSR[(1)]	0001 0300h
SSR	0000 0000h	SSR[(1)]	0000 0010h

(1) CSR.SAT and SSR.M1 set to 1, 5 cycles after instruction

Example 4

```
CMPYR .M2 B0,B1,B2
```

	Before instruction		4 cycles after instruction
B0	8000 8000h	B2	0001 7FFFh
B1	8000 8001h		
CSR	0001 0100h	CSR[(1)]	0001 0300h
SSR	0000 0000h	SSR[(1)]	0000 0020h

(1) CSR.SAT and SSR.M2 set to 1, 5 cycles after instruction

CMPYR1 *Complex Multiply Two Pairs, Signed, Packed 16-Bit With Rounding*

Syntax **CMPYR1** (.unit) *src1, src2, dst*

unit = .M1 or .M2

Compatibility C64x+ CPU

Opcode

31	30	29	28	27–23	22–18	17–13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	*dst*	*src2*	*src1*	x	0	0	1	1	0	0	1	1	0	0	*s*	*p*
				5	5	5	1											1	1

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

Description Performs two dot-products between two pairs of signed, packed 16-bit values. The values in *src1* and *src2* are treated as signed, packed 16-bit quantities. The signed results are rounded with saturation to 31 bits, shifted, packed and written to a 32-bit register.

The product of the lower halfwords of *src1* and *src2* is subtracted from the product of the upper halfwords of *src1* and *src2*. The intermediate result is rounded by adding 2^{14} to it. This value is shifted left by 1 with saturation. The 16 most-significant bits of the shifted value are written to the upper half of *dst.*

The product of the upper halfword of *src1* and the lower halfword of *src2* is added to the product of the lower halfword of *src1* and the upper halfword of *src2*. The intermediate result is rounded by adding 2^{14} to it. This value is shifted left by 1 with saturation. The 16 most-significant bits of the shifted value are written to the lower half of *dst.*

If either result saturates in the rounding or shifting process, the M1 or M2 bit in SSR and the SAT bit in CSR are written one cycle after the results are written to *dst*.

This instruction executes unconditionally.

Execution

sat((lsb16(*src1*) × msb16(*src2*)) + (msb16(*src1*) × lsb16(*src2*))) → *tmp_e*
msb16(sat((*tmp_e* + 0000 4000h) << 1)) → lsb16(*dst*)
sat((msb16(*src1*) × msb16(*src2*)) - (lsb16(*src1*) × lsb16(*src2*))) → *tmp_o*
msb16(sat((*tmp_e* + 0000 4000h) << 1)) → msb16(*dst*)

Instruction Type Four-cycle

Delay Slots 3

See Also CMPY, CMPYR, DOTP2, DOTPN2

Examples

Example 1

```
CMPYR1 .M1 A0,A1,A2
```

	Before instruction		4 cycles after instruction [(1)]
A0	0800 0400h	A2	0080 0068h
A1	0900 0200h		

(1) CSR.SAT and SSR.M1 unchanged by operation

Example 2

```
CMPYR1 .M2X B0,A1,B2
```

	Before instruction		4 cycles after instruction
B0	7FFF 7FFFh	B2	7FFF FFFFh
A1	7FFF 8000h		
CSR	0001 0100h	CSR[(1)]	0001 0300h
SSR	0000 0000h	SSR[(1)]	0000 0020h

(1) CSR.SAT and SSR.M2 set to 1, 5 cycles after instruction

Example 3

```
CMPYR1 .M1 A0,A1,A2
```

	Before instruction		4 cycles after instruction
A0	8000 8000h	A2	0000 7FFFh
A1	8000 8000h		
CSR	0001 0100h	CSR[(1)]	0001 0300h
SSR	0000 0000h	SSR[(1)]	0000 0010h

(1) CSR.SAT and SSR.M1 set to 1, 5 cycles after instruction

Example 4

```
CMPYR1 .M2 B0,B1,B2
```

	Before instruction		4 cycles after instruction
B0	C000 C000h	B2	0001 7FFFh
B1	8000 8001h		
CSR	0001 0100h	CSR[(1)]	0001 0300h
SSR	0000 0000h	SSR[(1)]	0000 0020h

(1) CSR.SAT and SSR.M2 set to 1, 5 cycles after instruction

CMTL *Commit Store Linked Word to Memory Conditionally*

Syntax

CMTL (.unit) **baseR, dst*

unit = .D2

Compatibility

C64x+ CPU

NOTE: The atomic operations are not supported on all C64x+ devices, see your device-specific data manual for more information.

Opcode

31 29	28	27 23	22 18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*baseR*	0	0	0	0	0	0	0	1	1	1	0	1	0	0	0	0	1	*p*
3	1	5	5																		1

Opcode map field used...	For operand type...	Unit
baseR	address	.D2
dst	int	

Description

The **CMTL** instruction performs a read of the 32-bit word in memory at the address specified by *baseR*. For linked-operation aware systems, the read request is interpreted as a request to write the corresponding linked-stored 32-bit word (previously buffered by an SL operation) to memory conditionally. The decision to perform the write to memory is based on whether the link valid flag is set and whether the previously buffered address is equal to the address specified by *baseR*. If the result is written, a value of 1 is returned as the 32-bit data for the read operation; otherwise a value of 0 is returned. The return value is written to *dst*.

When initiating the memory read operation, the CPU signals that this is a commit-linked read operation. Other than this signaling, the operation of the **CMTL** instruction from the CPU perspective is identical to that of LDW **baseR, dst*.

See Chapter 9 for more details.

Execution

```
if (cond)     mem → dst
              signal commit-linked operation
else nop
```

Instruction Type Load

Delay Slots 4

See Also LL, SL

DDOTP4 *Double Dot Product, Signed, Packed 16-Bit and Signed, Packed 8-Bit*

Syntax **DDOTP4** (.unit) *src1, src2, dst_o:dst_e*

unit = .M1 or .M2

Compatibility C64x+ CPU

Opcode

31	30	29	28	27–23	22–18	17–13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	*dst*	*src2*	*src1*	x	0	1	1	0	0	0	1	1	0	0	*s*	*p*
				5	5	5	1											1	1

Opcode map field used...	For operand type...	Unit
src1	ds2	.M1, .M2
src2	xs4	
dst	dint	

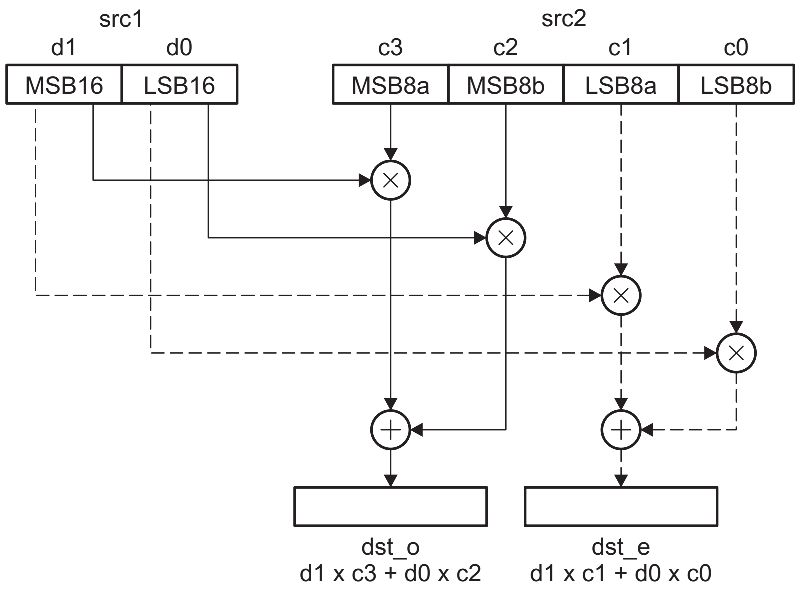
Description Performs two **DOTP2** operations simultaneously.

The lower byte of the lower halfword of *src2* is sign-extended to 16 bits and multiplied by the lower halfword of *src1*. The upper byte of the lower halfword of *src2* is sign-extended to 16 bits and multiplied by the upper halfword of *src1*. The two products are added together and the result is then written to *dst_e*.

The lower byte of the upper halfword of *src2* is sign-extended to 16 bits and multiplied by the lower halfword of *src1*. The upper byte of the upper halfword of *src2* is sign-extended to 16 bits and multiplied by the upper halfword of *src1*. The two products are added together and the result is then written to *dst_o*.

There are no saturation cases possible.

This instruction executes unconditionally.



Execution

(msb16(*src1*) × msb8(lsb16(*src2*))) + (lsb16(*src1*) × lsb8(lsb16(*src2*))) → *dst_e*
(msb16(*src1*) × msb8(msb16(*src2*))) + (lsb16(*src1*) × lsb8(msb16(*src2*))) → *dst_o*

Instruction Type Four-cycle

Delay Slots 3

Examples

Example 1

```
DDOTP4 .M1 A4,A5,A9:A8
```

	Before instruction			4 cycles after instruction	
A4	0005 0003h	5, 3	A8	0000 001Bh	(5 × 3) + (3 × 4) = 27
A5	0102 0304h	1, 2, 3, 4	A9	0000 000Bh	(5 × 1) + (3 × 2) = 11

Example 2

```
DDOTP4 .M1X A4,B5,A9:A8
```

	Before instruction		4 cycles after instruction
A4	8000 8000h	A8	FF81 0000h
B5	8080 7F7Fh	A9	0080 0000h

DDOTPH2 *Double Dot Product, Two Pairs, Signed, Packed 16-Bit*

Syntax **DDOTPH2** (.unit) *src1_o:src1_e, src2, dst_o:dst_e*

unit = .M1 or .M2

Compatibility C64x+ CPU

Opcode

31	30	29	28	27–23	22–18	17–13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	*dst*	*src2*	*src1*	x	0	1	0	1	1	1	1	1	0	0	*s*	*p*
				5	5	5	1											1	1

Opcode map field used...	For operand type...	Unit
src1	ds2	.M1, .M2
src2	xs2	
dst	dint	

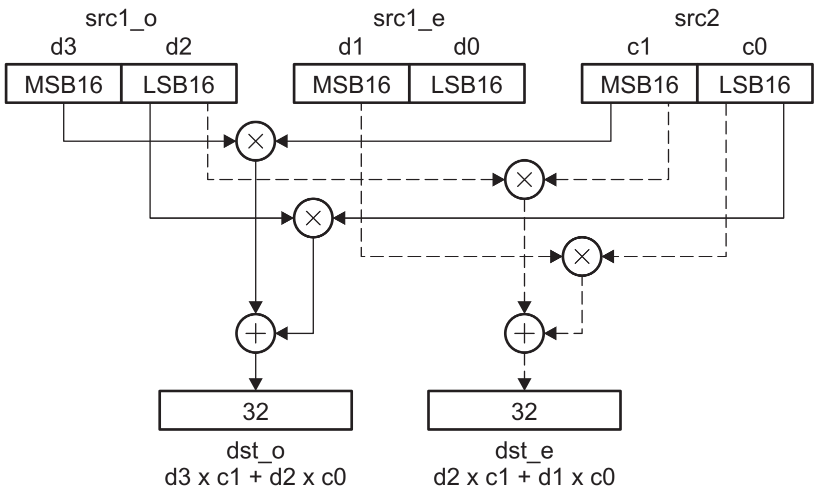
Description Returns two dot-products between two pairs of signed, packed 16-bit values. The values in *src1_e, src1_o*, and *src2* are treated as signed, packed 16-bit quantities. The signed results are written to a 64-bit register pair.

The product of the lower halfwords of *src1_o* and *src2* is added to the product of the upper halfwords of *src1_o* and *src2*. The result is then written to *dst_o*.

The product of the upper halfword of *src2* and the lower halfword of *src1_o* is added to the product of the lower halfword of *src2* and the upper halfword of *src1_e*. The result is then written to *dst_e*.

If either result saturates, the M1 or M2 bit in SSR and the SAT bit in CSR are written one cycle after the results are written to *dst_o:dst_e*.

This instruction executes unconditionally.



Execution

sat((msb16(*src1_o*) × msb16(*src2*)) + (lsb16(*src1_o*) × lsb16(*src2*))) → *dst_o*
sat((lsb16(*src1_o*) × msb16(*src2*)) + (msb16(*src1_e*) × lsb16(*src2*))) → *dst_e*

Instruction Type Four-cycle

Delay Slots 3

See Also DDOTPL2, DDOTPH2R, DDOTPL2R

Examples

Example 1

```
DDOTPH2 .M1 A5:A4,A6,A9:A8
```

	Before instruction			4 cycles after instruction [1]	
A4	0005 0003h	5 , 3	A8	0000 0021h	(4 × 7) + (5 × 1) = 33
A5	0002 0004h	2 , 4	A9	0000 0012h	(2 × 7) + (4 × 1) = 18
A6	0007 0001h	7 , 1			

(1) CSR.SAT and SSR.M1 unchanged by operation

Example 2

```
DDOTPH2 .M1 A5:A4,A6,A9:A8
```

	Before instruction		4 cycles after instruction
A4	8000 5678h	A8	7FFF FFFFh
A5	1234 8000h	A9	36E6 0000h
A6	8000 8000h		
CSR	0001 0100h	CSR[1]	0001 0300h
SSR	0000 0000h	SSR[1]	0000 0010h

(1) CSR.SAT and SSR.M1 set to 1, 5 cycles after instruction

Example 3

```
DDOTPH2 .M2X B5:B4,A6,B9:B8
```

	Before instruction		4 cycles after instruction [1]
B4	46B4 16BAh	B8	F41B 4AFFh
B5	BBAE D169h	B9	F3B4 FAADh
A6	340B F73Bh		

(1) CSR.SAT and SSR.M2 unchanged by operation

DDOTPH2R *Double Dot Product With Rounding, Two Pairs, Signed, Packed 16-Bit*

Syntax **DDOTPH2R** (.unit) *src1_o:src1_e, src2, dst*

unit = .M1 or .M2

Compatibility C64x+ CPU

Opcode

31	30	29	28	27 23	22 18	17 13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	*dst*	*src2*	*src1*	x	0	1	0	1	0	1	1	1	0	0	*s*	*p*
				5	5	5	1											1	1

Opcode map field used...	For operand type...	Unit
src1	ds2	.M1, .M2
src2	xs2	
dst	s2	

Description Returns two dot-products between two pairs of signed, packed 16-bit values. The values in *src1_e, src1_o*, and *src2* are treated as signed, packed 16-bit quantities. The signed results are rounded, shifted right by 16 and packed into a 32-bit register.

The product of the lower halfwords of *src1_o* and *src2* is added to the product of the upper halfwords of *src1_o* and *src2*. The result is rounded by adding 2^{15} to it and saturated if appropriate. The 16 most-significant bits of the result are written to the 16 most-significant bits of *dst*.

The product of the upper halfword of *src2* and the lower halfword of *src1_o* is added to the product of the lower halfword of *src2* and the upper halfword of *src1*_e. The result is rounded by adding 2^{15} to it and saturated if appropriate. The 16 most-significant bits of the result are written to the 16 least-significant bits of *dst*.

If either result saturates, the M1 or M2 bit in SSR and the SAT bit in CSR are written one cycle after the results are written to *dst*.

This instruction executes unconditionally.

Execution

msb16(sat((msb16(*src1_o*) × msb16(*src2*)) +
(lsb16(*src1_o*) × lsb16(*src2*)) + 0000 8000h)) → msb16(*dst*)

msb16(sat((lsb16(*src1_o*) × msb16(*src2*)) +
(msb16(*src1_e*) × lsb16(*src2*)) + 0000 8000h)) → lsb16(*dst*)

Instruction Type Four-cycle

Delay Slots 3

See Also DDOTPH2, DDOTPL2, DDOTPL2R

Examples

Example 1

```
DDOTPH2R .M1 A5:A4,A6,A8
```

	Before instruction		4 cycles after instruction [(1)]
A4	46B4 16BAh	A8	F3B5 F41Bh
A5	BBAE D169h		
A6	340B F73Bh		

(1) CSR.SAT and SSR.M1 unchanged by operation

Example 2

```
DDOTPH2R .M1 A5:A4,A6,A8
```

	Before instruction		4 cycles after instruction
A4	8000 5678h	A8	36E6 7FFFh
A5	1234 8000h		
A6	8000 8001h		
CSR	0001 0100h	CSR[(1)]	0001 0300h
SSR	0000 0000h	SSR[(1)]	0000 0010h

(1) CSR.SAT and SSR.M1 set to 1, 5 cycles after instruction

Example 3

```
DDOTPH2R .M2 B5:B4,B6,B8
```

	Before instruction		4 cycles after instruction
B4	8000 8000h	B8	7FFF 7FFFh
B5	8000 8000h		
B6	8000 8001h		
CSR	0001 0100h	CSR[(1)]	0001 0300h
SSR	0000 0000h	SSR[(1)]	0000 0020h

(1) CSR.SAT and SSR.M2 set to 1, 5 cycles after instruction

DDOTPL2 *Double Dot Product, Two Pairs, Signed, Packed 16-Bit*

Syntax **DDOTPL2** (.unit) *src1_o:src1_e, src2, dst_o:dst_e*

unit = .M1 or .M2

Compatibility C64x+ CPU

Opcode

31	30	29	28	27–23	22–18	17–13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	*dst*	*src2*	*src1*	x	0	1	0	1	1	0	1	1	0	0	*s*	*p*
				5	5	5	1											1	1

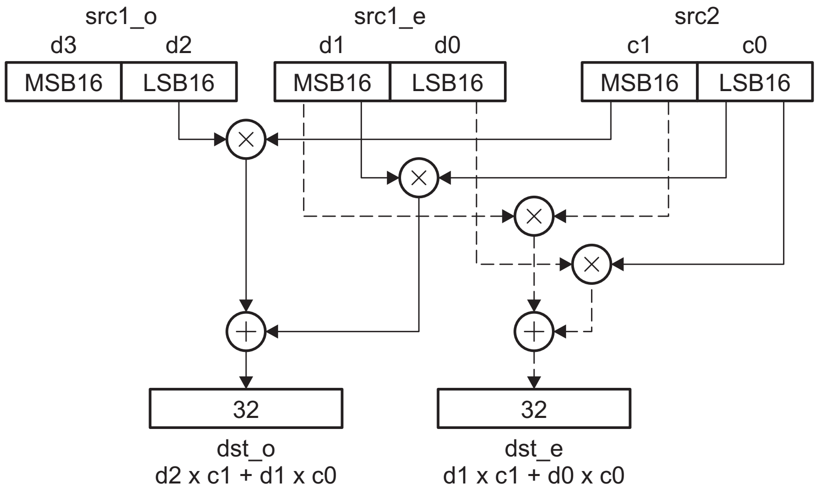
Opcode map field used...	For operand type...	Unit
src1	ds2	.M1, .M2
src2	xs2	
dst	dint	

Description Returns two dot-products between two pairs of signed, packed 16-bit values. The values in *src1_e, src1_o*, and *src2* are treated as signed, packed 16-bit quantities. The signed results are written to a 64-bit register pair.

The product of the lower halfwords of *src1_e* and *src2* is added to the product of the upper halfwords of *src1_e* and *src2*. The result is then written to *dst_e*.

The product of the upper halfword of *src2* and the lower halfword of *src1_o* is added to the product of the lower halfword of *src2* and the upper halfword of *src1*_e. The result is then written to *dst_o*.

If either result saturates, the M1 or M2 bit in SSR and the SAT bit in CSR are written one cycle after the results are written to *dst_o:dst_e*.



Execution

sat((msb16(*src1_e*) × msb16(*src2*)) + (lsb16(*src1_e*) × lsb16(*src2*))) → *dst_e*
sat((lsb16(*src1_o*) × msb16(*src2*)) + (msb16(*src1_e*) × lsb16(*src2*))) → *dst_o*

Instruction Type Four-cycle

Delay Slots 3

See Also DDOTPH2, DDOTPL2R, DDOTPH2R

Examples

Example 1

```
DDOTPL2 .M1 A5:A4,A6,A9:A8
```

	Before instruction			4 cycles after instruction [1]	
A4	0005 0003h	5 , 3	A8	0000 0026h	(4 × 7) + (5 × 1) = 33
A5	0002 0004h	2 , 4	A9	0000 0021h	(2 × 7) + (4 × 1) = 18
A6	0007 0001h	7 , 1			

[1] CSR.SAT and SSR.M1 unchanged by operation

Example 2

```
DDOTPL2 .M1 A5:A4,A6,A9:A8
```

	Before instruction		4 cycles after instruction [1]
A4	46B4 16BAh	A8	0D98 4C9Ah
A5	BBAE D169h	A9	F41B 4AFFh
A6	340B F73Bh		

[1] CSR.SAT and SSR.M1 unchanged by operation

Example 3

```
DDOTPL2 .M1 A5:A4,A6,A9:A8
```

	Before instruction		4 cycles after instruction
A4	8000 5678h	A8	14C4 0000h
A5	1234 8000h	A9	7FFF FFFFh
A6	8000 8000h		
CSR	0001 0100h	CSR[1]	0001 0300h
SSR	0000 0000h	SSR[1]	0000 0010h

[1] CSR.SAT and SSR.M1 set to 1, 5 cycles after instruction

DDOTPL2R *Double Dot Product With Rounding, Two Pairs, Signed Packed 16-Bit*

Syntax **DDOTPL2R** (.unit) *src1_o:src1_e, src2, dst*

unit = .M1 or .M2

Compatibility C64x+ CPU

Opcode

31	30	29	28	27–23	22–18	17–13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	*dst*	*src2*	*src1*	x	0	1	0	1	0	0	1	1	0	0	*s*	*p*
				5	5	5	1											1	1

Opcode map field used...	For operand type...	Unit
src1	ds2	.M1, .M2
src2	xs2	
dst	s2	

Description Returns two dot-products between two pairs of signed, packed 16-bit values. The values in *src1_e, src1_o*, and *src2* are treated as signed, packed 16-bit quantities. The signed results are rounded, shifted right by 16 and packed into a 32-bit register.

The product of the lower halfwords of *src1_e* and *src2* is added to the product of the upper halfwords of *src1_e* and *src2*. The result is rounded by adding 2^{15} to it and saturated if appropriate. The 16 most-significant bits of the result are written to the 16 least-significant bits of *dst*.

The product of the upper halfword of *src2* and the lower halfword of *src1_o* is added to the product of the lower halfword of *src2* and the upper halfword of *src1*_e. The result is rounded by adding 2^{15} to it and saturated if appropriate. The 16 most-significant bits of the result are written to the 16 most-significant bits of *dst*.

If either result saturates, the M1 or M2 bit in SSR and the SAT bit in CSR are written one cycle after the results are written to *dst*.

Execution

msb16(sat((msb16(*src1_e*) × msb16(*src2*)) +
(lsb16(*src1_e*) × lsb16(*src2*)) + 0000 8000h)) → lsb16(*dst*)

msb16(sat((lsb16(*src1_o*) × msb16(*src2*)) +
(msb16(*src1_e*) × lsb16(*src2*)) + 0000 8000h)) → msb16(*dst*)

Instruction Type Four-cycle

Delay Slots 3

See Also DDOTPH2R, DDOTPL2, DDOTPH2

Examples

Example 1

```
DDOTPL2R .M1 A5:A4,A6,A8
```

	Before instruction		4 cycles after instruction [1]
A4	46B4 16BAh	A8	F41B 0D98h
A5	BBAE D169h		
A6	340B F73Bh		

(1) CSR.SAT and SSR.M1 unchanged by operation

Example 2

```
DDOTPL2R .M1 A5:A4,A6,A8
```

	Before instruction		4 cycles after instruction
A4	8000 5678h	A8	7FFF 14C4h
A5	1234 8000h		
A6	8000 8001h		
CSR	0001 0100h	CSR[1]	0001 0300h
SSR	0000 0000h	SSR[1]	0000 0010h

(1) CSR.SAT and SSR.M1 set to 1, 5 cycles after instruction

Example 3

```
DDOTPL2R .M2 B5:B4,B6,B8
```

	Before instruction		4 cycles after instruction
B4	8000 8000h	B8	7FFF 7FFFh
B5	8000 8000h		
B6	8000 8001h		
CSR	0001 0100h	CSR[1]	0001 0300h
SSR	0000 0000h	SSR[1]	0000 0020h

(1) CSR.SAT and SSR.M2 set to 1, 5 cycles after instruction

DEAL *Deinterleave and Pack*

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

unit = .M1 or .M2

Compatibility C64x and C64x+ CPU

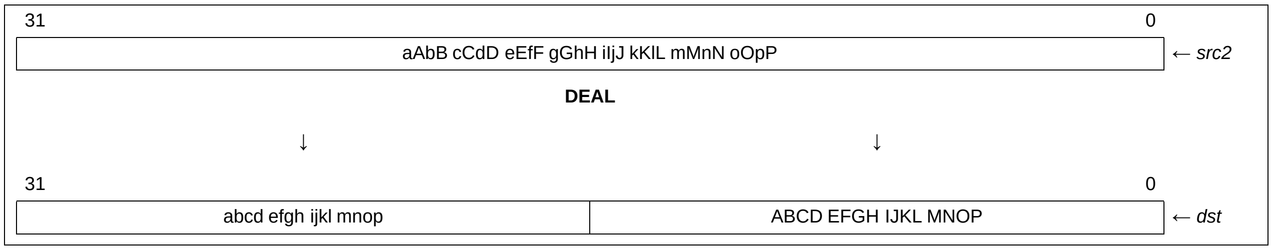
Opcode

31 29	28	27 23	22 18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	1	1	1	0	1	x	0	0	0	0	1	1	1	1	0	0	*s*	*p*
3	1	5	5						1											1	1

Opcode map field used...	For operand type...	Unit
src2	xuint	.M1, .M2
dst	uint	

Description Performs a deinterleave and pack operation on the bits in *src2*. The odd and even bits of *src2* are extracted into two separate, 16-bit quantities. These 16-bit quantities are then packed such that the even bits are placed in the lower halfword, and the odd bits are placed in the upper halfword.

As a result, bits 0, 2, 4, ... , 28, 30 of *src2* are placed in bits 0, 1, 2, ... , 14, 15 of *dst*. Likewise, bits 1, 3, 5, ... , 29, 31 of *src2* are placed in bits 16, 17, 18, ... , 30, 31 of *dst.*



NOTE: The **DEAL** instruction is the exact inverse of the **SHFL** instruction (see SHFL).

Execution

```
if (cond)    {
             $src2_{31,29,27...1} \rightarrow dst_{31,30,29...16}$
             $src2_{30,28,26...0} \rightarrow dst_{15,14,13...0}$
             }
else nop
```

Pipeline

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

Instruction Type Two-cycle

Delay Slots 1

See Also SHFL

Example `DEAL .M1 A1,A2`

	Before instruction		2 cycles after instruction
A1	9E52 6E30h	A1	9E52 6E30h
A2	xxxx xxxxh	A2	B174 6CA4h

DINT — *Disable Interrupts and Save Previous Enable State*

Syntax **DINT**

unit = none

Compatibility C64x+ CPU

Opcode

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	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	*p*
																															1

Description Disables interrupts in the current cycle, copies the contents of the GIE bit in TSR into the SGIE bit in TSR, and clears the GIE bit in both TSR and CSR. The PGIE bit in CSR is unchanged.

The CPU will not service a maskable interrupt in the cycle immediately following the **DINT** instruction. This behavior differs from writes to GIE using the **MVC** instruction. See section 5.2 for details.

The **DINT** instruction cannot be placed in parallel with the following instructions: **MVC** *reg*, **TSR**; **MVC** *reg*, **CSR**; **B IRP**; **B NRP**; **NOP** *n*; **RINT**; **SPKERNEL**; **SPKERNELR**; **SPLOOP**; **SPLOOPD**; **SPLOOPW**; **SPMASK**; or **SPMASKR**.

This instruction executes unconditionally.

NOTE: The use of the **DINT** and **RINT** instructions in a nested manner, like the following code:

```
DINT
DINT
RINT
RINT
```

leaves interrupts disabled. The first **DINT** leaves TSR.GIE cleared to 0, so the second **DINT** leaves TSR,.SGIE cleared to 0. The **RINT** instructions, therefore, copy zero to TSR.GIE (leaving interrupts disabled).

Execution Disable interrupts in current cycle

GIE bit in TSR → SGIE bit in TSR
0 → GIE bit in TSR
0 → GIE bit in CSR

Instruction Type Single-cycle

Delay Slots 0

See Also RINT

DMV *Move Two Independent Registers to Register Pair*

Syntax DMV (.unit) *src1, src2, dst_o:dst_e*

unit = .S1 or .S2

Compatibility C64x+ CPU

Opcode

31–29	28	27–23	22–18	17–13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	1	1	1	0	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	sint	.S1, .S2
src2	xsint	
dst	dint	

Description The *src1* operand is written to the odd register of the register pair specified by *dst* and the *src2* operand is written to the even register of the register pair specified by *dst*.

Execution

```
if (cond)     {
              src2 → dst_e
              src1 → dst_o
              }
else nop
```

Instruction Type Single-cycle

Delay Slots 0

Examples

Example 1

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

	Before instruction		1 cycle after instruction
A0	8765 4321h	A2	1234 5678h
A1	1234 5678h	A3	8765 4321h

Example 2

```
DMV .S2X B0,A1,B3:B2
```

	Before instruction		1 cycle after instruction
B0	0007 0009h	B2	1234 5678h
A1	1234 5678h	B3	0007 0009h

DOTP2 *Dot Product, Signed, Packed 16-Bit*

Syntax

DOTP2 (.unit) *src1, src2, dst*

or

DOTP2 (.unit) *src1, src2, dst_o:dst_e*

unit = .M1 or .M2

Compatibility C64x and C64x+ CPU

Opcode

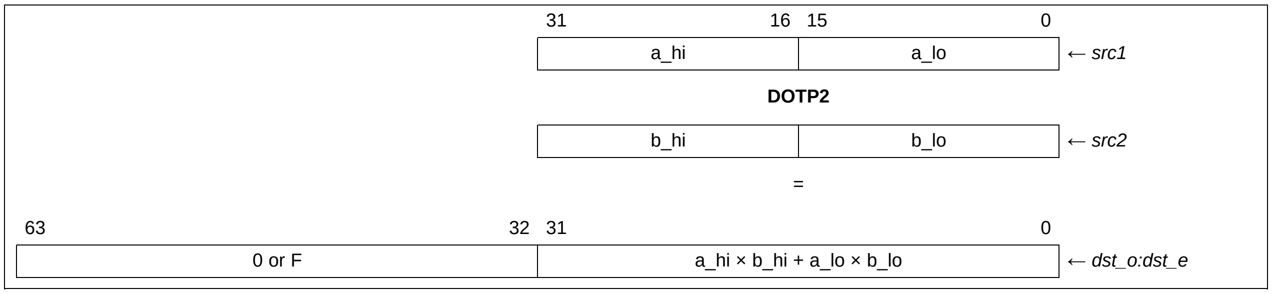
31–29	28	27–23	22–18	17–13	12	11	10–6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	0	*op*	1	1	0	0	*s*	*p*
3	1	5	5	5	1		5					1	1

Opcode map field used...	For operand type...	Unit	Opfield
src1 *src2* *dst*	s2 xs2 int	.M1, .M2	01100
src1 *src2* *dst*	s2 xs2 sllong	.M1, .M2	01011

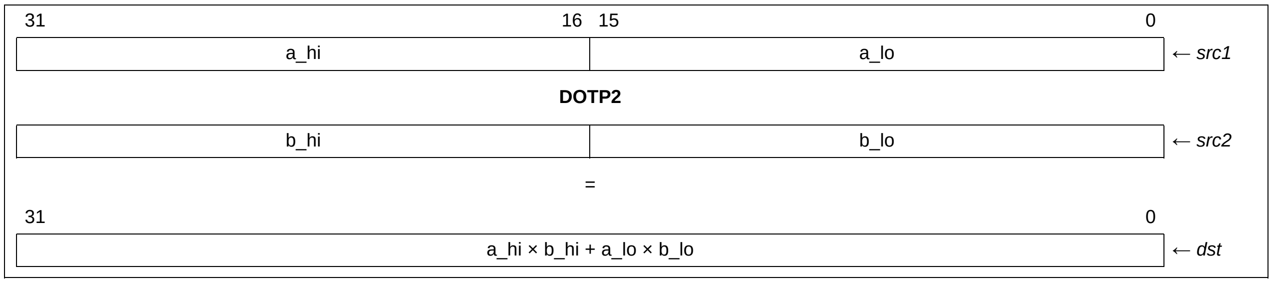
Description Returns the dot-product between two pairs of signed, packed 16-bit values. The values in *src1* and *src2* are treated as signed, packed 16-bit quantities. The signed result is written either to a single 32-bit register, or sign-extended into a 64-bit register pair.

The product of the lower halfwords of *src1* and *src2* is added to the product of the upper halfwords of *src1* and *src2*. The result is then written to the *dst*.

If the result is sign-extended into a 64-bit register pair, the upper word of the register pair always contains either all 0s or all 1s, depending on whether the result is positive or negative, respectively.



The 32-bit result version returns the same results that the 64-bit result version does in the lower 32 bits. The upper 32-bits are discarded.



NOTE: In the overflow case, where all four halfwords in *src1* and *src2* are 8000h, the value 8000 0000h is written into the 32-bit *dst* and 0000 0000 8000 0000h is written into the 64-bit *dst*.

Execution

if (cond) (lsb16(*src1*) × lsb16(*src2*)) + (msb16(*src1*) × msb16(*src2*)) → *dst*
else nop

Pipeline

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

Instruction Type Four-cycle

Delay Slots 3

See Also DOTPN2

Examples

Example 1

```
DOTP2 .M1 A5,A6,A8
```

	Before instruction			4 cycles after instruction	
A5	6A32 1193h	27186 4499	A5	6A32 1193h	
A6	B174 6CA4h	-20108 27812	A6	B174 6CA4h	
A8	xxxx xxxxh		A8	E6DF F6D4h	-421,529,900

Example 2

```
DOTP2 .M1 A5,A6,A9:A8
```

	Before instruction				4 cycles after instruction	
A5	6A32 1193h	27186 4499		A5	6A32 1193h	
A6	B174 6CA4h	-20108 27812		A6	B174 6CA4h	
A9:A8	xxxx xxxxh	xxxx xxxxh		A9:A8	FFFF FFFFh	E6DF F6D4h
						-421,529,900

Example 3

```
DOTP2 .M2 B2,B5,B8
```

	Before instruction			4 cycles after instruction	
B2	1234 3497h	4660 13463	B2	1234 3497h	
B5	21FF 50A7h	8703 20647	B5	21FF 50A7h	
B8	xxxx xxxxh		B8	12FC 544Dh	318,526,541

Example 4

```
DOTP2 .M2 B2,B5,B9:B8
```

	Before instruction				4 cycles after instruction	
B2	1234 3497h	4660 13463		B2	1234 3497h	
B5	21FF 50A7h	8703 20647		B5	21FF 50A7h	
B9:B8	xxxx xxxxh	xxxx xxxxh		B9:B8	0000 0000h	12FC 544Dh
						318,526,541

DOTPN2 *Dot Product With Negate, Signed, Packed 16-Bit*

Syntax **DOTPN2** (.unit) *src1, src2, dst*

unit = .M1 or .M2

Compatibility C64x and C64x+ CPU

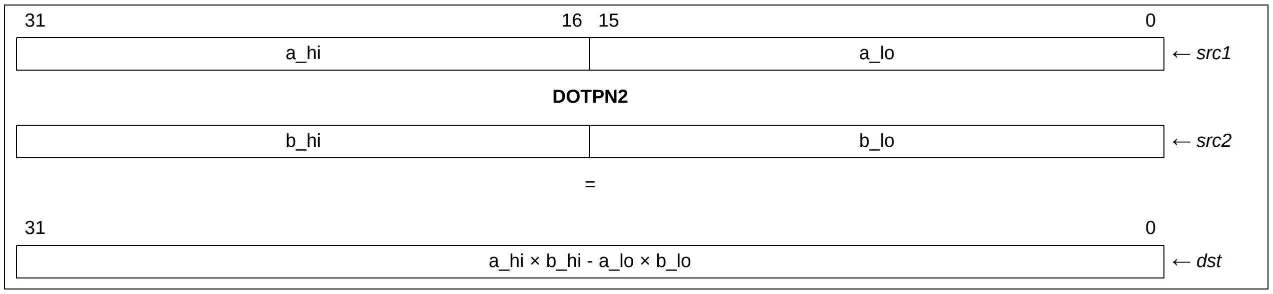
Opcode

31 29	28	27 23	22 18	17 13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	0	0	1	0	0	1	1	1	0	0	*s*	*p*
3	1	5	5	5	1											1	1

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

Description Returns the dot-product between two pairs of signed, packed 16-bit values where the second product is negated. The values in *src1* and *src2* are treated as signed, packed 16-bit quantities. The signed result is written to a single 32-bit register.

The product of the lower halfwords of *src1* and *src2* is subtracted from the product of the upper halfwords of *src1* and *src2*. The result is then written to *dst*.



Execution Note that unlike **DOTP2**, no overflow case exists for this instruction.

```
if (cond)    (msb16(src1) × msb16(src2)) - (lsb16(src1) × lsb16(src2)) → dst
else nop
```

Pipeline

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

Instruction Type Four-cycle

Delay Slots 3

See Also DOTP2

Examples

Example 1

```
DOTPN2 .M1 A5,A6,A8
```

	Before instruction			4 cycles after instruction	
A5	3629 274Ah	13865 10058	A5	3629 274Ah	
A6	325C 8036h	12892 -32714	A6	325C 8036h	
A8	xxxx xxxxh		A8	1E44 2F20h	507,784,992

Example 2

```
DOTPN2 .M2 B2,B5,B8
```

	Before instruction			4 cycles after instruction	
B2	3FF6 5010h	16374 20496	B2	3FF6 5010h	
B5	B1C3 0244h	-20029 580	B5	B1C3 0244h	
B8	xxxx xxxxh		B8	EBBE 6A22h	-339,842,526

DOTPNRSU2 *Dot Product With Negate, Shift and Round, Signed by Unsigned, Packed 16-Bit*

Syntax **DOTPNRSU2** (.unit) *src1, src2, dst*

unit = .M1 or .M2

Compatibility C64x and C64x+ CPU

Opcode

31–29	28	27–23	22–18	17–13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	0	0	0	1	1	1	1	1	0	0	*s*	*p*
3	1	5	5	5	1											1	1

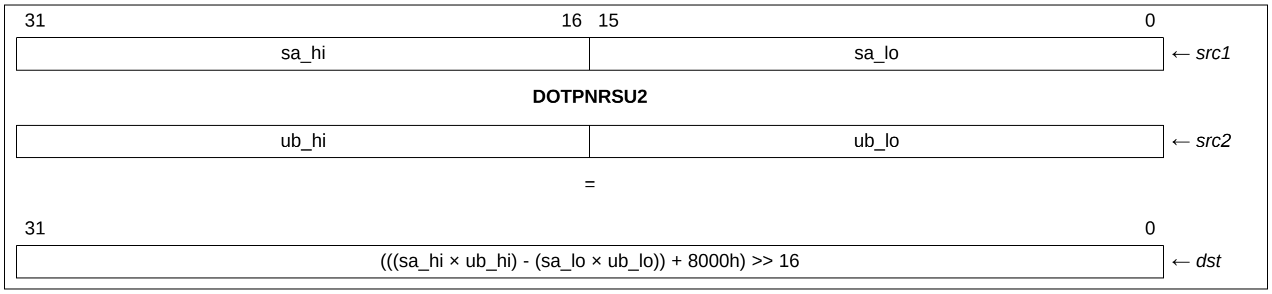
Opcode map field used...	For operand type...	Unit
src1	s2	.M1, .M2
src2	xu2	
dst	int	

Description Returns the dot-product between two pairs of packed 16-bit values, where the second product is negated. This instruction takes the result of the dot-product and performs an additional round and shift step. The values in *src1* are treated as signed, packed 16-bit quantities; whereas, the values in *src2* are treated as unsigned, packed 16-bit quantities. The results are written to *dst*.

The product of the lower halfwords of *src1* and *src2* is subtracted from the product of the upper halfwords of *src1* and *src2*. The value 2^{15} is then added to this sum, producing an intermediate 32 or 33-bit result. The intermediate result is signed shifted right by 16, producing a rounded, shifted result that is sign extended and placed in *dst*.

On the C64x CPU: The intermediate results of the **DOTPNRSU2** instruction are only maintained to a 32-bit precision. Overflow may occur during the rounding step. Overflow can be avoided if the difference of the two products plus the rounding term is less than or equal to 2^{31} - 1 for a positive sum and greater than or equal to -2^{31} for a negative sum.

On the C64x+ CPU: The intermediate results of the **DOTPNRSU2** instruction are maintained to a 33-bit precision, ensuring that no overflow may occur during the subtracting and rounding steps.



Execution

For C64x CPU:

```
if (cond)    {
             int32 = (smsb16(src1) × umsb16(src2)) -
             (slsb16(src1) × ulsb16(src2)) + 8000h;
             int32 >> 16 → dst
             }
else nop
```

For C64x+ CPU:

```
if (cond)    {
             int33 = (smsb16(src1) × umsb16(src2)) -
             (slsb16(src1) × ulsb16(src2)) + 8000h;
             int33 >> 16 → dst
             }
else nop
```

Pipeline

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

Instruction Type Four-cycle

Delay Slots 3

See Also DOTP2, DOTPN2, DOTPRSU2

Examples

Example 1

```
DOTPNRSU2 .M1 A5, A6, A8
```

	Before instruction			4 cycles after instruction	
A5	3629 274Ah	13865 10058 signed	A5	3629 274Ah	
A6	325C 8036h	12892 32822 unsigned	A6	325C 8036h	
A8	xxxx xxxxh		A8	FFFF F6FAh	-2310 (signed)

Example 2

```
DOTPNRSU2 .M2 B2, B5, B8
```

	Before instruction			4 cycles after instruction	
B2	3FF6 5010h	16374 20496 signed	B2	3FF6 5010h	
B5	B1C3 0244h	45507 580 unsigned	B5	B1C3 0244h	
B8	xxxx xxxxh		B8	0000 2BB4h	11188 (signed)

Example 3

```
DOTPNRSU2 .M2 B12, B23, B11
```

	Before instruction			4 cycles after instruction	
B12	7FFF 8000h	32767 -32768 signed	B12	7FFF 8000h	
B23	FFFF FFFFh	65535 65535 unsigned	B23	FFFF FFFFh	
B11	xxxx xxxxh		B11	xxxx xxxxh	Overflow occurs; result undefined

DOTPNRUS2 *Dot Product With Negate, Shift and Round, Unsigned by Signed, Packed 16-Bit*

Syntax **DOTPNRUS2** (.unit) *src2, src1, dst*

unit = .M1 or .M2

Compatibility C64x and C64x+ CPU

Opcode

31–29	28	27–23	22–18	17–13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	0	0	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	s2	.M1, .M2
src2	xu2	
dst	int	

Description The **DOTPNRUS2** pseudo-operation performs the dot-product between two pairs of packed 16-bit values, where the second product is negated. This instruction takes the result of the dot-product and performs an additional round and shift step. The values in *src1* are treated as signed, packed 16-bit quantities; whereas, the values in *src2* are treated as unsigned, packed 16-bit quantities. The results are written to *dst*. The assembler uses the **DOTPNRSU2** *src1, src2, dst* instruction to perform this task (see DOTPNRSU2).

The product of the lower halfwords of *src1* and *src2* is subtracted from the product of the upper halfwords of *src1* and *src2*. The value 2^{15} is then added to this sum, producing an intermediate 32 or 33-bit result. The intermediate result is signed shifted right by 16, producing a rounded, shifted result that is sign extended and placed in *dst*.

On the C64x CPU: The intermediate results of the **DOTPNRUS2** pseudo-operation are only maintained to a 32-bit precision. Overflow may occur during the rounding step. Overflow can be avoided if the difference of the two products plus the rounding term is less than or equal to 2^{31} - 1 for a positive sum and greater than or equal to -2^{31} for a negative sum.

On the C64x+ CPU: The intermediate results of the **DOTPNRUS2** pseudo-operation are maintained to a 33-bit precision, ensuring that no overflow may occur during the subtracting and rounding steps.

Execution **For C64x CPU:**

```
if (cond)    {
             int32 = (smsb16(src1) × umsb16(src2)) -
             (slsb16(src1) × ulsb16(src2)) + 8000h;
             int32 >> 16 → dst
             }
else nop
```

For C64x+ CPU:

```
if (cond)     {
              int33 = (smsb16(src1) × umsb16(src2)) -
              (slsb16(src1) × ulsb16(src2)) + 8000h;
              int33 >> 16 → dst
              }
else nop
```

Pipeline

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

Instruction Type Four-cycle

Delay Slots 3

See Also DOTP2, DOTPN2, DOTPNRSU2, DOTPRUS2

DOTPRSU2 *Dot Product With Shift and Round, Signed by Unsigned, Packed 16-Bit*

Syntax

DOTPRSU2 (.unit) *src1, src2, dst*

unit = .M1 or .M2

Compatibility

C64x and C64x+ CPU

Opcode

31–29	28	27–23	22–18	17–13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	0	0	1	1	0	1	1	1	0	0	*s*	*p*
3	1	5	5	5	1											1	1

Opcode map field used...	For operand type...	Unit
src1 *src2* *dst*	s2 xu2 int	.M1, .M2

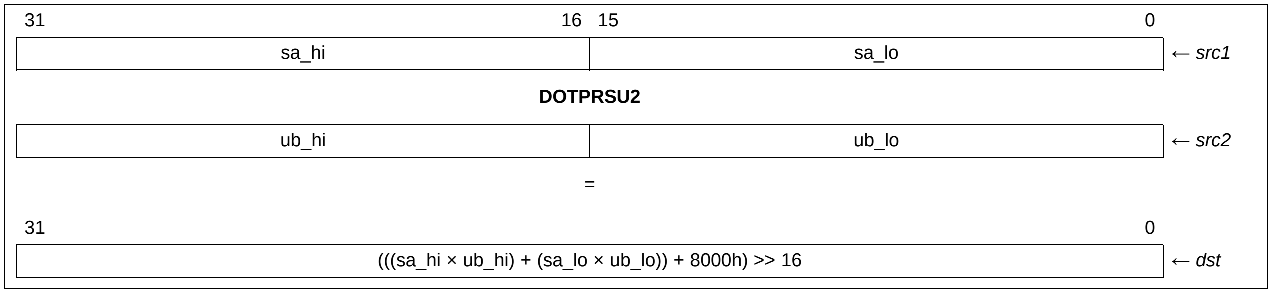
Description

Returns the dot-product between two pairs of packed 16-bit values. This instruction takes the result of the dot-product and performs an additional round and shift step. The values in *src1* are treated as signed packed 16-bit quantities; whereas, the values in *src2* are treated as unsigned packed 16-bit quantities. The results are written to *dst*.

The product of the lower halfwords of *src1* and *src2* is added to the product of the upper halfwords of *src1* and *src2*. The value $2^{15}$is then added to this sum, producing an intermediate 32 or 33-bit result. The intermediate result is signed shifted right by 16, producing a rounded, shifted result that is sign extended and placed in *dst*.

On the C64x CPU: The intermediate results of the **DOTPRSU2** instruction are only maintained to a 32-bit precision. Overflow may occur during the rounding step. Overflow can be avoided if the difference of the two products plus the rounding term is less than or equal to 2^{31} - 1 for a positive sum and greater than or equal to -2^{31} for a negative sum.

On the C64x+ CPU: The intermediate results of the **DOTPRSU2** instruction are maintained to a 33-bit precision, ensuring that no overflow may occur during the subtracting and rounding steps.



NOTE: Certain combinations of operands for the **DOTPRSU2** instruction results in an overflow condition. If an overflow does occur, the result is undefined. Overflow can be avoided if the sum of the two products plus the rounding term is less than or equal to $2^{31} - 1$ for a positive sum and greater than or equal to -2^{31} for a negative sum.

The intermediate results of the **DOTPRSU2** instruction are maintained to 33-bit precision, ensuring that no overflow may occur during the adding and rounding steps.

Execution

For C64x CPU:

```
if (cond)   {
            int32 = (smsb16(src1) × umsb16(src2)) +
            (slsb16(src1) × ulsb16(src2)) + 8000h;
            int32 >> 16 → dst
            }
else nop
```

For C64x+ CPU:

```
if (cond)   {
            int33 = (smsb16(src1) × umsb16(src2)) +
            (slsb16(src1) × ulsb16(src2)) + 8000h;
            int33 >> 16 → dst
            }
else nop
```

Pipeline

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

Instruction Type Four-cycle

Delay Slots 3

See Also DOTP2, DOTPN2, DOTPNRSU2

Examples

Example 1

```
DOTPRSU2 .M1 A5, A6, A8
```

	Before instruction			**4 cycles after instruction**	
A5	3629 274Ah	13865 10058 signed	A5	3629 274Ah	
A6	325C 8036h	12892 32822 unsigned	A6	325C 8036h	
A8	xxxx xxxxh		A8	0000 1E55h	7765 (signed)

Example 2

```
DOTPRSU2 .M2 B2, B5, B8
```

	Before instruction			**4 cycles after instruction**	
B2	B1C3 0244h	-20029 580 signed	B2	B1C3 0244h	20029 580 signed
B5	3FF6 5010h	16374 20496 unsigned	B5	3FF6 5010h	16374 20496 unsigned
B8	xxxx xxxxh		B8	FFFF ED29h	-4823 (signed)

Example 3

```
DOTPRSU2 .M2 B12, B23, B11
```

	Before instruction			**4 cycles after instruction**	
B12	7FFF 7FFFh	32767 32767 signed	B12	7FFF 7FFFh	
B23	FFFF FFFFh	65535 65535 unsigned	B23	FFFF FFFFh	
B11	xxxx xxxxh		B11	xxxx xxxxh	Overflow occurs; result undefined

DOTPRUS2 *Dot Product With Shift and Round, Unsigned by Signed, Packed 16-Bit*

Syntax **DOTPRUS2** (.unit) *src2, src1, dst*

unit = .M1 or .M2

Compatibility C64x and C64x+ CPU

Opcode

31–29	28	27–23	22–18	17–13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	0	0	1	1	0	1	1	1	0	0	*s*	*p*
3	1	5	5	5	1											1	1

Opcode map field used...	For operand type...	Unit
src1	s2	.M1, .M2
src2	xu2	
dst	int	

Description The **DOTPRUS2** pseudo-operation returns the dot-product between two pairs of packed 16-bit values. This instruction takes the result of the dot-product, and performs an additional round and shift step. The values in *src1* are treated as signed packed 16-bit quantities; whereas, the values in *src2* are treated as unsigned packed 16-bit quantities. The results are written to *dst*. The assembler uses the **DOTPRSU2** (.unit) *src1, src2, dst* instruction to perform this task (see DOTPRSU2).

The product of the lower halfwords of *src1* and *src2* is added to the product of the upper halfwords of *src1* and *src2*. The value $2^{15}$is then added to this sum, producing an intermediate 32-bit result. The intermediate result is signed shifted right by 16, producing a rounded, shifted result that is sign extended and placed in *dst*.

On the C64x CPU: The intermediate results of the **DOTPRUS2** pseudo-operation are only maintained to a 32-bit precision. Overflow may occur during the rounding step. Overflow can be avoided if the difference of the two products plus the rounding term is less than or equal to 2^{31} **-** 1 for a positive sum and greater than or equal to -2^{31} for a negative sum.

On the C64x+ CPU: The intermediate results of the **DOTPRUS2** pseudo-operation are maintained to a 33-bit precision, ensuring that no overflow may occur during the subtracting and rounding steps.

Execution **For C64x CPU:**

```
if (cond)    {
             int32 = (umsb16(src2) × smsb16(src1)) +
             (ulsb16(src2) × slsb16(src1)) + 8000h;
             int32 >> 16 → dst
             }
else nop
```

For C64x+ CPU:

```
if (cond)    {
             int33 = (umsb16(src2) × smsb16(src1)) +
             (ulsb16(src2) × slsb16(src1)) + 8000h;
             int33 >> 16 → dst
             }
else nop
```

Pipeline

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

Instruction Type Four-cycle

Delay Slots 3

See Also DOTP2, DOTPN2, DOTPNRUS2, DOTPRSU2

DOTPSU4 *Dot Product, Signed by Unsigned, Packed 8-Bit*

Syntax **DOTPSU4** (.unit) *src1, src2, dst*

unit = .M1 or .M2

Compatibility C64x and C64x+ CPU

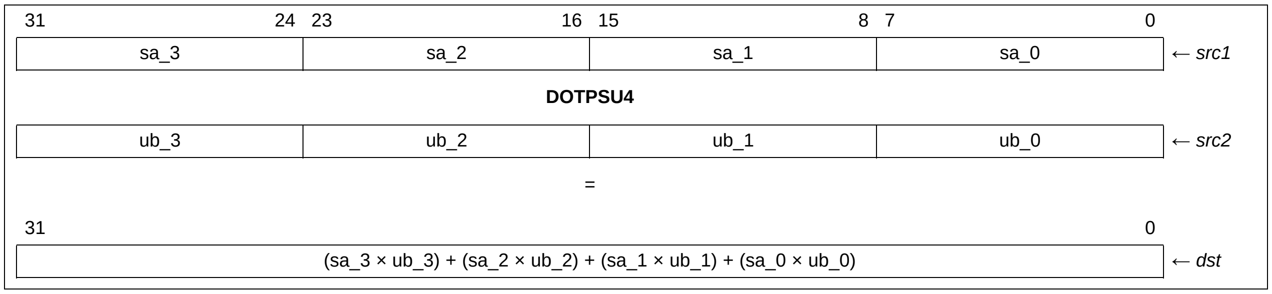
Opcode

31–29	28	27–23	22–18	17–13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	0	0	0	0	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	s4	.M1, .M2
src2	xu4	
dst	int	

Description Returns the dot-product between four sets of packed 8-bit values. The values in *src1* are treated as signed packed 8-bit quantities; whereas, the values in *src2* are treated as unsigned 8-bit packed data. The signed result is written into *dst*.

For each pair of 8-bit quantities in *src1* and *src2*, the signed 8-bit value from *src1* is multiplied with the unsigned 8-bit value from *src2*. The four products are summed together, and the resulting dot product is written as a signed 32-bit result to *dst*.



Execution

```
if (cond)   {
            (sbyte0(src1) × ubyte0(src2)) +
            (sbyte1(src1) × ubyte1(src2)) +
            (sbyte2(src1) × ubyte2(src2)) +
            (sbyte3(src1) × ubyte3(src2)) → dst
            }
else nop
```

Pipeline

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

Instruction Type Four-cycle

Delay Slots 3

See Also DOTPU4

Examples

Example 1

```
DOTPSU4 .M1 A5, A6, A8
```

	Before instruction			4 cycles after instruction	
A5	6A 32 11 93h	106 50 17 -109 signed	A5	6A 32 11 93h	
A6	B1 74 6C A4h	177 116 108 164 unsigned	A6	B1 74 6C A4h	
A8	xxxx xxxxh		A8	0000 214Ah	8522 (signed)

Example 2

```
DOTPSU4 .M2 B2, B5, B8
```

	Before instruction			4 cycles after instruction	
B2	3F F6 50 10h	63 -10 80 16 signed	B2	3F F6 50 10h	
B5	C3 56 02 44h	195 86 2 68 unsigned	B5	C3 56 02 44h	
B8	xxxx xxxxh		B8	0000 3181h	12,673 (signed)

DOTPUS4 *Dot Product, Unsigned by Signed, Packed 8-Bit*

Syntax

DOTPUS4 (.unit) *src2, src1, dst*

unit = .M1 or .M2

Compatibility C64x and C64x+ CPU

Opcode

31–29	28	27–23	22–18	17–13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	0	0	0	0	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	s4	.M1, .M2
src2	xu4	
dst	int	

Description

The **DOTPUS4** pseudo-operation returns the dot-product between four sets of packed 8-bit values. The values in *src1* are treated as signed packed 8-bit quantities; whereas, the values in *src2* are treated as unsigned 8-bit packed data. The signed result is written into *dst*. The assembler uses the **DOTPSU4** (.unit) *src1*, *src2*, *dst* instruction to perform this task (see DOTPSU4).

For each pair of 8-bit quantities in *src1* and *src2*, the signed 8-bit value from *src1* is multiplied with the unsigned 8-bit value from *src2*. The four products are summed together, and the resulting dot-product is written as a signed 32-bit result to *dst*.

Execution

```
if (cond)    {
             (ubyte0(src2) × sbyte0(src1)) +
             (ubyte1(src2) × sbyte1(src1)) +
             (ubyte2(src2) × sbyte2(src1)) +
             (ubyte3(src2) × sbyte3(src1)) → dst
             }
else nop
```

Pipeline

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

Instruction Type Four-cycle

Delay Slots 3

See Also DOTPU4, DOTPSU4

DOTPU4 *Dot Product, Unsigned, Packed 8-Bit*

Syntax **DOTPU4** (.unit) *src1, src2, dst*

unit = .M1 or .M2

Compatibility C64x and C64x+ CPU

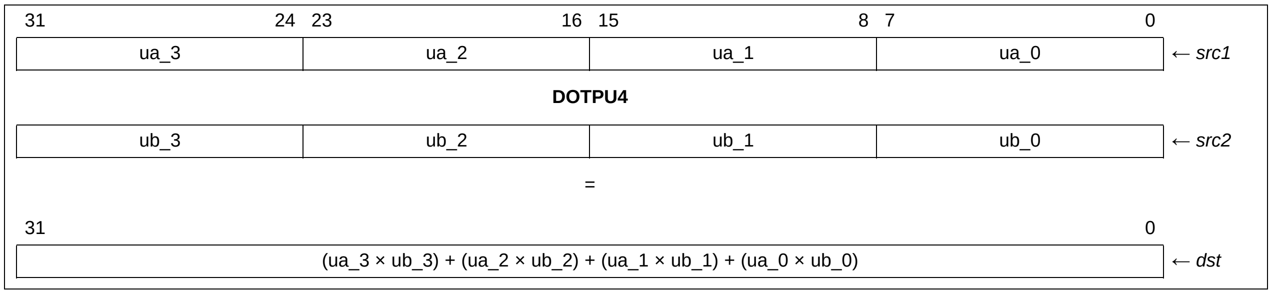
Opcode

31 29	28	27 23	22 18	17 13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	0	0	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	u4	.M1, .M2
src2	xu4	
dst	uint	

Description Returns the dot-product between four sets of packed 8-bit values. The values in both *src1* and *src2* are treated as unsigned, 8-bit packed data. The unsigned result is written into *dst*.

For each pair of 8-bit quantities in *src1* and *src2*, the unsigned 8-bit value from *src1* is multiplied with the unsigned 8-bit value from *src2*. The four products are summed together, and the resulting dot-product is written as a 32-bit result to *dst*.


31 24 23 16 15 8 7 0
ua_3 ua_2 ua_1 ua_0 ← src1
DOTPU4
ub_3 ub_2 ub_1 ub_0 ← src2
=
31 0
(ua_3 × ub_3) + (ua_2 × ub_2) + (ua_1 × ub_1) + (ua_0 × ub_0) ← dst

Execution

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

Pipeline

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

Instruction Type Four-cycle

Delay Slots 3

See Also DOTPSU4

Example `DOTPU4 .M1 A5, A6, A8`

	Before instruction			4 cycles after instruction	
A5	6A 32 11 93h	106 50 17 147 unsigned	A5	6A 32 11 93h	
A6	B1 74 6C A4h	177 116 108 164 unsigned	A6	B1 74 6C A4h	
A8	xxxx xxxxh		A8	0000 C54Ah	50,506 (unsigned)

DPACK2 *Parallel PACK2 and PACKH2 Operations*

Syntax **DPACK2** (.unit) *src1, src2, dst_o:dst_e*

unit = .L1 or .L2

Compatibility C64x+ CPU

Opcode

31	30	29	28	27–24	23	22–18	17–13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	*dst*	0	*src2*	*src1*	x	0	1	1	0	1	0	0	1	1	0	*s*	*p*
				4		5	5	1											1	1

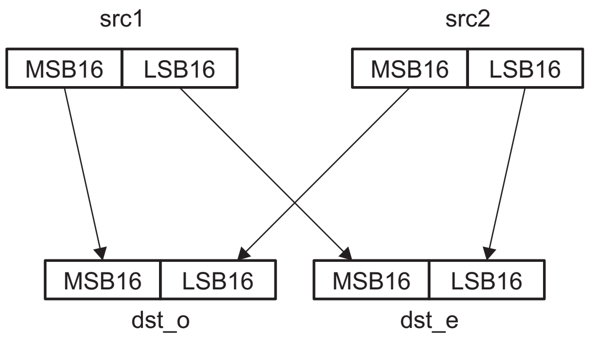
Opcode map field used...	For operand type...	Unit
src1	sint	.L1, .L2
src2	xsint	
dst	dint	

Description Executes a **PACK2** instruction in parallel with a **PACKH2** instruction.

The **PACK2** function of the **DPACK2** instruction takes the lower halfword from *src1* and the lower halfword from *src2,* and packs them both into *dst_e*. The lower halfword of *src1* is placed in the upper halfword of *dst_e*. The lower halfword of *src2* is placed in the lower halfword of *dst_e*.

The **PACKH2** function of the **DPACK2** instruction takes the upper halfword from *src1* and the upper halfword from *src2*, and packs them both into *dst_o*. The upper halfword of *src1* is placed in the upper halfword of *dst_o*. The upper halfword of *src2* is placed in the lower halfword of *dst_o*.

This instruction executes unconditionally.



Execution

lsb16(*src1*) → msb16(*dst_e*)
lsb16(*src2*) → lsb16(*dst_e*)
msb16(*src1*) → msb16(*dst_o*)
msb16(*src2*) → lsb16(*dst_o*)

Instruction Type Single-cycle

Delay Slots 0

Example

```
DPACK2 .L1 A0,A1,A3:A2
```

	Before instruction		1 cycle after instruction
A0	8765 4321h	A2	4321 5678h
A1	1234 5678h	A3	8765 1234h

DPACKX2 *Parallel PACKLH2 Operations*

Syntax **DPACKX2** (.unit) *src1, src2, dst_o:dst_e*

unit = .L1 or .L2

Compatibility C64x+ CPU

Opcode

31	30	29	28	27–24	23	22–18	17–13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	*dst*	0	*src2*	*src1*	x	0	1	1	0	0	1	1	1	1	0	*s*	*p*
				4		5	5	1											1	1

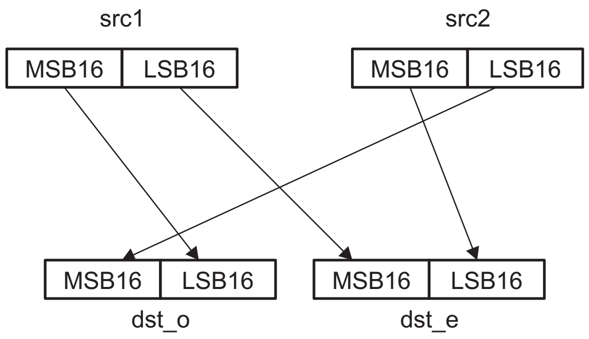
Opcode map field used...	For operand type...	Unit
src1	sint	.L1, .L2
src2	xsint	
dst	dint	

Description Executes two **PACKLH2** instructions in parallel.

One **PACKLH2** function of the **DPACKX2** instruction takes the lower halfword from *src1* and the upper halfword from *src2,* and packs them both into *dst_e*. The lower halfword of *src1* is placed in the upper halfword of *dst_e*. The upper halfword of *src2* is placed in the lower halfword of *dst_e*.

The other **PACKLH2** function of the **DPACKX2** instruction takes the upper halfword from *src1* and the lower halfword from *src2*, and packs them both into *dst_o*. The upper halfword of *src1* is placed in the lower halfword of *dst_o*. The lower halfword of *src2* is placed in the upper halfword of *dst_o*.

This instruction executes unconditionally.



Execution

lsb16(*src1*) → msb16(*dst_e*)
msb16(*src2*) → lsb16(*dst_e*)
msb16(*src1*) → lsb16(*dst_o*)
lsb16(*src2*) → msb16(*dst_o*)

Instruction Type Single-cycle

Delay Slots 0

Examples

Example 1

```
DPACKX2 .L1 A0,A1,A3:A2
```

	Before instruction		1 cycle after instruction
A0	8765 4321h	A2	4321 1234h
A1	1234 5678h	A3	5678 8765h

Example 2

```
DPACKX2 .L1X A0,B0,A3:A2
```

	Before instruction		1 cycle after instruction
A0	3FFF 8000h	A2	8000 4000h
B0	4000 7777h	A3	7777 3FFFh

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, and C64x+ CPU

Compact Instruction Format

Unit	Opcode Format	Figure
.S	S2ext	Figure F-27

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

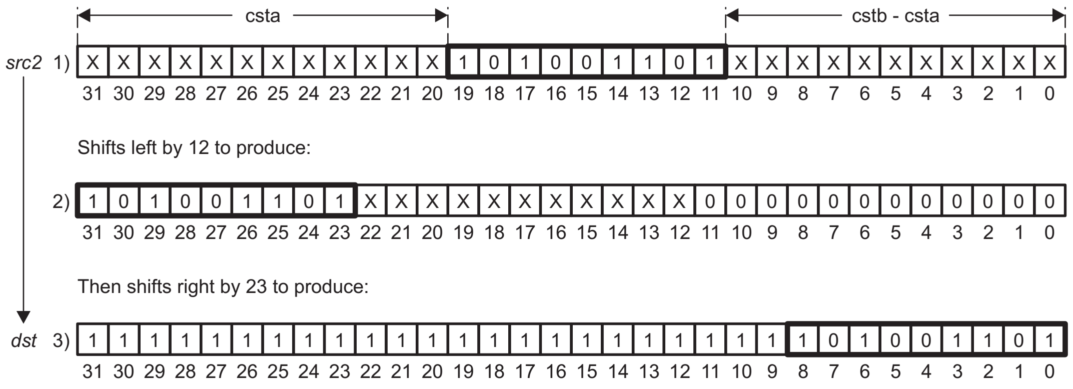
Opcode map field used...	For operand type...	Unit
src2	sint	.S1, .S2
csta	ucst5	
cstb	ucst5	
dst	sint	

Opcode Register form

31 29	28	27 23	22 18	17 13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	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	xsint	.S1, .S2
src1	uint	
dst	sint	

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.


csta
cstb - csta
src2 1)
X X X X X X X X X X X X 1 0 1 0 0 1 1 0 1 X X X X X X X X X X X
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
Shifts left by 12 to produce:
2)
1 0 1 0 0 1 1 0 1 X X X X X X X X X X X 0 0 0 0 0 0 0 0 0 0 0 0
Then shifts right by 23 to produce:
dst 3)
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 0 0 1 1 0 1

Execution If the constant form is used:

if (cond) *src2* 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

Examples

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, and C64x+ CPU

Compact Instruction Format

Unit	Opcode Format	Figure
.S	Sc5	Figure F-26
	S2ext	Figure F-27

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	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	

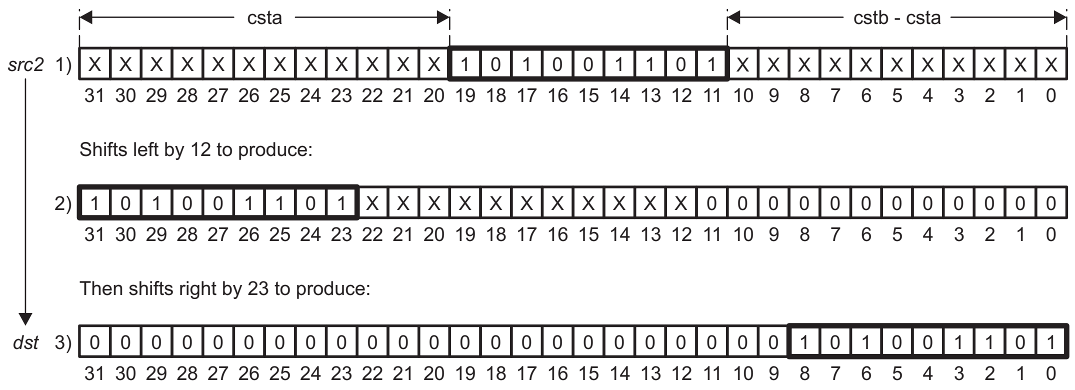
Opcode Register form:

31 29	28	27 23	22 18	17 13	12	11	10	9	8	7	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.


csta
cstb - csta
src2 1)
X X X X X X X X X X X X 1 0 1 0 0 1 1 0 1 X X X X X X X X X X X
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
Shifts left by 12 to produce:
2)
1 0 1 0 0 1 1 0 1 X X X X X X X X X X X X 0 0 0 0 0 0 0 0 0 0 0
Then shifts right by 23 to produce:
dst 3)
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 1 1 0 1

Execution

If the constant form is used:

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

If the register form is used:

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

Pipeline

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

Instruction Type Single-cycle

Delay Slots 0

See Also EXT

Examples

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

GMPY *Galois Field Multiply*

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

unit = .M1 or .M2

Compatibility C64x+ CPU

Opcode

31	30	29	28	27–23	22–18	17–13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	*dst*	*src2*	*src1*	x	0	1	1	1	1	1	1	1	0	0	*s*	*p*
				5	5	5	1											1	1

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

Description Performs a Galois field multiply, where *src1* is 32 bits and *src2* is limited to 9 bits. This utilizes the existing hardware and produces a 32-bit result. This multiply connects all levels of the gmpy4 together and only extends out by 8 bits, the resulting data is XORed down by the 32-bit polynomial.

The polynomial used comes from either the GPLYA or GPLYB control register depending on which side (A or B) the instruction executes. If the A-side M1 unit is used, the polynomial comes from GPLYA; if the B-side M2 unit, the polynomial comes from GPLYB.

This instruction executes unconditionally.

```
uword gmpy(uword src1,uword src2,uword polynomial)
{
  // the multiply is always between GF(2^9) and GF(2^32)
  // so no size information is needed

  uint pp;
  uint mask, tpp;
  uint I;

      pp = 0;
      mask = 0x00000100; // multiply by computing
                         // partial products.
      for ( I=0; i<8; I++ ){
        if ( src2 & mask )   pp ^= src1;
        mask >>= 1;
        tpp = pp << 1;
        if (pp & 0x80000000) pp = polynomial ^ tpp;
        else                 pp = tpp;
      }
if ( src2 & 0x1 ) pp ^= src1;

return (pp) ;   // leave it asserted left.
}
```

Execution

```
if (unit = M1)
        GMPY_poly = GPLYA
        lsb9(src2) gmpy src1 → dst
else if (unit = M2)
        GMPY_poly = GPLYB
        lsb9(src2) gmpy src1 → dst
```

Instruction Type Four-cycle

Delay Slots 3

See Also GMPY4, XORMPY, XOR

Example `GMPY .M1 A0,A1,A2 GPLYA = 87654321`

	Before instruction		4 cycles after instruction
A0	1234 5678h	A2	C721 A0EFh
A1	0000 0126h		

GMPY4 *Galois Field Multiply, Packed 8-Bit*

Syntax **GMPY4** (.unit) *src1, src2, dst*

unit = .M1 or .M2

Compatibility C64x and C64x+ CPU

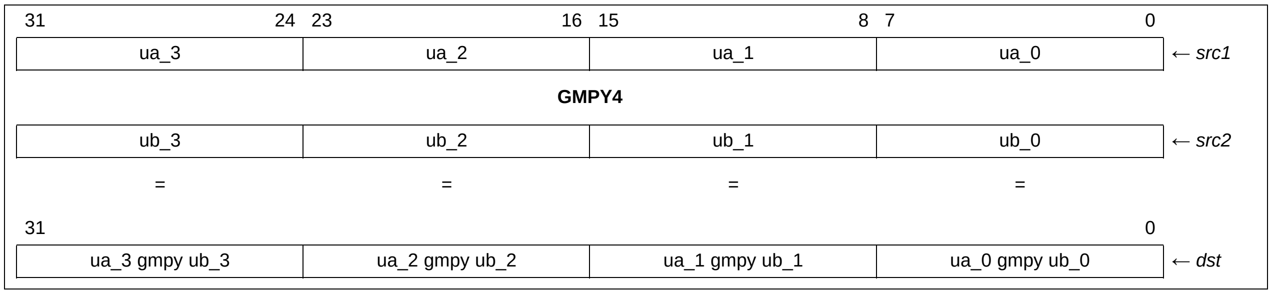
Opcode

31–29	28	27–23	22–18	17–13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	1	*dst*	*src2*	*src1*	x	0	1	0	0	0	1	1	1	0	0	*s*	*p*
3		5	5	5	1											1	1

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

Description Performs the Galois field multiply on four values in *src1* with four parallel values in *src2*. The four products are packed into dst. The values in both *src1* and *src2* are treated as unsigned, 8-bit packed data.

For each pair of 8-bit quantities in *src1* and *src2*, the unsigned, 8-bit value from *src1* is Galois field multiplied (gmpy) with the unsigned, 8-bit value from *src2*. The product of *src1* byte 0 and *src2* byte 0 is written to byte0 of *dst*. The product of *src1* byte 1 and *src2* byte 1 is written to byte1 of *dst*. The product of *src1* byte 2 and *src2* byte 2 is written to byte2 of *dst*. The product of *src1* byte 3 and *src2* byte 3 is written to the most-significant byte in *dst*.



The size and polynomial are controlled by the Galois field polynomial generator function register (GFPGFR). All registers in the control register file can be written using the **MVC** instruction (see MVC).

The default field generator polynomial is 1Dh, and the default size is 7. This setting is used for many communications standards.

Note that the **GMPY4** instruction is commutative, so:

```
GMPY4 .M1 A10,A12,A13
```

is equivalent to:

```
GMPY4 .M1 A12,A10,A13
```

Execution

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

Pipeline

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

Instruction Type　Four-cycle

Delay Slots　3

See Also　GMPY, MVC, XOR

Examples

Example 1

```
GMPY4 .M1 A5,A6,A7; polynomial = 0x1d
```

	Before instruction			4 cycles after instruction	
A5	45 23 00 01h	69 35 0 1 unsigned	A5	45 23 00 01h	
A6	57 34 00 01h	87 52 0 1 unsigned	A6	57 34 00 01h	
A7	xxxx xxxxh		A7	72 92 00 01h	114 146 0 1 unsigned

Example 2

```
GMPY4 .M1 A5,A6,A7; field size is 0x7
```

	Before instruction			4 cycles after instruction	
A5	FF FE 02 1Fh	255 254 2 31 unsigned	A5	FF FE 02 1Fh	
A6	FF FE 02 01h	255 254 2 1 unsigned	A6	FF FE 02 01h	
A7	xxxx xxxxh		A7	E2 E3 04 1Fh	226 227 4 31 unsigned

IDLE *Multicycle NOP With No Termination Until Interrupt*

Syntax **IDLE**

unit = none

Compatibility C62x, C64x, and C64x+ CPU

Opcode

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	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	*p*
																															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.

The **IDLE** instruction cannot be paired with any other multicycle **NOP** instruction in the same execute packet. Instructions that generate a multicycle **NOP** are: **ADDKPC**, **BNOP**, and the multicycle **NOP**.

Instruction Type NOP

Delay Slots 0

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

unit = .D1 or .D2

Compatibility C62x, C64x, and C64x+ CPU

Compact Instruction Format

Unit	Opcode Format	Figure
.D	Doff4	Figure C-9
	Dind	Figure C-11
	Dinc	Figure C-13
	Ddec	Figure C-15

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 3-18 summarizes the data types supported by loads. Table 3-6 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.

Table 3-18. 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

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.

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 2.8.3).

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.

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 → *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

Examples

Example 1

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

	Before instruction		1 cycle after instruction		5 cycles after instruction
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

Example 2

```
LDB .D1 *++A4[5],A8
```

	Before instruction		1 cycle after instruction		5 cycles after instruction
A4	0000 0400h	A4	0000 4005h	A4	0000 4005h
A8	0000 0000h	A8	0000 0000h	A8	0000 0067h
AMR	0000 0000h	AMR	0000 0000h	AMR	0000 0000h
mem 4000h	0112 2334h	mem 4000h	0112 2334h	mem 4000h	0112 2334h
mem 4004h	4556 6778h	mem 4004h	4556 6778h	mem 4004h	4556 6778h

Example 3

```
LDB .D1 *A4++[5],A8
```

	Before instruction		1 cycle after instruction		5 cycles after instruction
A4	0000 0400h	A4	0000 4005h	A4	0000 4005h
A8	0000 0000h	A8	0000 0000h	A8	0000 0034h
AMR	0000 0000h	AMR	0000 0000h	AMR	0000 0000h
mem 4000h	0112 2334h	mem 4000h	0112 2334h	mem 4000h	0112 2334h
mem 4004h	4556 6778h	mem 4004h	4556 6778h	mem 4004h	4556 6778h

Example 4

```
LDB .D1 *++A4[A12],A8
```

	Before instruction		1 cycle after instruction		5 cycles after instruction
A4	0000 0400h	A4	0000 4006h	A4	0000 4006h
A8	0000 0000h	A8	0000 0000h	A8	0000 0056h
A12	0000 0006h	A12	0000 0006h	A12	0000 0006h
AMR	0000 0000h	AMR	0000 0000h	AMR	0000 0000h
mem 4000h	0112 2334h	mem 4000h	0112 2334h	mem 4000h	0112 2334h
mem 4004h	4556 6778h	mem 4004h	4556 6778h	mem 4004h	4556 6778h

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, and C64x+ 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 3-19 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*15offset 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 3-19. 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 instruction		1 cycle after instruction
B1	xxxx xxxxh	B1	xxxx xxxxh
B14	0000 0100h	B14	0000 0100h
mem 124-127h	4E7A FF12h	mem 124-127h	4E7A FF12h
mem 124h	12h	mem 124h	12h
			5 cycles after instruction
		B1	0000 0012h
		B14	0000 0100h
		mem 124-127h	4E7A FF12h
		mem 124h	12h

LDDW *Load Doubleword From Memory With a 5-Bit 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 C64x and C64x+ CPU

Compact Instruction Format

Unit	Opcode Format	Figure
.D	Doff4DW	Figure C-10
	DindDW	Figure C-12
	DincDW	Figure C-14
	DdecDW	Figure C-16
	Dpp	Figure C-22

Opcode

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

Description Loads a 64-bit quantity from memory into a register pair *dst_o:dst_e.* Table 3-6 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 *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 2.8.3).

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 32 least-significant bits are loaded into the even-numbered register and the 32 most-significant 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

Examples

Example 1

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

	Before instruction				**5 cycles after instruction**	
A1:A0	xxxx xxxxh	xxxx xxxxh		A1:A0	4021 3333h	3333 3333h
B10	0000 0010h		16	B10	0000 0010h	
mem 18h	3333 3333h	4021 3333h	8.6	mem 18h	3333 3333h	4021 3333h
					Little-endian mode	

Example 2

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

	Before instruction				**1 cycle after instruction**		
A1:A0	xxxx xxxxh	xxxx xxxxh		A1:A0	xxxx xxxxh	xxxx xxxxh	
A10	0000 0010h		16	A10	0000 0018h		24
mem 18h	4021 3333h	3333 3333h	8.6	mem 18h	4021 3333h	3333 3333h	
					5 cycles after instruction		
				A1:A0	4021 3333h	3333 3333h	
				A10	0000 0018h		24
				mem 18h	4021 3333h	3333 3333h	
					Big-endian mode		

Example 3

```
LDDW .D1 *A4++[5],A9:A8
```

	Before instruction			**1 cycle after instruction**	
A9:A8	xxxx xxxxh	xxxx xxxxh	A9:A8	xxxx xxxxh	xxxx xxxxh
A4	0000 40B0h		A4	0000 40B0h	
mem 40B0h	0112 2334h	4556 6778h	mem 40B0h	0112 2334h	4556 6778h
				5 cycles after instruction	
			A9:A8	4556 6778h	0112 2334h
			A4	0000 40B0h	
			mem 40B0h	0112 2334h	4556 6778h
				Little-endian mode	

Example 4

```
LDDW .D1 *++A4[A12],A9:A8
```

	Before instruction				1 cycle after instruction	
A9:A8	xxxx xxxxh	xxxx xxxxh		A9:A8	xxxx xxxxh	xxxx xxxxh
A4	0000 40B0h			A4	0000 40E0h	
A12	0000 0006h			A12	0000 0006h	
mem 40E0h	0112 2334h	4556 6778h	8	mem 40E0h	0112 2334h	4556 6778h
					5 cycles after instruction	
				A9:A8	4556 6778h	0112 2334h
				A4	0000 40E0h	
				A12	0000 0006h	
				mem 40E0h	0112 2334h	4556 6778h
					Little-endian mode	

Example 5

```
LDDW .D1 *++A4(16),A9:A8
```

	Before instruction			1 cycle after instruction	
A9:A8	xxxx xxxxh	xxxx xxxxh	A9:A8	xxxx xxxxh	xxxx xxxxh
A4	0000 40B0h		A4	0000 40C0h	
mem 40C0h	4556 6778h	899A ABBCh	mem 40C0h	4556 6778h	899A ABBCh
				5 cycles after instruction	
			A9:A8	899A ABBCh	4556 6778h
			A4	0000 40C0h	
			mem 40C0h	4556 6778h	899A ABBCh
				Little-endian mode	

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

Syntax

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

unit = .D1 or .D2

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

Compatibility C62x, C64x, and C64x+ CPU

Compact Instruction Format

Unit	Opcode Format	Figure
.D	Doff4	Figure C-9
	Dind	Figure C-11
	Dinc	Figure C-13
	Ddec	Figure C-15

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 3-20 summarizes the data types supported by halfword loads. Table 3-6 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.

Table 3-20. 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

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.

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 2.8.3).

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.

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 → 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 instruction		**1 cycle after instruction**		**5 cycles after instruction**
A1	0000 0002h	A1	0000 0002h	A1	0000 0002h
A4	0000 0020h	A4	0000 0024h	A4	0000 0024h
A8	1103 51FFh	A8	1103 51FFh	A8	FFFF A21Fh
AMR	0000 0000h	AMR	0000 0000h	AMR	0000 0000h
mem 24h	A21Fh	mem 24h	A21Fh	mem 24h	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, and C64x+ 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 3-21 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*15offset 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 3-21. 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

LDNDW — *Load Nonaligned Doubleword From Memory With Constant or Register Offset*

Syntax

Register Offset

LDNDW (.unit) **+baseR[offsetR], dst*

unit = .D1 or .D2

Unsigned Constant Offset

LDNDW (.unit) **+baseR[ucst5], dst*

Compatibility C64x and C64x+ CPU

Compact Instruction Format

Unit	Opcode Format	Figure
.D	Doff4DW	Figure C-10
	DindDW	Figure C-12
	DincDW	Figure C-14
	DdecDW	Figure C-16

Opcode

31 29	28	27 24	23	22 18	17 13	12 9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*sc*	*baseR*	*offsetR/ucst5*	*mode*	1	*y*	0	1	0	0	1	*s*	*p*
3	1	4	1	5	5	4		1						1	1

Opcode map field used...	For operand type...	Unit
baseR *offsetR* *dst*	uint uint ullong	.D1, .D2
baseR *offsetR* *dst*	uint ucst5 ullong	.D1, .D2

Description Loads a 64-bit quantity from memory into a register pair, *dst_o:dst_e*. Table 3-6 describes the addressing generator options. The **LDNDW** instruction may read a 64-bit value from any byte boundary. Thus alignment to a 64-bit boundary is not required. The memory address is formed from a base address register (*baseR*) and an optional offset that is either a register (*offsetR*) or a 5-bit unsigned constant (*ucst5*).

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

The **LDNDW** instruction supports both scaled offsets and nonscaled offsets. The *sc* field is used to indicate whether the *offsetR/ucst5* is scaled or not. If *sc* is 1 (scaled), the *offsetR/ucst5* is shifted left 3 bits before adding or subtracting from the *baseR*. If *sc* is 0 (nonscaled), the *offsetR/ucst5* is not shifted before adding or subtracting from the *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 from memory.

The addressing arithmetic that performs the additions and subtractions defaults to linear mode. However, for A4-A7 and for B4-B7, the mode can be changed to circular mode by writing the appropriate value to the AMR (see Section 2.8.3).

The *dst* field of the instruction selects a register pair, a consecutive even-numbered and odd-numbered register pair from the same register file. The instruction can be used to load a pair of 32-bit integers. The 32 least-significant bits are loaded into the even-numbered register and the 32 most-significant bits are loaded into the next register (that is always an odd-numbered register).

The *dst* can be in either register file, regardless of the .D unit or *baseR* or *offsetR* used. The *s* bit determines which file *dst* will be loaded into: *s* = 0 indicates *dst* will be in the A register file and *s* = 1 indicates *dst* will be loaded in the B register file.

NOTE: No other memory access may be issued in parallel with a nonaligned memory access. The other .D unit can be used in parallel as long as it is not performing a memory access.

Assembler Notes

When no bracketed register or constant is specified, the assembler defaults increments and decrements to 1 and offsets to 0. Loads that do no modification to the *baseR* can use the assembler syntax *R. Square brackets, [ ], indicate that the *ucst5* offset is left-shifted by 3 for doubleword loads.

Parentheses, ( ), can be used to tell the assembler that the offset is a non-scaled offset.

For example, **LDNDW** (.unit) *+*baseR* (14), *dst* represents an offset of 14 bytes, and the assembler writes out the instruction with *offsetC* = 14 and *sc* = 0.

LDNDW (.unit) *+*baseR* [16], *dst* represents an offset of 16 doublewords, or 128 bytes, and the assembler writes out the instruction with *offsetC* = 16 and *sc* = 1.

Either brackets or parentheses must be typed around the specified offset if the optional offset parameter is used.

Execution

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

Pipeline

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

Instruction Type Load

Delay Slots 4 for loaded value

0 for address modification from pre/post increment/decrement

See Also LDNW, STNDW, STNW

Examples

Example 1

```
LDNDW .D1 *A0++, A3:A2
```

	Before instruction			1 cycle after instruction	
A0	0000 1001h		A0	0000 1009h	
A3:A2	xxxx xxxxh	xxxx xxxxh	A3:A2	xxxx xxxxh	xxxx xxxxh
mem 1000h	12B6 C5D4h		mem 1000h	12B6 C5D4h	
mem 1004h	1C4F 29A8h		mem 1004h	1C4F 29A8h	
mem 1008h	0569 345Eh		mem 1008h	0569 345Eh	

	5 cycles after instruction	
A0	0000 1009h	
A3:A2	5E1C 4F29h	A812 B6C5h
	Little-endian mode	
mem 1000h	12B6 C5D4h	
mem 1004h	1C4F 29A8h	
mem 1008h	0569 345Eh	

Byte Memory Address	**100C**	**100B**	**100A**	**1009**	**1008**	**1007**	**1006**	**1005**	**1004**	**1003**	**1002**	**1001**	**1000**
Data Value	11	05	69	34	5E	1C	4F	29	A8	12	B6	C5	D4

Example 2

```
LDNDW .D1 *A0++, A3:A2
```

	Before instruction	
A0	0000 1003h	
A3:A2	xxxx xxxxh	xxxx xxxxh
mem 1000h	12B6 C5D4h	
mem 1004h	1C4F 29A8h	
mem 1008h	0569 345Eh	

	1 cycle after instruction	
A0	0000 100Bh	
A3:A2	xxxx xxxxh	xxxx xxxxh
mem 1000h	12B6 C5D4h	
mem 1004h	1C4F 29A8h	
mem 1008h	0569 345Eh	

	5 cycles after instruction	
A0	0000 100Bh	
A3:A2	6934 5E1Ch	4F29 A812h
	Little-endian mode	
mem 1000h	12B6 C5D4h	
mem 1004h	1C4F 29A8h	
mem 1008h	0569 345Eh	

Byte Memory Address	**100C**	**100B**	**100A**	**1009**	**1008**	**1007**	**1006**	**1005**	**1004**	**1003**	**1002**	**1001**	**1000**
Data Value	11	05	69	34	5E	1C	4F	29	A8	12	B6	C5	D4

LDNW *Load Nonaligned Word From Memory With Constant or Register Offset*

Syntax

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

unit = .D1 or .D2

Compatibility C64x and C64x+ CPU

Compact Instruction Format

Unit	Opcode Format	Figure
.D	Doff4	Figure C-9
	Dind	Figure C-11
	Dinc	Figure C-13
	Ddec	Figure C-15

Opcode

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

Opcode map field used...	For operand type...	Unit
baseR *offset* *dst*	uint uint int	.D1, .D2
baseR *offset* *dst*	uint ucst5 int	.D1, .D2

Description Loads a 32-bit quantity from memory into a 32-bit register, *dst*. Table 3-6 describes the addressing generator options. The **LDNW** instruction may read a 32-bit value from any byte boundary. Thus alignment to a 32-bit boundary is not required. The memory address is formed from a base address register (*baseR*), and an optional offset that is either a register (*offsetR*) or a 5-bit unsigned constant (*ucst5*). If an offset is not given, the assembler assigns an offset of zero.

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

The *offsetR/ucst5* is scaled by a left shift of 2 bits. After scaling, *offsetR/ucst5* is added to, or subtracted from, *baseR*. For the 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 from memory.

The addressing arithmetic that performs the additions and subtractions defaults to linear mode. However, for A4-A7 and for B4-B7, the mode can be changed to circular mode by writing the appropriate value to the AMR (see Section 2.8.3).

The *dst* can be in either register file, regardless of the .D unit or *baseR* or *offsetR* used. The *s* bit determines which file *dst* will be loaded into: *s* = 0 indicates *dst* will be in the A register file and *s* = 1 indicates *dst* will be loaded in the B register file.

NOTE: No other memory access may be issued in parallel with a nonaligned memory access. The other .D unit can be used in parallel, as long as it is not doing a memory access.

Assembler Notes

When no bracketed register or constant is specified, the assembler defaults increments and decrements to 1 and offsets to 0. Loads that do no modification to the *baseR* can use the assembler syntax *R. Square brackets, [ ], indicate that the *ucst5* offset is left-shifted by 2 for word loads.

Parentheses, ( ), can be used to tell the assembler that the offset is a nonscaled, constant offset. The assembler right shifts the constant by 2 bits for word loads before using it for the *ucst5* field. After scaling by the **LDNW** instruction, this results in the same constant offset as the assembler source if the least-significant two bits are zeros.

For example, **LDNW** (.unit) *+*baseR* (12), *dst* represents an offset of 12 bytes (3 words), and the assembler writes out the instruction with *ucst5* = 3.

LDNW (.unit) *+*baseR* [12], *dst* represents an offset of 12 words, or 48 bytes, and the assembler writes out the instruction with *ucst5* = 12.

Either brackets or parentheses must be typed around the specified offset if the optional offset parameter is used.

Execution

if (cond)	mem → *dst*
else nop	

Pipeline

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

Instruction Type Load

Delay Slots 4 for loaded value

0 for address modification from pre/post increment/decrement

See Also LDNDW, STNDW, STNW

Examples

Example 1

```
LDNW .D1 *A0++, A2
```

	Before instruction		1 cycle after instruction		5 cycles after instruction
A0	0000 1001h	A0	0000 1005h	A0	0000 1005h
A2	xxxx xxxxh	A2	xxxx xxxxh	A2	A812 B6C5h
					Little-endian mode
mem 1000h	12B6 C5D4h	mem 1000h	12B6 C5D4h	mem 1000h	12B6 C5D4h
mem 1004h	1C4F 29A8h	mem 1004h	1C4F 29A8h	mem 1004h	1C4F 29A8h

Byte Memory Address	**1007**	**1006**	**1005**	**1004**	**1003**	**1002**	**1001**	**1000**
Data Value	1C	4F	29	A8	12	B6	C5	D4

Example 2

```
LDNW .D1 *A0++, A2
```

	Before instruction		1 cycle after instruction		5 cycles after instruction
A0	0000 1003h	A0	0000 1007h	A0	0000 1007h
A2	xxxx xxxxh	A2	xxxx xxxxh	A2	4F29 A812h
					Little-endian mode
mem 1000h	12B6 C5D4h	mem 1000h	12B6 C5D4h	mem 1000h	12B6 C5D4h
mem 1004h	1C4F 29A8h	mem 1004h	1C4F 29A8h	mem 1004h	1C4F 29A8h

Byte Memory Address	**1007**	**1006**	**1005**	**1004**	**1003**	**1002**	**1001**	**1000**
Data Value	1C	4F	29	A8	12	B6	C5	D4

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

Syntax

Register Offset

LDW (.unit) **+baseR[offsetR], dst*

unit = .D1 or .D2

Unsigned Constant Offset

LDW (.unit) **+baseR[ucst5], dst*

Compatibility C62x, C64x, and C64x+ CPU

Compact Instruction Format

Unit	Opcode Format	Figure
.D	Doff4	Figure C-9
	Dind	Figure C-11
	Dinc	Figure C-13
	Ddec	Figure C-15

Opcode

31 29	28	27 23	22 18	17 13	12 9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*baseR*	*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 3-6 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 2.8.3).

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.

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, LDH

Examples

Example 1

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

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

Example 2

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

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

Example 3

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

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

Example 4

```
LDW .D1 *++A4[A12],A8
```

	Before instruction		1 cycle after instruction		5 cycles after instruction
A4	0000 40B0h	A4	0000 40C8h	A4	0000 40C8h
A8	0000 0000h	A8	0000 0000h	A8	DCCB BAA8h
A12	0000 0006h	A12	0000 0006h	A12	0000 0006h
AMR	0000 0000h	AMR	0000 0000h	AMR	0000 0000h
mem 40C8h	DCCB BAA8h	mem 40C8h	DCCB BAA8h	mem 40C8h	DCCB BAA8h

Example 5

```
LDW .D1 *++A4(8),A8
```

	Before instruction		1 cycle after instruction		5 cycles after instruction
A4	0000 40B0h	A4	0000 40B8h	A4	0000 40B8h
A8	0000 0000h	A8	0000 0000h	A8	9AAB BCCDh
AMR	0000 0000h	AMR	0000 0000h	AMR	0000 0000h
mem 40B8h	9AAB BCCDh	mem 40B8h	9AAB BCCDh	mem 40B8h	9AAB BCCDh

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

Syntax

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

unit = .D2

Compatibility C62x, C64x, and C64x+ CPU

Compact Instruction Format

Unit	Opcode Format	Figure
.D	Dstk	Figure C-17
	Dpp	Figure C-22

Opcode

31 29	28	27 23	22 8	7	6	5	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*15offset 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

LL *Load Linked Word from Memory*

Syntax **LL** (.unit) **baseR, dst*

unit = .D2

Compatibility C64x+ CPU

NOTE: The atomic operations are not supported on all C64x+ devices, see your device-specific data manual for more information.

Opcode

31 29	28	27 23	22 18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*baseR*	0	0	0	0	0	0	0	1	1	0	0	1	0	0	0	0	1	*p*
3	1	5	5																		1

Opcode map field used...	For operand type...	Unit
baseR *dst*	address int	.D2

Description The **LL** instruction performs a read of the 32-bit word in memory at the address specified by *baseR*. The result is placed in *dst*. For linked-operation aware systems, the read request also results in a request to store the address specified by *baseR* in a linked operation register and the CPU signals that this is a linked read operation by setting the link valid flag. Other than this signaling, the operation of the **LL** instruction from the CPU perspective is identical to that of LDW **baseR, dst*.

See Chapter 9 for more details.

Execution

```
if (cond)    mem → dst
             signal load-linked operation
else nop
```

Instruction Type Load

Delay Slots 4

See Also CMTL, SL

LMBD *Leftmost Bit Detection*

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

unit = .L1 or .L2

Compatibility C62x, C64x, and C64x+ 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:

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

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

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

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

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)    {
             if ($src1_0$ == 0), lmb0($src2$) $\rightarrow$ $dst$
             if ($src1_0$ == 1), lmb1($src2$) $\rightarrow$ $dst$
             }
else nop
```

Pipeline

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

Instruction Type Single-cycle

Delay Slots 0

Example `LMBD .L1 A1,A2,A3`

	Before instruction		**1 cycle after instruction**
A1	0000 0001h	A1	0000 0001h
A2	009E 3A81h	A2	009E 3A81h
A3	xxxx xxxxh	A3	0000 0008h

MAX2 *Maximum, Signed, Packed 16-Bit*

Syntax **MAX2** (.unit) *src1, src2, dst*

unit = .L1 or .L2 (C64x and C64x+ CPU)

unit = .S1 or .S2 (C64x+ CPU)

Compatibility C64x and C64x+ CPU

Opcode .L unit (C64x and C64x+ CPU)

31 29	28	27 23	22 18	17 13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	1	0	0	0	0	1	0	1	1	0	*s*	*p*
3	1	5	5	5	1											1	1

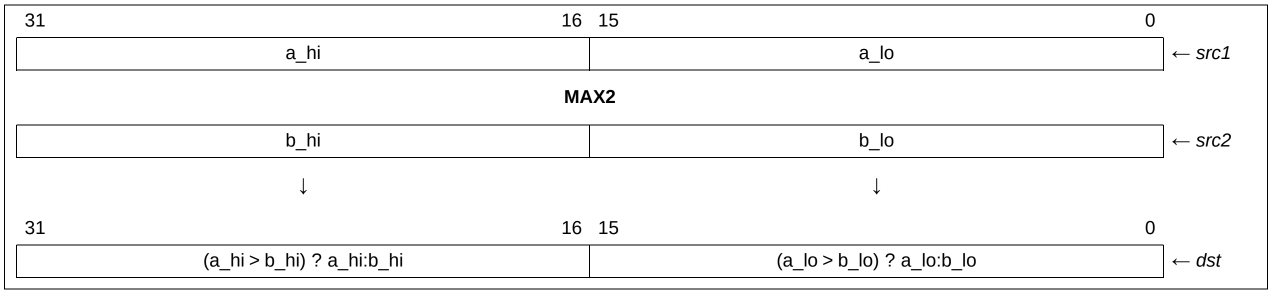
Opcode map field used...	For operand type...	Unit
src1	s2	.L1, .L2
src2	xs2	
dst	s2	

Opcode .S unit (C64x+ CPU)

31 29	28	27 23	22 18	17 13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	1	1	1	1	0	1	1	1	0	0	*s*	*p*
3	1	5	5	5	1											1	1

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

Description Performs a maximum operation on signed, packed 16-bit values. For each pair of signed 16-bit values in *src1* and *src2*, **MAX2** places the larger value in the corresponding position in *dst*.



Execution

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

Pipeline

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

Instruction Type Single-cycle

Delay Slots 0

See Also MAXU4, MIN2, MINU4

Examples

Example 1

```
MAX2 .L1 A2, A8, A9
```

	Before instruction		1 cycle after instruction	
A2	3789 F23Ah	A2	3789 F23Ah	14217 -3526
A8	04B8 4975h	A8	04B8 4975h	1208 18805
A9	xxxx xxxxh	A9	3789 4975h	14217 18805

Example 2

```
MAX2 .L2X A2, B8, B12
```

	Before instruction		1 cycle after instruction	
A2	0124 2451h	A2	0124 2451h	292 9297
B8	01A6 A051h	B8	01A6 A051h	422 -24495
B12	xxxx xxxxh	B12	01A6 2451h	422 9297

Example 3 (C64x+ CPU)

```
MAX2 .S1 A2, A8, A9
```

	Before instruction		1 cycle after instruction	
A2	3789 F23Ah	A2	3789 F23Ah	14217 -3526
A8	04B8 4975h	A8	04B8 4975h	1208 18805
A9	xxxx xxxxh	A9	3789 4975h	14217 18805

Example 4 (C64x+ CPU)

```
MAX2 .S2X A2, B8, B12
```

	Before instruction		1 cycle after instruction	
A2	0124 2451h	A2	0124 2451h	292 9297
B8	01A6 A051h	B8	01A6 A051h	422 -24495
B12	xxxx xxxxh	B12	01A6 2451h	422 9297

MAXU4 *Maximum, Unsigned, Packed 8-Bit*

Syntax **MAXU4** (.unit) *src1, src2, dst*

unit = .L1 or .L2

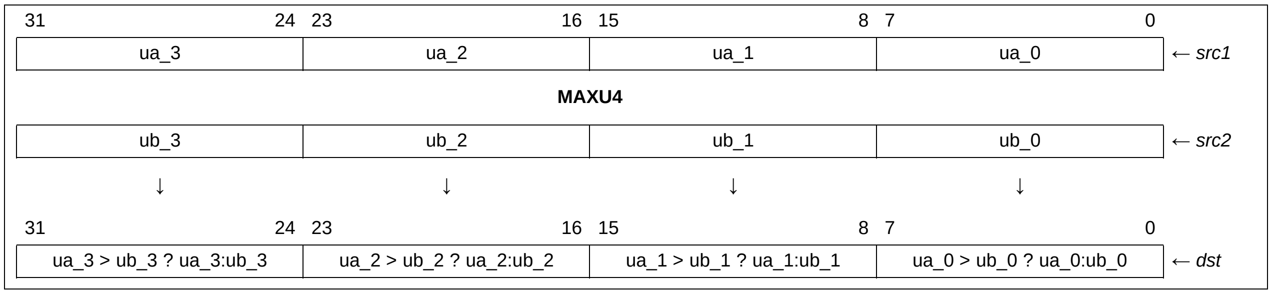
Compatibility C64x and C64x+ CPU

Opcode

31–29	28	27–23	22–18	17–13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	1	0	0	0	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	u4	.L1, .L2
src2	xu4	
dst	u4	

Description Performs a maximum operation on unsigned, packed 8-bit values. For each pair of unsigned 8-bit values in *src1* and *src2*, **MAXU4** places the larger value in the corresponding position in *dst*.



Execution

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

Pipeline

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

Instruction Type Single-cycle

Delay Slots 0

See Also MAX2, MIN2, MINU4

Examples

Example 1

```
MAXU4 .L1 A2, A8, A9
```

	Before instruction		1 cycle after instruction	
A2	37 89 F2 3Ah	A2	37 89 F2 3Ah	55 137 242 58 unsigned
A8	04 B8 49 75h	A8	04 B8 49 75h	4 184 73 117 unsigned
A9	xxxx xxxxh	A9	37 B8 F2 75h	55 184 242 117 unsigned

Example 2

```
MAXU4 .L2X A2, B8, B12
```

	Before instruction		1 cycle after instruction	
A2	01 24 24 B9h	A2	01 24 24 B9h	1 36 36 185 unsigned
B8	01 A6 A0 51h	B8	01 A6 A0 51h	1 166 160 81 unsigned
B12	xxxx xxxxh	B12	01 A6 A0 B9h	1 166 160 185 unsigned

MIN2 *Minimum, Signed, Packed 16-Bit*

Syntax **MIN2** (.unit) *src1, src2, dst*

unit = .L1 or .L2 (C64x and C64x+ CPU)

unit = .S1 or .S2 (C64x+ CPU)

Compatibility C64x and C64x+ CPU

Opcode .L unit (C64x and C64x+ CPU)

31–29	28	27–23	22–18	17–13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	1	0	0	0	0	0	1	1	1	0	*s*	*p*
3	1	5	5	5	1											1	1

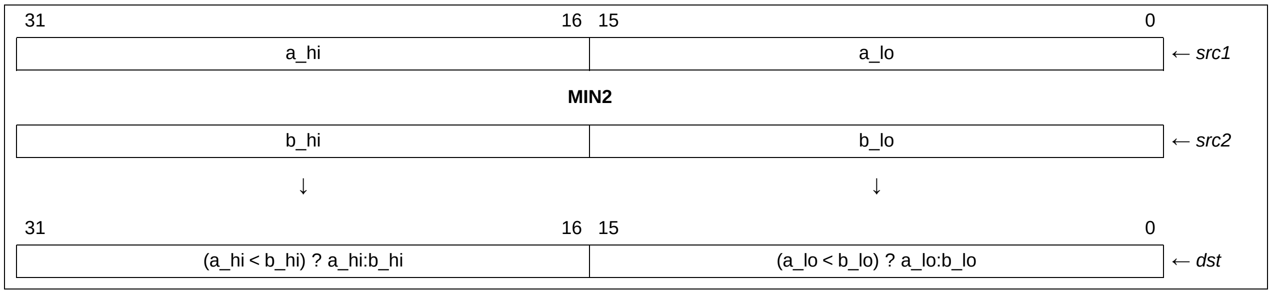
Opcode map field used...	For operand type...	Unit
src1	s2	.L1, .L2
src2	xs2	
dst	s2	

Opcode .S unit (C64x+ CPU)

31–29	28	27–23	22–18	17–13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	1	1	1	1	0	0	1	1	0	0	*s*	*p*
3	1	5	5	5	1											1	1

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

Description Performs a minimum operation on signed, packed 16-bit values. For each pair of signed 16-bit values in *src1* and *src2*, **MIN2** instruction places the smaller value in the corresponding position in *dst*.



Execution

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

Pipeline

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

Instruction Type Single-cycle

Delay Slots 0

See Also MAX2, MAXU4, MINU4

Examples

Example 1

```
MIN2 .L1 A2, A8, A9
```

	Before instruction		1 cycle after instruction	
A2	3789 F23Ah	A2	3789 F23Ah	14217 -3526
A8	04B8 4975h	A8	04B8 4975h	1208 18805
A9	xxxx xxxxh	A9	04B8 F23Ah	1208 -3526

Example 2

```
MIN2 .L2X A2, B8, B12
```

	Before instruction		1 cycle after instruction	
A2	0124 8003h	A2	0124 8003h	292 -32765
B8	0A37 8001h	B8	0A37 8001h	2615 -32767
B12	xxxx xxxxh	B12	0124 8001h	292 -32767

Example 3 (C64x+ CPU)

```
MIN2 .S1 A2, A8, A9
```

	Before instruction		1 cycle after instruction	
A2	3789 F23Ah	A2	3789 F23Ah	14217 -3526
A8	04B8 4975h	A8	04B8 4975h	1208 18805
A9	xxxx xxxxh	A9	04B8 F23Ah	1208 -3526

Example 4 (C64x+ CPU)

```
MIN2 .S2X A2, B8, B12
```

	Before instruction		1 cycle after instruction	
A2	0124 8003h	A2	0124 8003h	292 -32765
B8	0A37 8001h	B8	0A37 8001h	2615 -32767
B12	xxxx xxxxh	B12	0124 8001h	292 -32767

MINU4 *Minimum, Unsigned, Packed 8-Bit*

Syntax **MINU4** (.unit) *src1, src2, dst*

unit = .L1 or .L2

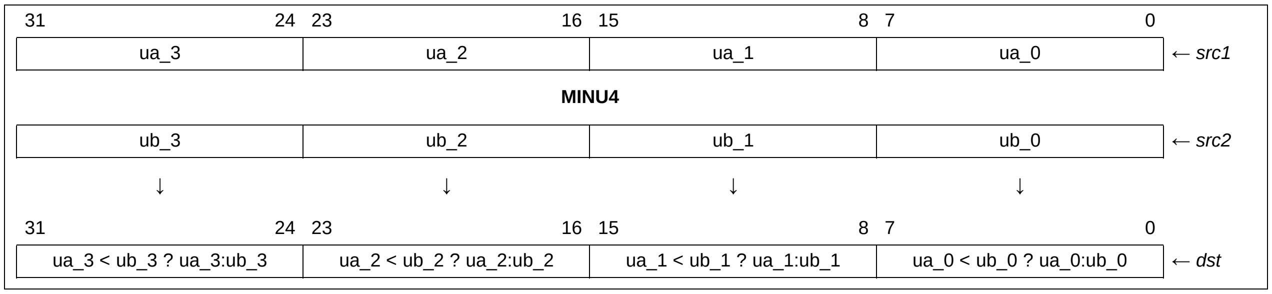
Compatibility C64x and C64x+ CPU

Opcode

31 29	28	27 23	22 18	17 13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	1	0	0	1	0	0	0	1	1	0	*s*	*p*
3	1	5	5	5	1											1	1

Opcode map field used...	For operand type...	Unit
src1	u4	.L1, .L2
src2	xu4	
dst	u4	

Description Performs a minimum operation on unsigned, packed 8-bit values. For each pair of unsigned 8-bit values in *src1* and *src2*, **MINU4** places the smaller value in the corresponding position in *dst*.



Execution

```
if (cond)       {
                if (ubyte0(src1) <= ubyte0(src2)), ubyte0(src1) → ubyte0(dst)
                    else ubyte0(src2) → ubyte0(dst);
                if (ubyte1(src1) <= ubyte1(src2)), ubyte1(src1) → ubyte1(dst)
                    else ubyte1(src2) → ubyte1(dst);
                if (ubyte2(src1) <= ubyte2(src2)), ubyte2(src1) → ubyte2(dst)
                    else ubyte2(src2) → ubyte2(dst);
                if (ubyte3(src1) <= ubyte3(src2)), ubyte3(src1) → ubyte3(dst)
                    else ubyte3(src2) → ubyte3(dst)
                }
else nop
```

Pipeline

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

Instruction Type Single-cycle

Delay Slots 0

See Also MAX2, MAXU4, MIN2

Examples

Example 1

```
MINU4 .L1 A2, A8, A9
```

	Before instruction		1 cycle after instruction	
A2	37 89 F2 3Ah	A2	37 89 F2 3Ah	55 137 242 58 unsigned
A8	04 B8 49 75h	A8	04 B8 49 75h	4 184 73 117 unsigned
A9	xxxx xxxxh	A9	04 89 49 3Ah	4 137 73 58 unsigned

Example 2

```
MINU4 .L2 B2, B8, B12
```

	Before instruction		1 cycle after instruction	
B2	01 24 24 B9h	B2	01 24 24 B9h	1 36 36 185 unsigned
B8	01 A6 A0 51h	B8	01 A6 A0 51h	1 166 160 81 unsigned
B12	xxxx xxxxh	B12	01 24 24 51h	1 36 36 81 unsigned

MPY *Multiply Signed 16 LSB × Signed 16 LSB*

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

unit = .M1 or .M2

Compatibility C62x, C64x, and C64x+ CPU

Compact Instruction Format

Unit	Opcode Format	Figure
.M	M3	Figure E-5

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

Examples

Example 1

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

	Before instruction			2 cycles after instruction	
A1	0000 0123h	291[1]	A1	0000 0123h	
A2	01E0 FA81h	-1407[1]	A2	01E0 FA81h	
A3	xxxx xxxxh		A3	FFF9 C0A3h	-409,437

[1] Signed 16-LSB integer

Example 2

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

	Before instruction			2 cycles after instruction	
A1	3497 FFF3h	-13[1]	A1	3497 FFF3h	
A2	xxxx xxxxh		A2	FFFF FF57h	-169

[1] Signed 16-LSB integer

MPYH *Multiply Signed 16 MSB × Signed 16 MSB*

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

unit = .M1 or .M2

Compatibility C62x, C64x, and C64x+ CPU

Compact Instruction Format

Unit	Opcode Format	Figure
.M	M3	Figure E-5

Opcode

31–29	28	27–23	22–18	17–13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*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 *src2* *dst*	smsb16 xsmsb16 sint	.M1, .M2

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[1]	A1	0023 0000h	
A2	FFA7 1234h	-89[1]	A2	FFA7 1234h	
A3	xxxx xxxxh		A3	FFFF F3D5h	-3115

[1] Signed 16-MSB integer

MPYHI *Multiply 16 MSB × 32-Bit Into 64-Bit Result*

Syntax **MPYHI** (.unit) *src1, src2, dst_o:dst_e*

unit = .M1 or .M2

Compatibility C64x and C64x+ CPU

Opcode

31–29	28	27–23	22–18	17–13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	0	1	0	1	0	0	1	1	0	0	*s*	*p*
3	1	5	5	5	1											1	1

Opcode map field used...	For operand type...	Unit
src1 *src2* *dst*	int xint sllong	.M1, .M2

Description Performs a 16-bit by 32-bit multiply. The upper half of *src1* is used as a signed 16-bit input. The value in *src2* is treated as a signed 32-bit value. The result is written into the lower 48 bits of a 64-bit register pair, *dst_o:dst_e*, and sign extended to 64 bits.

Execution

```
if (cond)    msb16(src1) × src2 → dst_o:dst_e
else nop
```

Pipeline

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

Instruction Type Four-cycle

Delay Slots 3

See Also MPYLI

Examples

Example 1

```
MPYHI .M1 A5,A6,A9:A8
```

	Before instruction			4 cycles after instruction	
A5	6A32 1193h	27,186	A5	6A32 1193h	
A6	B174 6CA4h	-1,317,770,076	A6	B174 6CA4h	
A9:A8	xxxx xxxxh	xxxx xxxxh	A9:A8	FFFF DF6Ah	DDB9 2008h
				-35,824,897,286,136	

Example 2

```
MPYHI .M2 B2,B5,B9:B8
```

	Before instruction			4 cycles after instruction	
B2	1234 3497h	4660	B2	1234 3497h	
B5	21FF 50A7h	570,380,455	B5	21FF 50A7h	
B9:B8	xxxx xxxxh	xxxx xxxxh	B9:B8	0000 026Ah	DB88 1FECh
				2,657,972,920,300	

MPYHIR *Multiply 16 MSB × 32-Bit, Shifted by 15 to Produce a Rounded 32-Bit Result*

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

unit = .M1 or .M2

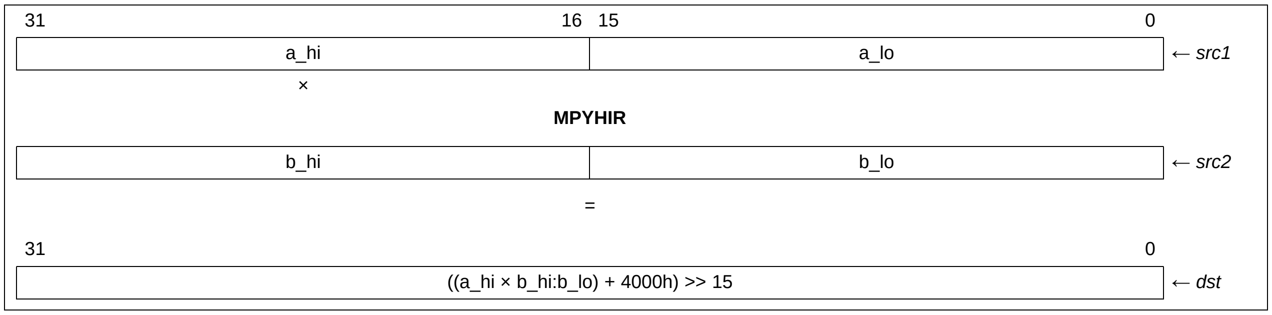
Compatibility C64x and C64x+ CPU

Opcode

31 29	28	27 23	22 18	17 13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	0	1	0	0	0	0	1	1	0	0	*s*	*p*
3	1	5	5	5	1											1	1

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

Description Performs a 16-bit by 32-bit multiply. The upper half of *src1* is treated as a signed 16-bit input. The value in *src2* is treated as a signed 32-bit value. The product is then rounded to a 32-bit result by adding the value 2^{14} and then this sum is right shifted by 15. The lower 32 bits of the result are written into *dst*.



Execution

if (cond) lsb32(((msb16(*src1*) × (*src2*)) + 4000h) >> 15) → *dst*
else nop

Pipeline

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

Instruction Type Four-cycle

Delay Slots 3

See Also MPYLIR

Example `MPYHIR .M2 B2,B5,B9`

	Before instruction			4 cycles after instruction	
B2	1234 3497h	4660	B2	1234 3497h	
B5	21FF 50A7h	570,380,455	B5	21FF 50A7h	
B9	xxxx xxxxh		B9	04D5 B710h	81,114,896

MPYHL *Multiply Signed 16 MSB × Signed 16 LSB*

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

unit = .M1 or .M2

Compatibility C62x, C64x, and C64x+ CPU

Compact Instruction Format

Unit	Opcode Format	Figure
.M	M3	Figure E-5

Opcode

31 29	28	27 23	22 18	17 13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*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 *src2* *dst*	smsb16 xslsb16 sint	.M1, .M2

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[1]	A1	008A 003Eh	
A2	21FF 00A7h	167[2]	A2	21FF 00A7h	
A3	xxxx xxxxh		A3	0000 5A06h	23,046

[1] Signed 16-MSB integer
[2] 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, and C64x+ CPU

Opcode

31–29	28	27–23	22–18	17–13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*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	umsb16	.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)      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, and C64x+ CPU

Opcode

31–29	28	27–23	22–18	17–13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*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, and C64x+ CPU

Opcode

31 29	28	27 23	22 18	17 13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*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	smsb16	.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)     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[1]	A1	0023 0000h	
A2	FFA7 FFFFh	65,447[2]	A2	FFA7 FFFFh	
A3	xxxx xxxxh		A3	0022 F3D5h	2,290,645

[1] Signed 16-MSB integer
[2] 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, and C64x+ CPU

Opcode

31–29	28	27–23	22–18	17–13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*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	umsb16	.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)    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[1]	A1	0023 0000h	
A2	FFA7 1234h	65,447[1]	A2	FFA7 1234h	
A3	xxxx xxxxh		A3	0022 F3D5h	2,290,645[2]

(1) Unsigned 16-MSB integer
(2) 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, and C64x+ CPU

Opcode

31–29	28	27–23	22–18	17–13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*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 *src2* *dst*	umsb16 xslsb16 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) × 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, and C64x+ CPU

Opcode

31–29	28	27–23	22–18	17–13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*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	umsb16	.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)    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

MPYIH *Multiply 32-Bit × 16-MSB Into 64-Bit Result*

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

unit = .M1 or .M2

Compatibility C64x and C64x+ CPU

Opcode

31 29	28	27 23	22 18	17 13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	0	1	0	1	0	0	1	1	0	0	*s*	*p*
3	1	5	5	5	1											1	1

Opcode map field used...	For operand type...	Unit
src1	int	.M1, .M2
src2	xint	
dst	sllong	

Description The **MPYIH** pseudo-operation performs a 16-bit by 32-bit multiply. The upper half of *src1* is used as a signed 16-bit input. The value in *src2* is treated as a signed 32-bit value. The result is written into the lower 48 bits of a 64-bit register pair, *dst_o:dst_e*, and sign extended to 64 bits. The assembler uses the **MPYHI** (.unit) *src1*, *src2*, *dst* instruction to perform this operation (see MPYHI).

Execution

```
if (cond)    src2 × msb16(src1) → dst_o:dst_e
else nop
```

Pipeline

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

Instruction Type Four-cycle

Delay Slots 3

See Also MPYHI, MPYIL

MPYIHR *Multiply 32-Bit × 16 MSB, Shifted by 15 to Produce a Rounded 32-Bit Result*

Syntax

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

unit = .M1 or .M2

Compatibility

C64x and C64x+ CPU

Opcode

31 29	28	27 23	22 18	17 13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	0	1	0	0	0	0	1	1	0	0	*s*	*p*
3	1	5	5	5	1											1	1

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

Description

The **MPYIHR** pseudo-operation performs a 16-bit by 32-bit multiply. The upper half of *src1* is treated as a signed 16-bit input. The value in *src2* is treated as a signed 32-bit value. The product is then rounded to a 32-bit result by adding the value 2^{14} and then this sum is right shifted by 15. The lower 32 bits of the result are written into *dst*. The assembler uses the **MPYHIR** (.unit) *src1*, *src2*, *dst* instruction to perform this operation (see MPYHIR).

Execution

```
if (cond)    lsb32((((src2) × msb16(src1)) + 4000h) >> 15) → dst
else nop
```

Pipeline

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

Instruction Type Four-cycle

Delay Slots 3

See Also MPYHIR, MPYILR

MPYIL *Multiply 32-Bit × 16 LSB Into 64-Bit Result*

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

unit = .M1 or .M2

Compatibility C64x and C64x+ CPU

Opcode

31 29	28	27 23	22 18	17 13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	0	1	0	1	0	1	1	1	0	0	*s*	*p*
3	1	5	5	5	1											1	1

Opcode map field used...	For operand type...	Unit
src1	int	.M1, .M2
src2	xint	
dst	sllong	

Description The **MPYIL** pseudo-operation performs a 16-bit by 32-bit multiply. The lower half of *src1* is used as a signed 16-bit input. The value in *src2* is treated as a signed 32-bit value. The result is written into the lower 48 bits of a 64-bit register pair, *dst_o:dst_e*, and sign extended to 64 bits. The assembler uses the **MPYLI** (.unit) *src1*, *src2*, *dst* instruction to perform this operation (see MPYLI).

Execution

```
if (cond)    src2 × lsb16(src1) → dst_o:dst_e
else nop
```

Pipeline

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

Instruction Type Four-cycle

Delay Slots 3

See Also MPYIH, MPYLI

MPYILR *Multiply 32-Bit × 16 LSB, Shifted by 15 to Produce a Rounded 32-Bit Result*

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

unit = .M1 or .M2

Compatibility C64x and C64x+ CPU

Opcode

31 29	28	27 23	22 18	17 13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	0	0	1	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	int	.M1, .M2
src2	xint	
dst	int	

Description The **MPYILR** pseudo-operation performs a 16-bit by 32-bit multiply. The lower half of *src1* is used as a signed 16-bit input. The value in *src2* is treated as a signed 32-bit value. The product is then rounded to a 32-bit result by adding the value 2^{14} and then this sum is right shifted by 15. The lower 32 bits of the result are written into *dst.* The assembler uses the **MPYLIR** (.unit) *src1*, *src2*, *dst* instruction to perform this operation (see MPYLIR).

Execution

if (cond)	lsb32((((*src2*) × lsb16(*src1*)) + 4000h) >> 15) → *dst*
else nop	

Pipeline

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

Instruction Type Four-cycle

Delay Slots 3

See Also MPYIHR, MPYLIR

MPYLH *Multiply Signed 16 LSB × Signed 16 MSB*

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

unit = .M1 or .M2

Compatibility C62x, C64x, and C64x+ CPU

Compact Instruction Format

Unit	Opcode Format	Figure
.M	M3	Figure E-5

Opcode

31 29	28	27 23	22 18	17 13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*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 *src2* *dst*	slsb16 xsmsb16 sint	.M1, .M2

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[1]	A1	0900 000Eh	
A2	0029 00A7h	41[2]	A2	0029 00A7h	
A3	xxxx xxxxh		A3	0000 023Eh	574

[1] Signed 16-LSB integer
[2] 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, and C64x+ CPU

Opcode

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

MPYLI *Multiply 16 LSB × 32-Bit Into 64-Bit Result*

Syntax

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

unit = .M1 or .M2

Compatibility C64x and C64x+ CPU

Opcode

31–29	28	27–23	22–18	17–13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	0	1	0	1	0	1	1	1	0	0	*s*	*p*
3	1	5	5	5	1											1	1

Opcode map field used...	For operand type...	Unit
src1	int	.M1, .M2
src2	xint	
dst	sllong	

Description Performs a 16-bit by 32-bit multiply. The lower half of *src1* is used as a signed 16-bit input. The value in *src2* is treated as a signed 32-bit value. The result is written into the lower 48 bits of a 64-bit register pair, *dst_o:dst_e*, and sign extended to 64 bits.

Execution

```
if (cond)    lsb16(src1) × src2 → dst_o:dst_e
else nop
```

Pipeline

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

Instruction Type Four-cycle

Delay Slots 3

See Also MPYHI

Examples

Example 1

```
MPYLI .M1 A5,A6,A9:A8
```

	Before instruction			4 cycles after instruction	
A5	6A32 1193h	4499	A5	6A32 1193h	
A6	B174 6CA4h	-1,317,770,076	A6	B174 6CA4h	
A9:A8	xxxx xxxxh	xxxx xxxxh	A9:A8	FFFF FA9Bh	A111 462Ch
				-5,928,647,571,924	

Example 2

```
MPYLI .M2 B2,B5,B9:B8
```

	Before instruction			4 cycles after instruction	
B2	1234 3497h	13,463	B2	1234 3497h	
B5	21FF 50A7h	570,380,455	B5	21FF 50A7h	
B9:B8	xxxx xxxxh	xxxx xxxxh	B9:B8	0000 06FBh	E9FA 7E81h
				7,679,032,065,665	

MPYLIR *Multiply 16 LSB × 32-Bit, Shifted by 15 to Produce a Rounded 32-Bit Result*

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

unit = .M1 or .M2

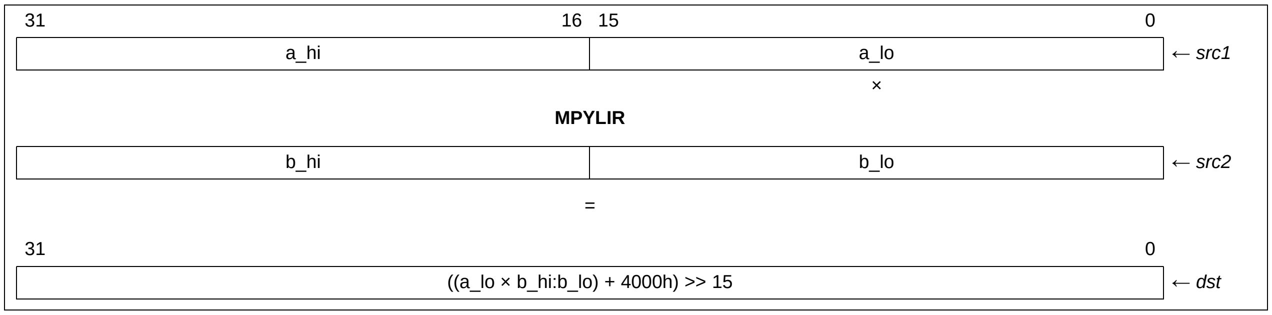
Compatibility C64x and C64x+ CPU

Opcode

31 29	28	27 23	22 18	17 13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	0	0	1	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	int	.M1, .M2
src2	xint	
dst	int	

Description Performs a 16-bit by 32-bit multiply. The lower half of *src1* is treated as a signed 16-bit input. The value in *src2* is treated as a signed 32-bit value. The product is then rounded into a 32-bit result by adding the value 2^{14} and then this sum is right shifted by 15. The lower 32 bits of the result are written into *dst.*



Execution

if (cond)	lsb32(((lsb16(*src1*) × (*src2*)) + 4000h) >> 15) → *dst*
else nop	

Pipeline

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

Instruction Type Four-cycle

Delay Slots 3

See Also MPYHIR

Example `MPYLIR .M2 B2,B5,B9`

	Before instruction			4 cycles after instruction	
B2	1234 3497h	13,463	B2	1234 3497h	
B5	21FF 50A7h	570,380,455	B5	21FF 50A7h	
B9	xxxx xxxxh		B9	0DF7 D3F5h	234,345,461

MPYLSHU *Multiply Signed 16 LSB × Unsigned 16 MSB*

Syntax

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

unit = .M1 or .M2

Compatibility C62x, C64x, and C64x+ CPU

Opcode

31–29	28	27–23	22–18	17–13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*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, and C64x+ CPU

Opcode

31 29	28	27 23	22 18	17 13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*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 *src2* *dst*	ulsb16 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)     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, MPYLSHU

MPYSU *Multiply Signed 16 LSB × Unsigned 16 LSB*

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

unit = .M1 or .M2

Compatibility C62x, C64x, and C64x+ 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	65,523[1]	A1	3497 FFF3h	
A2	xxxx xxxxh		A2	000C FF57h	851,779

[1] Unsigned 16-LSB integer

MPYSU4 *Multiply Signed × Unsigned, Four 8-Bit Pairs for Four 8-Bit Results*

Syntax **MPYSU4** (.unit) *src1, src2, dst*

unit = .M1 or .M2

Compatibility C64x and C64x+ CPU

Opcode

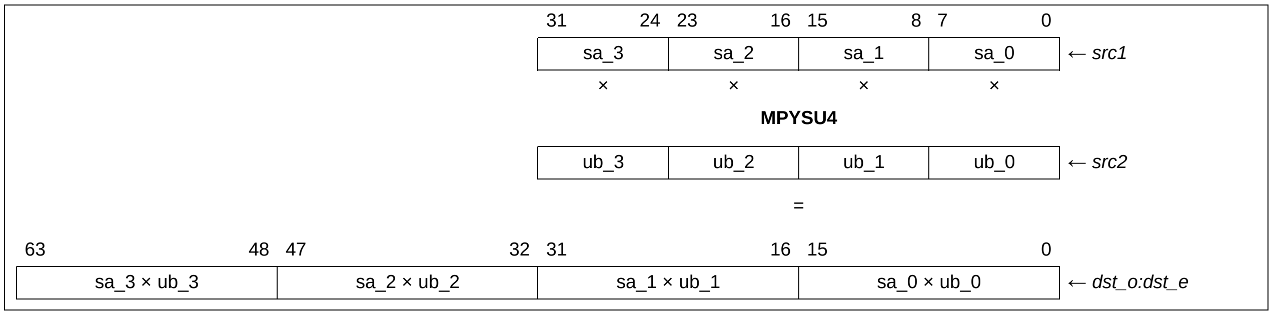
31–29	28	27–23	22–18	17–13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	0	0	0	1	0	1	1	1	0	0	*s*	*p*
3	1	5	5	5	1											1	1

Opcode map field used...	**For operand type...**	**Unit**
src1	s4	.M1, .M2
src2	xu4	
dst	dws4	

Description Returns the product between four sets of packed 8-bit values producing four signed 16-bit results. The four signed 16-bit results are packed into a 64-bit register pair, *dst_o:dst_e*. The values in *src1* are treated as signed 8-bit packed quantities; whereas, the values in *src2* are treated as unsigned 8-bit packed data.

For each pair of 8-bit quantities in *src1* and *src2*, the signed 8-bit value from *src1* is multiplied with the unsigned 8-bit value from *src2*:

- The product of *src1* byte 0 and *src2* byte 0 is written to the lower half of *dst_e*.
- The product of *src1* byte 1 and *src2* byte 1 is written to the upper half of *dst_e*.
- The product of *src1* byte 2 and *src2* byte 2 is written to the lower half of *dst_o*.
- The product of *src1* byte 3 and *src2* byte 3 is written to the upper half of *dst_o*.



Execution

```
if (cond)    {
             (sbyte0(src1) × ubyte0(src2)) → lsb16(dst_e);
             (sbyte1(src1) × ubyte1(src2)) → msb16(dst_e);
             (sbyte2(src1) × ubyte2(src2)) → lsb16(dst_o);
             (sbyte3(src1) × ubyte3(src2)) → msb16(dst_o)
             }
else nop
```

Pipeline

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

Instruction Type Four-cycle

Delay Slots 3

See Also MPYU4

Examples

Example 1

```
MPYSU4 .M1 A5,A6,A9:A8
```

	Before instruction			4 cycles after instruction	
A5	6A 32 11 93h	106 50 17 -109 signed	A5	6A 32 11 93h	
A6	B1 74 6C A4h	177 116 108 164 unsigned	A6	B1 74 6C A4h	
A9:A8	xxxx xxxxh	xxxx xxxxh	A9:A8	494A 16A8h	072C BA2Ch
				18762 5800	1386 -17876 signed

Example 2

```
MPYSU4 .M2 B5,B6,B9:B8
```

	Before instruction			4 cycles after instruction	
B5	3F F6 50 10h	63 -10 80 16 signed	B5	3F F6 50 10h	
B6	C3 56 02 44h	195 86 2 68 unsigned	B6	C3 56 02 44h	
B9:B8	xxxx xxxxh	xxxx xxxxh	B9:B8	2FFD FCA4h	00A0 0440h
				12285 -680	160 1088 signed

MPYU *Multiply Unsigned 16 LSB × Unsigned 16 LSB*

Syntax

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

unit = .M1 or .M2

Compatibility

C62x, C64x, and C64x+ CPU

Opcode

31–29	28	27–23	22–18	17–13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*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[1]	A1	0000 0123h	
A2	0F12 FA81h	64,129[1]	A2	0F12 FA81h	
A3	xxxx xxxxh		A3	011C C0A3h	18,661,539[2]

[1] Unsigned 16-LSB integer
[2] Unsigned 32-bit integer

MPYU4 *Multiply Unsigned × Unsigned, Four 8-Bit Pairs for Four 8-Bit Results*

Syntax **MPYU4** (.unit) *src1, src2, dst_o:dst_e*

unit = .M1 or .M2

Compatibility C64x and C64x+ CPU

Opcode

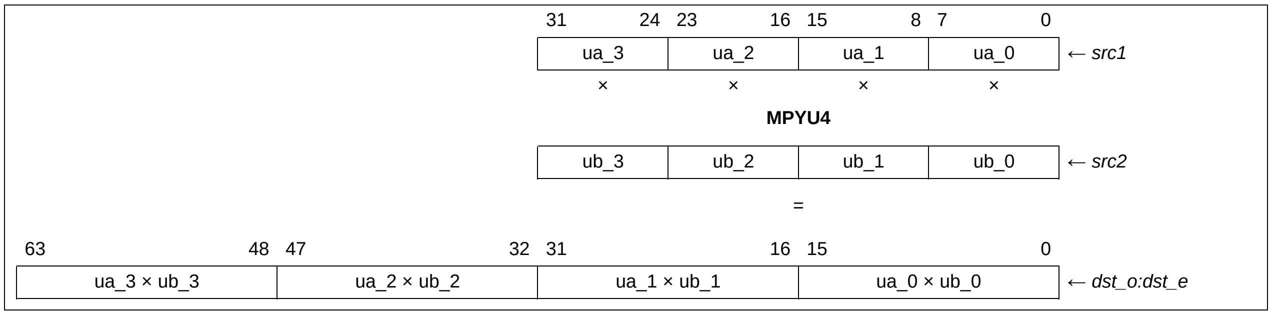
31 29	28	27 23	22 18	17 13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	0	0	0	1	0	0	1	1	0	0	*s*	*p*
3	1	5	5	5	1											1	1

Opcode map field used...	For operand type...	Unit
src1	u4	.M1, .M2
src2	xu4	
dst	dwu4	

Description Returns the product between four sets of packed 8-bit values producing four unsigned 16-bit results that are packed into a 64-bit register pair, *dst_o:dst_e*. The values in both *src1* and *src2* are treated as unsigned 8-bit packed data.

For each pair of 8-bit quantities in *src1* and *src2*, the unsigned 8-bit value from *src1* is multiplied with the unsigned 8-bit value from *src2*:

- The product of *src1* byte 0 and *src2* byte 0 is written to the lower half of *dst_e*.
- The product of *src1* byte 1 and *src2* byte 1 is written to the upper half of *dst_e*.
- The product of *src1* byte 2 and *src2* byte 2 is written to the lower half of *dst_o*.
- The product of *src1* byte 3 and *src2* byte 3 is written to the upper half of *dst_o*.



Execution

```
if (cond)      {
               (ubyte0(src1) × ubyte0(src2)) → lsb16(dst_e);
               (ubyte1(src1) × ubyte1(src2)) → msb16(dst_e);
               (ubyte2(src1) × ubyte2(src2)) → lsb16(dst_o);
               (ubyte3(src1) × ubyte3(src2)) → msb16(dst_o)
               }
else nop
```

Pipeline

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

Instruction Type Four-cycle

Delay Slots 3

See Also MPYSU4

Examples

Example 1

```
MPYU4 .M1 A5,A6,A9:A8
```

	Before instruction			4 cycles after instruction	
A5	68 32 C1 93h	104 50 193 147 unsigned	A5	68 32 C1 93h	
A6	B1 74 2C ABh	177 116 44 171 unsigned	A6	B1 74 2C ABh	
A9:A8	xxxx xxxxh	xxxx xxxxh	A9:A8	47E8 16A8h	212C 6231h
				18408 5800	8492 25137 unsigned

Example 2

```
MPYU4 .M2 B2,B5,B9:B8
```

	Before instruction			4 cycles after instruction	
B2	3D E6 50 7Fh	61 230 80 127 unsigned	B2	3D E6 50 7Fh	
B5	C3 56 02 44h	195 86 2 68 unsigned	B5	C3 56 02 44h	
B9:B8	xxxx xxxxh	xxxx xxxxh	B9:B8	2E77 4D44h	00A0 21BCh
				11895 19780	160 8636 unsigned

MPYUS *Multiply Unsigned 16 LSB × Signed 16 LSB*

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

unit = .M1 or .M2

Compatibility C62x, C64x, and C64x+ CPU

Opcode

31–29	28	27–23	22–18	17–13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*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	65,441[1]	A1	1234 FFA1h	
A2	1234 FFA1h	-95[2]	A2	1234 FFA1h	
A3	xxxx xxxxh		A3	FFA1 2341h	-6,216,895

(1) Unsigned 16-LSB integer
(2) Signed 16-LSB integer

MPYUS4 *Multiply Unsigned × Signed, Four 8-Bit Pairs for Four 8-Bit Results*

Syntax **MPYUS4** (.unit) *src2, src1, dst*

unit = .M1 or .M2

Compatibility C64x and C64x+ CPU

Opcode

31–29	28	27–23	22–18	17–13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	0	0	0	1	0	1	1	1	0	0	*s*	*p*
3	1	5	5	5	1											1	1

Opcode map field used...	For operand type...	Unit
src1	s4	.M1, .M2
src2	xu4	
dst	dws4	

Description The **MPYUS4** pseudo-operation returns the product between four sets of packed 8-bit values, producing four signed 16-bit results. The four signed 16-bit results are packed into a 64-bit register pair, *dst_o:dst_e*. The values in *src1* are treated as signed 8-bit packed quantities; whereas, the values in *src2* are treated as unsigned 8-bit packed data. The assembler uses the **MPYSU4** (.unit)*src1*, *src2*, *dst* instruction to perform this operation (see MPYSU4).

For each pair of 8-bit quantities in *src1* and *src2*, the signed 8-bit value from *src1* is multiplied with the unsigned 8-bit value from *src2*:

- The product of *src1* byte 0 and *src2* byte 0 is written to the lower half of *dst_e*.
- The product of *src1* byte 1 and *src2* byte 1 is written to the upper half of *dst_e*.
- The product of *src1* byte 2 and *src2* byte 2 is written to the lower half of *dst_o*.
- The product of *src1* byte 3 and *src2* byte 3 is written to the upper half of *dst_o*.

Execution

```
if (cond)    {
             (ubyte0(src2) × sbyte0(src1)) → lsb16(dst_e);
             (ubyte1(src2) × sbyte1(src1)) → msb16(dst_e);
             (ubyte2(src2) × sbyte2(src1)) → lsb16(dst_o);
             (ubyte3(src2) × sbyte3(src1)) → msb16(dst_o)
             }
else nop
```

Pipeline

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

Instruction Type Four-cycle

Delay Slots 3

See Also MPYSU4, MPYU4

MPY2 *Multiply Signed by Signed, 16 LSB × 16 LSB and 16 MSB × 16 MSB*

Syntax **MPY2** (.unit) *src1, src2, dst_o:dst_e*

unit = .M1 or .M2

Compatibility C64x and C64x+ CPU

Opcode

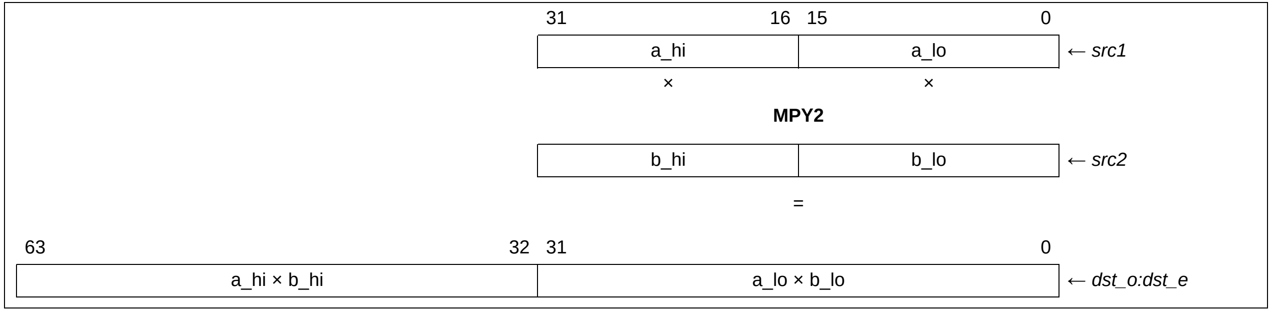
31–29	28	27–23	22–18	17–13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	0	0	0	0	0	0	1	1	0	0	*s*	*p*
3	1	5	5	5	1											1	1

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

Description Performs two 16-bit by 16-bit multiplications between two pairs of signed, packed 16-bit values. The values in *src1* and *src2* are treated as signed, packed 16-bit quantities. The two 32-bit results are written into a 64-bit register pair.

The product of the lower halfwords of *src1* and *src2* is written to the even destination register, *dst_e*. The product of the upper halfwords of *src1* and *src2* is written to the odd destination register, *dst_o*.

This instruction helps reduce the number of instructions required to perform two 16-bit by 16-bit multiplies on both the lower and upper halves of two registers.



The following code:

```
MPY    .M1   A0, A1, A2
MPYH   .M1   A0, A1, A3
```

may be replaced by:

```
MPY2   .M1   A0, A1, A3:A2
```

Execution

```
if (cond)    {
             lsb16(src1) × lsb16(src2) → dst_e;
             msb16(src1) × msb16(src2) → dst_o
             }
else nop
```

Pipeline

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

Instruction Type Four-cycle

Delay Slots 3

See Also MPYSU4, MPY2IR, SMPY2

Examples

Example 1

```
MPY2 .M1 A5,A6, A9:A8
```

	Before instruction			4 cycles after instruction	
A5	6A32 1193h	27186 4499	A5	6A32 1193h	
A6	B174 6CA4h	-20108 27812	A6	B174 6CA4h	
A9:A8	xxxx xxxxh	xxxx xxxxh	A9:A8	DF6A B0A8h	0775 462Ch
				-546,656,088	125,126,188

Example 2

```
MPY2 .M2 B2, B5, B9:B8
```

	Before instruction			4 cycles after instruction	
B2	1234 3497h	4660 13463	B2	1234 3497h	
B5	21FF 50A7h	8703 20647	B5	21FF 50A7h	
B9:B8	xxxx xxxxh	xxxx xxxxh	B9:B8	026A D5CCh	1091 7E81h
				40,555,980	277,970,561

MPY2IR *Multiply Two 16-Bit × 32-Bit, Shifted by 15 to Produce a Rounded 32-Bit Result*

Syntax

MPY2IR (.unit) *src1, src2, dst_o:dst_e*

unit = .M1 or .M2

Compatibility C64x+ CPU

Opcode

31	30	29	28	27 23	22 18	17 13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	*dst*	*src2*	*src1*	x	0	0	1	1	1	1	1	1	0	0	*s*	*p*
				5	5	5	1											1	1

Opcode map field used...	For operand type...	Unit
src1	int	.M1, .M2
src2	xint	
dst	dint	

Description

Performs two 16-bit by 32-bit multiplies. The upper and lower halves of *src1* are treated as 16-bit signed inputs. The value in *src2* is treated as a 32-bit signed value. The products are then rounded to a 32-bit result by adding the value 2^{14} and then these sums are right shifted by 15. The lower 32 bits of the two results are written into *dst_o:dst_e*.

If either result saturates, the M1 or M2 bit in SSR and the SAT bit in CSR are written one cycle after the results are written to *dst_o:dst_e*.

This instruction executes unconditionally and cannot be predicated.

NOTE: In the overflow case, where the 16-bit input to the **MPYIR** operation is 8000h and the 32-bit input is 8000 0000h, the saturation value 7FFF FFFFh is written into the corresponding 32-bit *dst* register.

Execution

```
if (msb16(src1) = 8000h && src2 = 8000 0000h), 7FFF FFFFh → dst_o
        else lsb32(((msb16(src1) × (src2)) + 4000h) >> 15) → dst_o;
if (lsb16(src1) = 8000h && src2 = 8000 0000h), 7FFF FFFFh → dst_e
        else lsb32(((lsb16(src1) × (src2)) + 4000h) >> 15) → dst_e
```

Instruction Type Four-cycle

Delay Slots 3

See Also MPYLIR, MPYHIR

Examples

Example 1

```
MPY2IR .M2 B2,B5,B9:B8
```

	Before instruction		4 cycles after instruction
B2	8000 8001h	B8	7FFF 0000h
B5	8000 0000h	B9	7FFF FFFFh
CSR	0001 0100h	CSR[1]	0001 0300h
SSR	0000 0000h	SSR[1]	0000 0020h

[1] CSR.SAT and SSR.M2 set to 1, 5 cycles after instruction

Example 2

```
MPY2IR .M1X A2,B5,A9:A8
```

	Before instruction		4 cycles after instruction
A2	8765 4321h	A8	098C 16C1h
B5	1234 5678h	A9	EED8 E38Fh
CSR	0001 0100h	CSR[1]	0001 0100h
SSR	0000 0000h	SSR[1]	0000 0000h

[1] CSR.SAT and SSR.M1 unchanged by operation

MPY32 — *Multiply Signed 32-Bit × Signed 32-Bit Into 32-Bit Result*

Syntax

MPY32 (.unit) *src1, src2, dst*

unit = .M1 or .M2

Compatibility C64x+ CPU

Opcode

31–29	28	27–23	22–18	17–13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	1	0	0	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	int	.M1, .M2
src2	xint	
dst	int	

Description Performs a 32-bit by 32-bit multiply. *src1* and *src2* are signed 32-bit values. Only the lower 32 bits of the 64-bit result are written to *dst*.

Execution

```
if (cond)    src1 × src2 → dst
else nop
```

Instruction Type Four-cycle

Delay Slots 3

See Also MPY32, MPY32SU, MPY32US, MPY32U, SMPY32

MPY32 *Multiply Signed 32-Bit × Signed 32-Bit Into Signed 64-Bit Result*

Syntax **MPY32** (.unit) *src1, src2, dst_o:dst_e*

unit = .M1 or .M2

Compatibility C64x+ CPU

Opcode

31 29	28	27 23	22 18	17 13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	1	0	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	int	.M1, .M2
src2	xint	
dst	dint	

Description Performs a 32-bit by 32-bit multiply. *src1* and *src2* are signed 32-bit values. The signed 64-bit result is written to the register pair specified by *dst*.

Execution

```
if (cond)    src1 × src2 → dst_o:dst_e
else nop
```

Instruction Type Four-cycle

Delay Slots 3

See Also MPY32, MPY32SU, MPY32US, MPY32U, SMPY32

MPY32SU *Multiply Signed 32-Bit × Unsigned 32-Bit Into Signed 64-Bit Result*

Syntax **MPY32SU** (.unit) *src1, src2, dst_o:dst_e*

unit = .M1 or .M2

Compatibility C64x+ CPU

Opcode

31–29	28	27–23	22–18	17–13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	1	0	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	int	.M1, .M2
src2	xuint	
dst	dint	

Description Performs a 32-bit by 32-bit multiply. *src1* is a signed 32-bit value and *src2* is an unsigned 32-bit value. The signed 64-bit result is written to the register pair specified by *dst*.

Execution

```
if (cond)     src1 × src2 → dst_o:dst_e
else nop
```

Instruction Type Four-cycle

Delay Slots 3

See Also MPY32, MPY32U, MPY32US, SMPY32

MPY32U *Multiply Unsigned 32-Bit × Unsigned 32-Bit Into Unsigned 64-Bit Result*

Syntax **MPY32U** (.unit) *src1, src2, dst_o:dst_e*

unit = .M1 or .M2

Compatibility C64x+ CPU

Opcode

31 29	28	27 23	22 18	17 13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	0	1	1	0	0	0	1	1	0	0	*s*	*p*
3	1	5	5	5	1											1	1

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

Description Performs a 32-bit by 32-bit multiply. *src1* and *src2* are unsigned 32-bit values. The unsigned 64-bit result is written to the register pair specified by *dst*.

Execution

```
if (cond)    src1 × src2 → dst_o:dst_e
else nop
```

Instruction Type Four-cycle

Delay Slots 3

See Also MPY32, MPY32SU, MPY32US, SMPY32

MPY32US *Multiply Unsigned 32-Bit × Signed 32-Bit Into Signed 64-Bit Result*

Syntax **MPY32US** (.unit) *src1, src2, dst_o:dst_e*

unit = .M1 or .M2

Compatibility C64x+ CPU

Opcode

31–29	28	27–23	22–18	17–13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	0	1	1	0	0	1	1	1	0	0	*s*	*p*
3	1	5	5	5	1											1	1

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

Description Performs a 32-bit by 32-bit multiply. *src1* is an unsigned 32-bit value and *src2* is a signed 32-bit value. The signed 64-bit result is written to the register pair specified by *dst*.

Execution

```
if (cond)    src1 × src2 → dst_o:dst_e
else nop
```

Instruction Type Four-cycle

Delay Slots 3

See Also MPY32, MPY32SU, MPY32U, SMPY32

MV *Move From Register to Register*

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

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

Compatibility C62x, C64x, and C64x+ CPU

Compact Instruction Format

Unit	Opcode Format	Figure
.L, .S, .D	LSDmvto	Figure G-1
	LSDmvtfr	Figure G-2

Opcode .L unit (if the cross path form is not used)

31 29	28	27 23	22 18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	0	0	0	0	0	0	0	1	0	0	0	0	0	1	1	0	*s*	*p*
3	1	5	5																	1	1

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

Opcode .L unit (if the cross path form is used)

31 29	28	27 23	22 18	17	16	15	14	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*	xuint uint	.L1, .L2	111 1110

Opcode .S unit

31 29	28	27 23	22 18	17	16	15	14	13	12	11	10	9	8	7	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 (if the cross path form is not used)

31 29	28	27 23	22 18	17	16	15	14	13	12	11	10	9	8	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
src2 *dst*	sint sint	.D1, .D2

Opcode .D unit (if the cross path form is used)

31 29	28	27 23	22 18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	0	0	0	0	0	x	1	0	0	0	1	1	1	1	0	0	*s*	*p*
3	1	5	5						1											1	1

Opcode map field used...	For operand type...	Unit
src2 *dst*	xuint uint	.D1, .D2

Description The **MV** pseudo-operation moves a value from one register to another. The assembler will either use the **ADD** (.unit) 0, *src2, dst* instruction (see ADD) or the **OR** (.unit) 0, *src2, dst* instruction (see OR) to perform this operation.

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, and C64x+ CPU

Compact Instruction Format

Unit	Opcode Format	Figure
.S	Sx1	Figure F-30

Opcode C64x and C64x+ CPU

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

31 29	28	27 23	22 18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*crlo*	0	0	0	0	0	x	0	0	1	1	1	1	1	0	0	0	1	*p*
3	1	5	5						1												1

Opcode map field used...	For operand type...	Unit
crlo *dst*	uint uint	.S2

Description For the C64x and C64x+ CPU, the contents of the control file specified by the *crlo* field is moved to the register file specified by the *dst* field.

Register addresses for accessing the control registers are in Table 3-22.

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

31 29	28	27 23	22 18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*crlo*	*src2*	0	0	0	0	0	x	0	0	1	1	1	0	1	0	0	0	1	*p*
3	1	5	5						1												1

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

Description For the C64x and C64x+ CPU, the contents of the register file specified by the *src2* field is moved to the control file specified by the *crlo* field.

Register addresses for accessing the control registers are in Table 3-22.

Opcode C64x+ CPU

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

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

Opcode map field used...	For operand type...	Unit
crlo	ucst5	.S2
dst	uint	
crhi	ucst5	

Description For the C64x+ CPU, the contents of the control file specified by the *crhi* and *crlo* fields is moved to the register file specified by the *dst* field. Valid assembler values for *crlo* and *crhi* are shown in Table 3-22.

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

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

Opcode map field used...	For operand type...	Unit
src2	xuint	.S2
crlo	ucst5	
crhi	ucst5	

Description For the C64x+ CPU, the contents of the register file specified by the *src2* field is moved to the control file specified by the *crhi* and *crlo* fields. Valid assembler values for *crlo* and *crhi* are shown in Table 3-22.

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 3-22. Register Addresses for Accessing the Control Registers

		Address			
Acronym	**Register Name**	*crhi*	*crlo*	**Supervisor Read/Write** [(1)]	**User Read/Write** [(1)]
AMR	Addressing mode register	00000	00000	R, W	R, W
		0xxxx	00000		
CSR	Control status register	00000	00001	R, W*	R, W*
		00001	00001		
		0xxxx	00001		
DIER	Debug interrupt enable register	00000	11001	R, W	X
DNUM	DSP core number register	00000	10001	R	R
ECR	Exception clear register	00000	11101	W	X
EFR	Exception flag register	00000	11101	R	X
GFPGFR	Galois field multiply control register	00000	11000	R, W	R, W
GPLYA	GMPY A-side polynomial register	00000	10110	R, W	R, W
GPLYB	GMPY B-side polynomial register	00000	10111	R, W	R, W
ICR	Interrupt clear register	00000	00011	W	X
		0xxxx	00011		
IER	Interrupt enable register	00000	00100	R, W	X
		0xxxx	00100		
IERR	Internal exception report register	00000	11111	R,W	X
IFR	Interrupt flag register	00000	00010	R	X
		00010	00010		
ILC	Inner loop count register	00000	01101	R, W	R, W
IRP	Interrupt return pointer register	00000	00110	R, W	R, W
		0xxxx	00110		
ISR	Interrupt set register	00000	00010	W	X
		0xxxx	00010		
ISTP	Interrupt service table pointer register	00000	00101	R, W	X
		0xxxx	00101		
ITSR	Interrupt task state register	00000	11011	R, W	X
NRP	Nonmaskable interrupt or exception return pointer register	00000	00111	R, W	R, W
		0xxxx	00111		
NTSR	NMI/Exception task state register	00000	11100	R, W	X
PCE1	Program counter, E1 phase	00000	10000	R	R
		10000	10000		
REP	Restricted entry point address register	00000	01111	R, W	X
RILC	Reload inner loop count register	00000	01110	R, W	R, W
SSR	Saturation status register	00000	10101	R, W	R, W
TSCH	Time-stamp counter (high 32 bits) register	00000	01011	R	R
TSCL	Time-stamp counter (low 32 bits) register	00000	01010	R	R
TSR	Task state register	00000	11010	R, W*	R,W*

(1) R = Readable by the **MVC** instruction; W = Writeable by the **MVC** instruction; W* = Partially writeable by the **MVC** instruction; X = Access causes exception

MVD *Move From Register to Register, Delayed*

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

unit = .M1 or .M2

Compatibility C64x and C64x+ CPU

Opcode

31–29	28	27–23	22–18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	1	1	0	1	0	x	0	0	0	0	1	1	1	1	0	0	*s*	*p*
3	1	5	5						1											1	1

Opcode map field used...	For operand type...	Unit
src2 *dst*	xint int	.M1, .M2

Description Moves data from the *src2* register to the *dst* register over 4 cycles. This is done using the multiplier path.

```
MVD    .M2x  A0, B0   ;
NOP                   ;
NOP                   ;
NOP                   ; B0 = A0
```

Execution

if (cond) *src2* → *dst*

else nop

Pipeline

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

Instruction Type Four-cycle

Delay Slots 3

Example `MVD .M2X A5,B8`

	Before instruction		4 cycles after instruction
A5	6A32 1193h	A5	6A32 1193h
B8	xxxx xxxxh	B8	6A32 1193h

MVK *Move Signed Constant Into Register and Sign Extend*

Syntax

MVK (.unit) *cst, dst*

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

Compatibility C62x, C64x, and C64x+ CPU

Compact Instruction Format

Unit	Opcode Format	Figure
.L	Lx5	Figure D-8
.S	Smvk8	Figure F-23
.L, .S, .D	LSDx1c	Figure G-3
	LSDx1	Figure G-4

Opcode .S unit

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 *dst*	scst16 sint	.S1, .S2

Opcode .L unit (C64x and C64x+ CPU)

31 29	28	27 23	22 18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*cst5*	0	0	1	0	1	x	0	0	1	1	0	1	0	1	1	0	*s*	*p*
3	1	5	5						1											1	1

Opcode map field used...	For operand type...	Unit
cst5 *dst*	scst5 sint	.L1, .L2

Opcode .D unit (C64x and C64x+ CPU)

31 29	28	27 23	22	21	20	19	18	17 13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	0	0	0	0	0	*cst5*	0	0	0	0	0	0	1	0	0	0	0	*s*	*p*
3	1	5						5												1	1

Opcode map field used...	For operand type...	Unit
cst5 *dst*	scst5 sint	.D1, .D2

Description The constant *cst* is sign extended and placed in *dst*. The .S unit form allows for a 16-bit signed constant.

Since many nonaddress constants fall into a 5-bit sign constant range, this allows the flexibility to schedule the **MVK** instruction on the .L or .D units. In the .D unit form, the constant is in the position normally used by *src1*, as for address math.

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	.L, .S, or .D

Instruction Type Single cycle

Delay Slots 0

See Also MVKH, MVKL, MVKLH

Examples

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, and C64x+ CPU

Opcode

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	1

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

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.

NOTE: Use the **MVK** instruction (see MVK) 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
```

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

See Also MVK, MVKL

Examples

Example 1

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

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

Example 2

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

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

MVKL *Move Signed Constant Into Register and Sign Extend*

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

unit = .S1 or .S2

Compatibility C62x, C64x, and C64x+ 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 *dst*	scst16 sint	.S1, .S2

Description The 16-bit constant, *cst16*, is sign extended and placed in *dst*.

The **MVKL** instruction is equivalent to the **MVK** instruction (see MVK), 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 (see MVKH) 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

Examples

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*

or

NEG (.L1 or .L2) *src2_h:src2_l, dst_h:dst_l*

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

Compatibility C62x, C64x, and C64x+ CPU

Opcode .S unit

31 29	28	27 23	22 18	17	16	15	14	13	12	11	10	9	8	7	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
src2 *dst*	xsint sint	.S1, .S2

Opcode .L unit

31 29	28	27 23	22 18	17	16	15	14	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 0110
src2 *dst*	slong slong	.L1, .L2	010 0100

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

Execution

```
if (cond)     0 -s src2 → dst
else nop
```

Instruction Type Single-cycle

Delay Slots 0

See Also SUB

NOP — *No Operation*

Syntax

NOP [*count*]

unit = none

Compatibility C62x, C64x, and C64x+ CPU

Compact Instruction Format

Unit	Opcode Format	Figure
none	Unop	Figure H-9

Opcode

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

A multicycle **NOP** instruction cannot be paired with any other multicycle **NOP** instruction in the same execute packet. Instructions that generate a multicycle **NOP** are: **ADDKPC**, **BNOP**, **CALLP**, and **IDLE**.

Execution No operation for *count* cycles

Instruction Type **NOP**

Delay Slots 0

Examples

Example 1

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

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

Example 2

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

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

NORM *Normalize Integer*

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

or

NORM (.unit) *src2_h:src2_l, dst*

unit = .L1 or .L2

Compatibility C62x, C64x, and C64x+ CPU

Opcode

31 29	28	27 23	22 18	17	16	15	14	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:

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:

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:

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:

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

Examples

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

Example 3

```
NORM .L1 A1:A0,A3
```

	Before instruction		1 cycle after instruction	
A0	0000 0007h	A0	0000 0007h	
A1	0000 0000h	A1	0000 0000h	
A3	xxxx xxxxh	A3	0000 0024h	36

NOT *Bitwise NOT*

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

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

Compatibility C62x, C64x, and C64x+ CPU

Opcode .L unit

31 29	28	27 23	22 18	17	16	15	14	13	12	11	10	9	8	7	6	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	16	15	14	13	12	11	10	9	8	7	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						1											1	1

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

Opcode .D unit

31 29	28	27 23	22 18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	1	1	1	1	1	x	1	0	1	1	1	1	1	1	0	0	*s*	*p*
3	1	5	5						1											1	1

Opcode map field used...	For operand type...	Unit
src2 *dst*	xuint uint	.D1, .D2

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

Execution

```
if (cond)        -1 XOR src2 → dst
else nop
```

Instruction Type Single-cycle

Delay Slots 0

See Also XOR

OR *Bitwise OR*

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

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

Compatibility C62x, C64x, and C64x+ CPU

Compact Instruction Format

Unit	Opcode Format	Figure
.L	L2c	Figure D-7

Opcode .D unit

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

Opcode map field used...	For operand type...	Unit	Opfield
src1 *src2* *dst*	uint xuint uint	.D1, .D2	0010
src1 *src2* *dst*	scst5 xuint uint	.D1, .D2	0011

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, .S, or .D

Instruction Type Single-cycle

Delay Slots 0

See Also AND, ANDN, XOR

Examples **Example 1**

`OR .S1 A3,A4,A5`

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

Example 2

```
OR .D2 -12,B2,B8
```

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

PACK2 *Pack Two 16 LSBs Into Upper and Lower Register Halves*

Syntax **PACK2** (.unit) *src1, src2, dst*

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

Compatibility C64x and C64x+ CPU

Opcode .L unit

31 29	28	27 23	22 18	17 13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	0	0	0	0	0	0	0	1	1	0	*s*	*p*
3	1	5	5	5	1											1	1

Opcode map field used...	For operand type...	Unit
src1	i2	.L1, .L2
src2	xi2	
dst	i2	

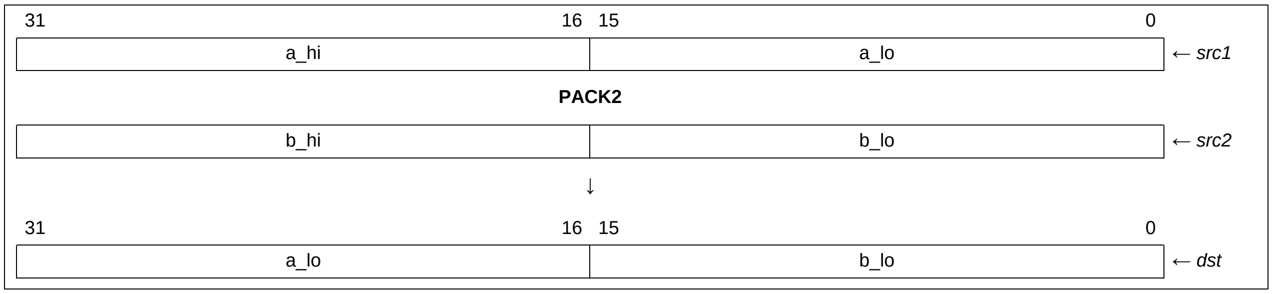
Opcode .S unit

31 29	28	27 23	22 18	17 13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	1	1	1	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	i2	.S1, .S2
src2	xi2	
dst	i2	

Description Moves the lower halfwords from *src1* and *src2* and packs them both into *dst*. The lower halfword of *src1* is placed in the upper halfword of *dst*. The lower halfword of *src2* is placed in the lower halfword of *dst*.

This instruction is useful for manipulating and preparing pairs of 16-bit values to be used by the packed arithmetic operations, such as **ADD2** (see ADD2).



Execution

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

Pipeline

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

Instruction Type Single-cycle

Delay Slots 0

See Also PACKH2, PACKHL2, PACKLH2, SPACK2

Examples

Example 1

```
PACK2 .L1 A2,A8,A9
```

	Before instruction		1 cycle after instruction
A2	3789 F23Ah	A2	3789 F23Ah
A8	04B8 4975h	A8	04B8 4975h
A9	xxxx xxxxh	A9	F23A 4975h

Example 2

```
PACK2 .S2 B2,B8,B12
```

	Before instruction		1 cycle after instruction
B2	0124 2451h	B2	0124 2451h
B8	01A6 A051h	B8	01A6 A051h
B12	xxxx xxxxh	B12	2451 A051h

PACKH2 *Pack Two 16 MSBs Into Upper and Lower Register Halves*

Syntax **PACKH2** (.unit) *src1, src2, dst*

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

Compatibility C64x and C64x+ CPU

Opcode .L unit

31 29	28	27 23	22 18	17 13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	0	0	1	1	1	1	0	1	1	0	*s*	*p*
3	1	5	5	5	1											1	1

Opcode map field used...	For operand type...	Unit
src1	i2	.L1, .L2
src2	xi2	
dst	i2	

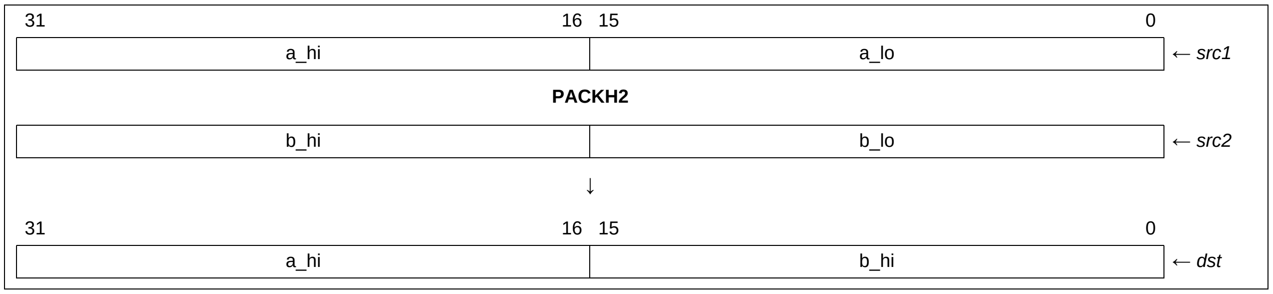
Opcode .S unit

31 29	28	27 23	22 18	17 13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	0	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	i2	.S1, .S2
src2	xi2	
dst	i2	

Description Moves the upper halfwords from *src1* and *src2* and packs them both into *dst*. The upper halfword of *src1* is placed in the upper half-word of *dst*. The upper halfword of *src2* is placed in the lower halfword of *dst*.

This instruction is useful for manipulating and preparing pairs of 16-bit values to be used by the packed arithmetic operations, such as **ADD2** (see ADD2).



Execution

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

Pipeline

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

Instruction Type Single-cycle

Delay Slots 0

See Also PACK2, PACKHL2, PACKLH2, SPACK2

Examples

Example 1

```
PACKH2 .L1 A2,A8,A9
```

	Before instruction		1 cycle after instruction
A2	3789 F23Ah	A2	3789 F23Ah
A8	04B8 4975h	A8	04B8 4975h
A9	xxxx xxxxh	A9	3789 04B8h

Example 2

```
PACKH2 .S2 B2,B8,B12
```

	Before instruction		1 cycle after instruction
B2	0124 2451h	B2	0124 2451h
B8	01A6 A051h	B8	01A6 A051h
B12	xxxx xxxxh	B12	0124 01A6h

PACKH4 *Pack Four High Bytes Into Four 8-Bit Halfwords*

Syntax

PACKH4 (.unit) *src1, src2, dst*

unit = .L1 or .L2

Compatibility

C64x and C64x+ CPU

Opcode

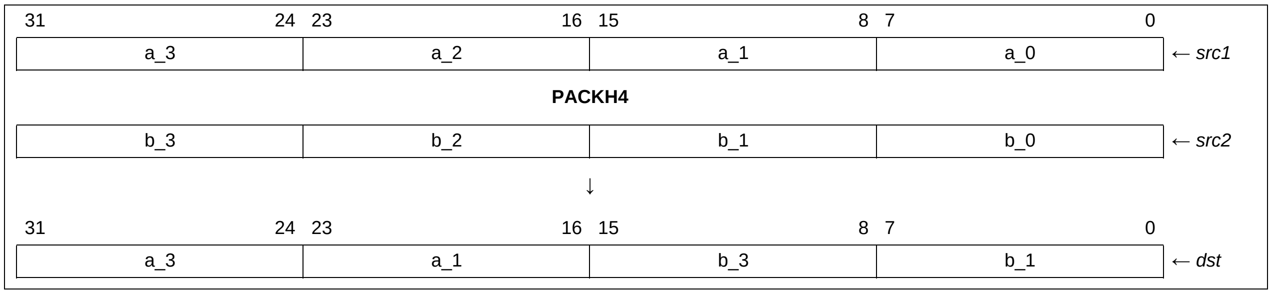
31–29	28	27–23	22–18	17–13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	1	1	0	1	0	0	1	1	1	0	*s*	*p*
3	1	5	5	5	1											1	1

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

Description

Moves the high bytes of the two halfwords in *src1* and *src2*, and packs them into *dst*. The bytes from *src1* are packed into the most-significant bytes of *dst*, and the bytes from *src2* are packed into the least-significant bytes of *dst*.

- The high byte of the upper halfword of *src1* is moved to the upper byte of the upper halfword of *dst*. The high byte of the lower halfword of *src1* is moved to the lower byte of the upper halfword of *dst*.
- The high byte of the upper halfword of *src2* is moved to the upper byte of the lower halfword of *dst*. The high byte of the lower halfword of *src2* is moved to the lower byte of the lower halfword of *dst*.



Execution

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

Pipeline

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

Instruction Type Single-cycle

Delay Slots 0

See Also PACKL4, SPACKU4

Examples

Example 1

```
PACKH4 .L1 A2,A8,A9
```

	Before instruction		1 cycle after instruction
A2	37 89 F2 3Ah	A2	37 89 F2 3Ah
A8	04 B8 49 75h	A8	04 B8 49 75h
A9	xxxx xxxxh	A9	37 F2 04 49h

Example 2

```
PACKH4 .L2 B2,B8,B12
```

	Before instruction		1 cycle after instruction
B2	01 24 24 51h	B2	01 24 24 51h
B8	01 A6 A0 51h	B8	01 A6 A0 51h
B12	xxxx xxxxh	B12	01 24 01 A0h

PACKHL2 *Pack 16 MSB Into Upper and 16 LSB Into Lower Register Halves*

Syntax

PACKHL2 (.unit) *src1, src2, dst*

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

Compatibility C64x and C64x+ CPU

Opcode .L unit

31 29	28	27 23	22 18	17 13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	0	0	1	1	1	0	0	1	1	0	*s*	*p*
3	1	5	5	5	1											1	1

Opcode map field used...	For operand type...	Unit
src1	i2	.L1, .L2
src2	xi2	
dst	i2	

Opcode .S unit

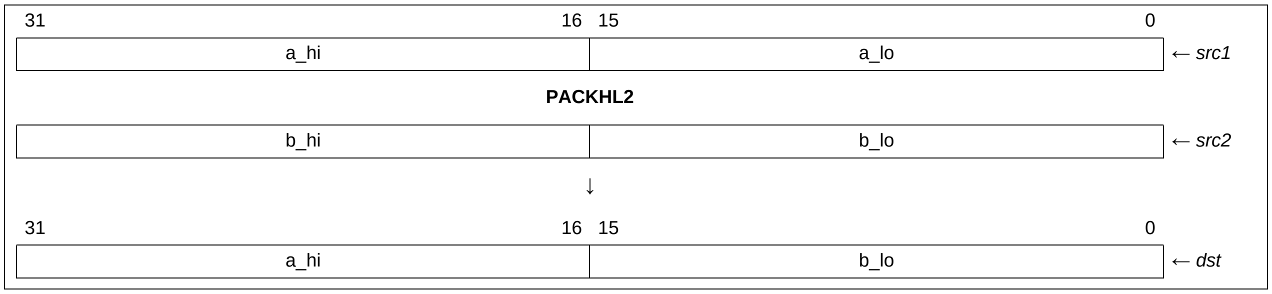
31 29	28	27 23	22 18	17 13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	0	0	1	0	0	0	1	0	0	0	*s*	*p*
3	1	5	5	5	1											1	1

Opcode map field used...	For operand type...	Unit
src1	i2	.S1, .S2
src2	xi2	
dst	i2	

Description

Moves the upper halfword from *src1* and the lower halfword from *src2* and packs them both into *dst*. The upper halfword of *src1* is placed in the upper halfword of *dst*. The lower halfword of *src2* is placed in the lower halfword of *dst*.

This instruction is useful for manipulating and preparing pairs of 16-bit values to be used by the packed arithmetic operations, such as **ADD2** (see ADD2).



Execution

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

Pipeline

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

Instruction Type Single-cycle

Delay Slots 0

See Also PACK2, PACKH2, PACKLH2, SPACK2

Examples

Example 1

```
PACKHL2 .L1 A2,A8,A9
```

	Before instruction		1 cycle after instruction
A2	3789 F23Ah	A2	3789 F23Ah
A8	04B8 4975h	A8	04B8 4975h
A9	xxxx xxxxh	A9	3789 4975h

Example 2

```
PACKHL2 .S2 B2,B8,B12
```

	Before instruction		1 cycle after instruction
B2	0124 2451h	B2	0124 2451h
B8	01A6 A051h	B8	01A6 A051h
B12	xxxx xxxxh	B12	0124 A051h

PACKLH2 *Pack 16 LSB Into Upper and 16 MSB Into Lower Register Halves*

Syntax **PACKLH2** (.unit) *src1, src2, dst*

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

Compatibility C64x and C64x+ CPU

Opcode .L unit

31 29	28	27 23	22 18	17 13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	0	0	1	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	i2	.L1, .L2
src2	xi2	
dst	i2	

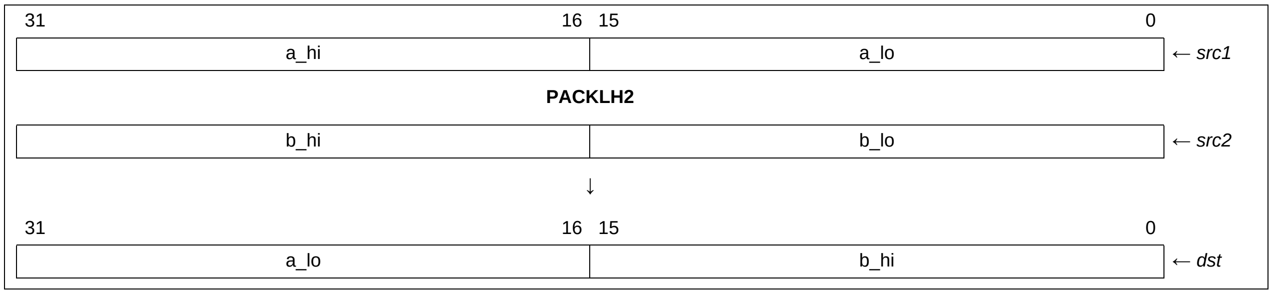
Opcode .S unit

31 29	28	27 23	22 18	17 13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	0	1	0	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	i2	.S1, .S2
src2	xi2	
dst	i2	

Description Moves the lower halfword from *src1,* and the upper halfword from *src2,* and packs them both into *dst*. The lower halfword of *src1* is placed in the upper halfword of *dst*. The upper halfword of *src2* is placed in the lower halfword of *dst*.

This instruction is useful for manipulating and preparing pairs of 16-bit values to be used by the packed arithmetic operations, such as **ADD2** (see ADD2).



Execution

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

Pipeline

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

Instruction Type Single-cycle

Delay Slots 0

See Also PACK2, PACKH2, PACKHL2, SPACK2

Examples

Example 1

```
PACKLH2 .L1 A2,A8,A9
```

	Before instruction		1 cycle after instruction
A2	3789 F23Ah	A2	3789 F23Ah
A8	04B8 4975h	A8	04B8 4975h
A9	xxxx xxxxh	A9	F23A 04B8h

Example 2

```
PACKLH2 .S2 B2,B8,B12
```

	Before instruction		1 cycle after instruction
B2	0124 2451h	B2	0124 2451h
B8	01A6 A051h	B8	01A6 A051h
B12	xxxx xxxxh	B12	2451 01A6h

PACKL4 *Pack Four Low Bytes Into Four 8-Bit Halfwords*

Syntax **PACKL4** (.unit) *src1, src2, dst*

unit = .L1 or .L2

Compatibility C64x and C64x+ CPU

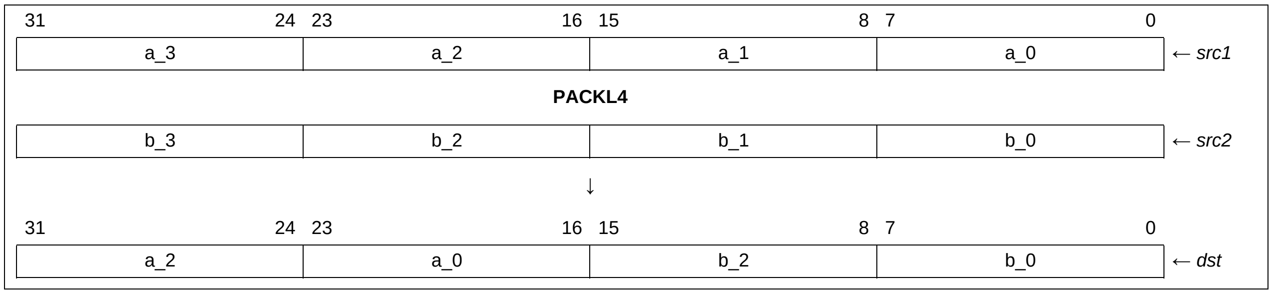
Opcode

31–29	28	27–23	22–18	17–13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	1	1	0	1	0	0	0	1	1	0	*s*	*p*
3	1	5	5	5	1											1	1

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

Description Moves the low bytes of the two halfwords in *src1* and *src2,* and packs them into *dst*. The bytes from *src1* are packed into the most-significant bytes of *dst*, and the bytes from *src2* are packed into the least-significant bytes of *dst*.

- The low byte of the upper halfword of *src1* is moved to the upper byte of the upper halfword of *dst*. The low byte of the lower halfword of *src1* is moved to the lower byte of the upper halfword of *dst*.
- The low byte of the upper halfword of *src2* is moved to the upper byte of the lower halfword of *dst*. The low byte of the lower halfword of *src2* is moved to the lower byte of the lower halfword of *dst*.



Execution

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

Pipeline

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

Instruction Type Single-cycle

Delay Slots 0

See Also PACKH4, SPACKU4

Examples

Example 1

```
PACKL4 .L1 A2,A8,A9
```

	Before instruction		1 cycle after instruction
A2	37 89 F2 3Ah	A2	37 89 F2 3Ah
A8	04 B8 49 75h	A8	04 B8 49 75h
A9	xxxx xxxxh	A9	89 3A B8 75h

Example 2

```
PACKL4 .L2 B2,B8,B12
```

	Before instruction		1 cycle after instruction
B2	01 24 24 51h	B2	01 24 24 51h
B8	01 A6 A0 51h	B8	01 A6 A0 51h
B12	xxxx xxxxh	B12	24 51 A6 51h

RINT *Restore Previous Enable State*

Syntax **RINT**

unit = none

Compatibility C64x+ CPU

Opcode

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	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	*p*
																															1

Description Copies the contents of the SGIE bit in TSR into the GIE bit in TSR and CSR, and clears the SGIE bit in TSR. The value of the SGIE bit in TSR is used for the current cycle as the GIE indication; if restoring the GIE bit to 1, interrupts are enabled and can be taken after the E1 phase containing the **RINT** instruction.

The CPU may service a maskable interrupt in the cycle immediately following the **RINT** instruction. See section 5.2 for details.

The **RINT** instruction cannot be placed in parallel with: **MVC** *reg*, **TSR**; **MVC** *reg*, **CSR**; **B IRP**; **B NRP**; **NOP** *n*; **DINT**; **SPKERNEL**; **SPKERNELR**; **SPLOOP**; **SPLOOPD**; **SPLOOPW**; **SPMASK**; or **SPMASKR**.

This instruction executes unconditionally and cannot be predicated.

NOTE: The use of the **DINT** and **RINT** instructions in a nested manner, like the following code:

```
DINT
DINT
RINT
RINT
```

leaves interrupts disabled. The first **DINT** leaves TSR.GIE cleared to 0, so the second **DINT** leaves TSR,.SGIE cleared to 0. The **RINT** instructions, therefore, copy zero to TSR.GIE (leaving interrupts disabled).

Execution Enable interrupts in current cycle

SGIE bit in TSR → GIE bit in TSR
SGIE bit in TSR → GIE bit in CSR
0 → SGIE bit in TSR

Instruction Type Single-cycle

Delay Slots 0

See Also DINT

ROTL *Rotate Left*

Syntax

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

unit = .M1 or .M2

Compatibility

C64x and C64x+ CPU

Opcode

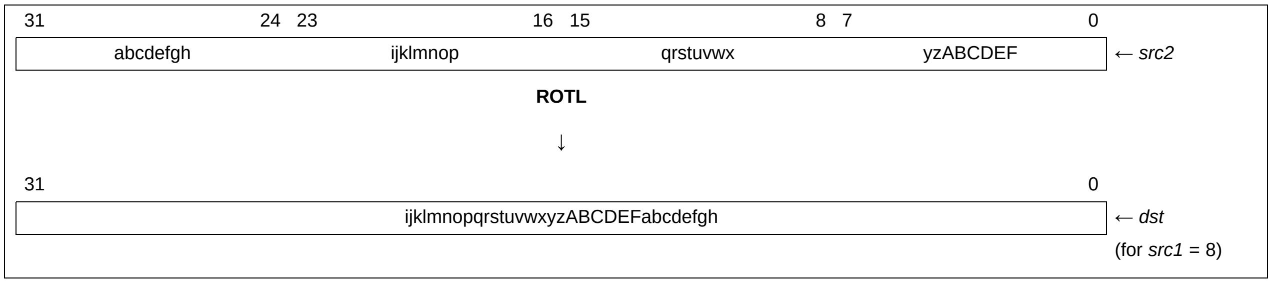
31–29	28	27–23	22–18	17–13	12	11	10–6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	0	*op*	1	1	0	0	*s*	*p*
3	1	5	5	5	1		5					1	1

Opcode map field used...	For operand type...	Unit	Opfield
src1 *src2* *dst*	uint xuint uint	.M1, .M2	11101
src1 *src2* *dst*	ucst5 xuint uint	.M1, .M2	11110

Description

Rotates the 32-bit value of *src2* to the left, and places the result in *dst*. The number of bits to rotate is given in the 5 least-significant bits of *src1*. Bits 5 through 31 of *src1* are ignored and may be non-zero.

In the following figure, *src1* is equal to 8.



NOTE: The **ROTL** instruction is useful in cryptographic applications.

Execution

```
if (cond)        (src2 << src1) | (src2 >> (32 - src1)) → dst
else nop
```

Pipeline

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

Instruction Type Two-cycle

Delay Slots 1

See Also SHL, SHLMB, SHRMB, SHR, SHRU

Examples

Example 1

```
ROTL .M2 B2,B4,B5
```

	Before instruction		**2 cycles after instruction**
B2	A6E2 C179h	B2	A6E2 C179h
B4	1458 3B69h	B4	1458 3B69h
B5	xxxx xxxxh	B5	C582 F34Dh

Example 2

```
ROTL .M1 A4,10h,A5
```

	Before instruction		**2 cycles after instruction**
A4	187A 65FCh	A4	187A 65FCh
A5	xxxx xxxxh	A5	65FC 187Ah

RPACK2 *Shift With Saturation and Pack Two 16 MSBs Into Upper and Lower Register Halves*

Syntax

RPACK2 (.unit) *src1, src2, dst*

unit = .S1 or .S2

Compatibility

C64x+ CPU

Opcode

31	30	29	28	27 … 23	22 … 18	17 … 13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	*dst*	*src2*	*src1*	x	1	1	1	0	1	1	1	1	0	0	*s*	*p*
				5	5	5	1											1	1

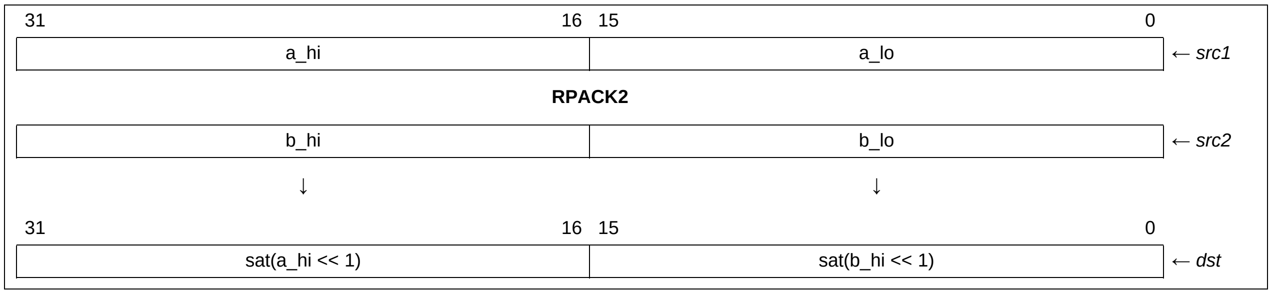
Opcode map field used...	For operand type...	Unit
src1	sint	.S1, .S2
src2	xsint	
dst	s2	

Description

src1 and *src2* are shifted left by 1 with saturation. The 16 most-significant bits of the shifted *src1* value are placed in the 16 most-significant bits of *dst*. The 16 most-significant bits of the shifted *src2* value are placed in the 16 least-significant bits of *dst*.

If either value saturates, the S1 or S2 bit in SSR and the SAT bit in CSR are written one cycle after the result is written to *dst*.

This instruction executes unconditionally and cannot be predicated.



Execution

msb16(sat(*src1* << 1)) → msb16(*dst*)
msb16(sat(*src2* << 1)) → lsb16(*dst*)

Instruction Type

Single-cycle

Delay Slots

0

See Also

PACK2, PACKH2, SPACK2

Examples

Example 1

```
RPACK2 .S1 A0,A1,A2
```

	Before instruction		1 cycle after instruction
A0	FEDC BA98h	A2	FDBA 2468h
A1	1234 5678h		
CSR	0001 0100h	CSR[1]	0001 0100h
SSR	0000 0000h	SSR[1]	0000 0000h

(1) CSR.SAT and SSR.S1 unchanged by operation

Example 2

```
RPACK2 .S2X B0,A1,B2
```

	Before instruction		1 cycle after instruction
B0	8765 4321h	B2	8000 2468h
A1	1234 5678h		
CSR	0001 0100h	CSR[1]	0001 0300h
SSR	0000 0000h	SSR[1]	0000 0008h

(1) CSR.SAT and SSR.S2 set to 1, 2 cycles after instruction

SADD *Add Two Signed Integers With Saturation*

Syntax

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

or

SADD (.L1 or .L2) *src1, src2_h:src2_l, dst_h:dst_l*

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

Compatibility C62x, C64x, and C64x+ CPU

Compact Instruction Format

Unit	Opcode Format	Figure
.L	L3	Figure D-4
.S	S3	Figure F-21

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	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

Opcode .S unit

31 29	28	27 23	22 18	17 13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	1	0	0	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*	sint xsint sint	.S1, .S2

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, .S

Instruction Type Single-cycle

Delay Slots 0

See Also ADD

Examples

Example 1

`SADD .L1 A1,A2,A3`

	Before instruction			1 cycle after instruction	
A1	5A2E 51A3h	1,512,984,995	A1	5A2E 51A3h	
A2	012A 3FA2h	19,546,018	A2	012A 3FA2h	
A3	xxxx xxxxh		A3	5B58 9145h	1,532,531,013
CSR	0001 0100h		CSR	0001 0100h	
SSR[(1)]	0000 0000h		SSR	0000 0000h	
				2 cycles after instruction	
			A1	5A2E 51A3h	
			A2	012A 3FA2h	
			A3	5B58 9145h	
			CSR	0001 0100h	Not saturated
			SSR	0000 0000h	

(1) Saturation status register (SSR) is only available on the C64x+ DSP.

Example 2

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

	Before instruction			1 cycle after instruction	
A1	4367 71F2h	1,130,852,850	A1	4367 71F2h	
A2	5A2E 51A3h	1,512,984,995	A2	5A2E 51A3h	
A3	xxxx xxxxh		A3	7FFF FFFFh	2,147,483,647
CSR	0001 0100h		CSR	0001 0100h	
SSR[1]	0000 0000h		SSR	0000 0000h	
				2 cycles after instruction	
			A1	4367 71F2h	
			A2	5A2E 51A3h	
			A3	7FFF FFFFh	
			CSR	0001 0300h	Saturated
			SSR	0000 0001h	

(1) Saturation status register (SSR) is only available on the C64x+ DSP.

Example 3

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

	Before instruction				1 cycle after instruction	
A5:A4	0000 0000h	7C83 39B1h		A5:A4	0000 0000h	7C83 39B1h
	2,088,974,769[1]					
A7:A6	xxxx xxxxh	xxxx xxxxh		A7:A6	0000 0000h	8DAD 7953h
					2,376,956,243[1]	
B2	112A 3FA2h	287,981,474		B2	112A 3FA2h	
CSR	0001 0100h			CSR	0001 0100h	
SSR[2]	0000 0000h			SSR	0000 0000h	
					2 cycles after instruction	
				A5:A4	0000 0000h	7C83 39B1h
				A7:A6	0000 0000h	8DAD 7953h
				B2	112A 3FA2h	
				CSR	0001 0100h	Not saturated
				SSR	0000 0000h	

(1) Signed 40-bit (long) integer
(2) Saturation status register (SSR) is only available on the C64x+ DSP.

SADD2 *Add Two Signed 16-Bit Integers on Upper and Lower Register Halves With Saturation*

Syntax **SADD2** (.unit) *src1, src2, dst*

unit = .S1 or .S2

Compatibility C64x and C64x+ CPU

Opcode

31–29	28	27–23	22–18	17–13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	1	1	0	0	0	0	1	1	0	0	*s*	*p*
3	1	5	5	5	1											1	1

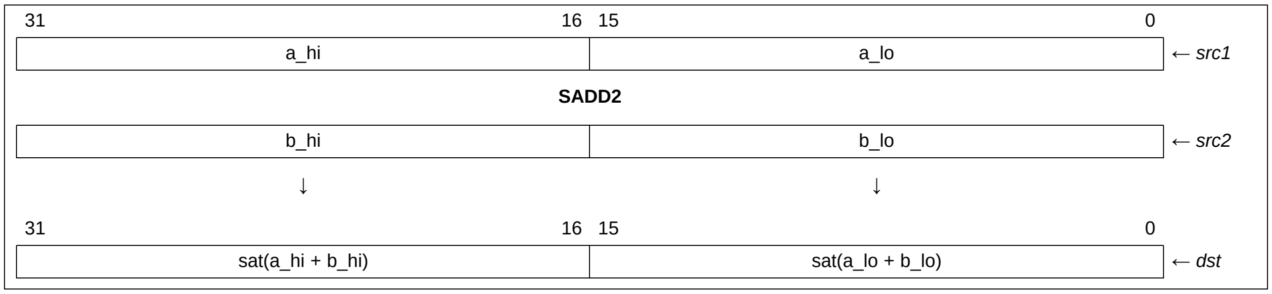
Opcode map field used...	For operand type...	Unit
src1	s2	.S1, .S2
src2	xs2	
dst	s2	

Description Performs 2s-complement addition between signed, packed 16-bit quantities in *src1* and *src2*. The results are placed in a signed, packed 16-bit format into *dst*.

For each pair of 16-bit quantities in *src1* and *src2*, the sum between the signed 16-bit value from *src1* and the signed 16-bit value from *src2* is calculated and saturated to produce a signed 16-bit result. The result is placed in the corresponding position in *dst*.

Saturation is performed on each 16-bit result independently. For each sum, the following tests are applied:

- If the sum is in the range -2^{15} to $2^{15} - 1$, inclusive, then no saturation is performed and the sum is left unchanged.
- If the sum is greater than $2^{15} - 1$, then the result is set to $2^{15} - 1$.
- If the sum is less than -2^{15}, then the result is set to -2^{15}.



NOTE: This operation is performed on each halfword separately. This instruction does not affect the SAT bit in CSR.

Execution

```
if (cond)    {
             sat(msb16(src1) + msb16(src2)) → msb16(dst);
             sat(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 ADD2, SADD, SADDUS2, SADDU4, SUB2

Examples

Example 1

```
SADD2 .S1 A2,A8,A9
```

	Before instruction			1 cycle after instruction	
A2	5789 F23Ah	22409 -3526	A2	5789 F23Ah	
A8	74B8 4975h	29880 18805	A8	74B8 4975h	
A9	xxxx xxxxh		A9	7FFF 3BAFh	32767 15279

Example 2

```
SADD2 .S2 B2,B8,B12
```

	Before instruction			1 cycle after instruction	
B2	0124 847Ch	292 -31260	B2	0124 847Ch	
B8	01A6 A051h	422 -24495	B8	01A6 A051h	
B12	xxxx xxxxh		B12	02CA 8000h	714 -32768

SADDSUB *Parallel SADD and SSUB Operations On Common Inputs*

Syntax **SADDSUB** (.unit) *src1, src2, dst_o:dst_e*

unit = .L1 or .L2

Compatibility C64x+ CPU

Opcode

31	30	29	28	27–24	23	22–18	17–13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	*dst*	0	*src2*	*src1*	x	0	0	0	1	1	1	0	1	1	0	*s*	*p*
				4		5	5	1											1	1

Opcode map field used...	For operand type...	Unit
src1	sint	.L1, .L2
src2	xsint	
dst	dint	

Description The following is performed in parallel:

1. *src2* is added with saturation to *src1*. The result is placed in *dst_o*.
2. *src2* is subtracted with saturation from *src1*. The result is placed in *dst_e*.

If either result saturates, the L1 or L2 bit in SSR and the SAT bit in CSR are written one cycle after the results are written to *dst_o:dst_e*.

This instruction executes unconditionally and cannot be predicated.

Execution

sat(*src1* + *src2*) → *dst_o*
sat(*src1* - *src2*) → *dst_e*

Instruction Type Single-cycle

Delay Slots 0

See Also ADDSUB, SADDSUB2

Examples

Example 1

```
SADDSUB .L1 A0,A1,A3:A2
```

	Before instruction		1 cycle after instruction
A0	0700 C005h	A2	0700 C006h
A1	FFFF FFFFh	A3	0700 C004h
CSR	0001 0100h	CSR[(1)]	0001 0100h
SSR	0000 0000h	SSR[(1)]	0000 0000h

(1) CSR.SAT and SSR.L1 unchanged by operation

Example 2

```
SADDSUB .L2X B0,A1,B3:B2
```

	Before instruction		1 cycle after instruction
B0	7FFF FFFFh	B2	7FFF FFFEh
A1	0000 0001h	B3	7FFF FFFFh
CSR	0001 0100h	CSR[1]	0001 0300h
SSR	0000 0000h	SSR[1]	0000 0002h

(1) CSR.SAT and SSR.L2 set to 1, 2 cycles after instruction

Example 3

```
SADDSUB .L1X A0,B1,A3:A2
```

	Before instruction		1 cycle after instruction
A0	8000 0000h	A2	8000 0000h
B1	0000 0001h	A3	8000 0001h
CSR	0001 0100h	CSR[1]	0001 0300h
SSR	0000 0000h	SSR[1]	0000 0001h

(1) CSR.SAT and SSR.L1 set to 1, 2 cycles after instruction

SADDSUB2 *Parallel SADD2 and SSUB2 Operations On Common Inputs*

Syntax

SADDSUB2 (.unit) *src1, src2, dst_o:dst_e*

unit = .L1 or .L2

Compatibility

C64x+ CPU

Opcode

31	30	29	28	27 24	23	22 18	17 13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	*dst*	0	*src2*	*src1*	x	0	0	0	1	1	1	1	1	1	0	*s*	*p*
				4		5	5	1											1	1

Opcode map field used...	For operand type...	Unit
src1	sint	.L1, .L2
src2	xsint	
dst	dint	

Description

A **SADD2** and a **SSUB2** operation are done in parallel.

For the **SADD2** operation, the upper and lower halves of the *src2* operand are added with saturation to the upper and lower halves of the *src1* 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_o*.

For the **SSUB2** operation, the upper and lower halves of the *src2* operand are subtracted with saturation from the upper and lower halves of the *src1* 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_e*.

This instruction executes unconditionally and cannot be predicated.

NOTE: These operations are performed separately on each halfword. This instruction does not affect the SAT bit in CSR or the L1 or L2 bits in SSR.

Execution

sat(lsb16(*src1*) + lsb16(*src2*)) → lsb16(*dst_o*)
sat(msb16(*src1*) + msb16(*src2*)) → msb16(*dst_o*)
sat(lsb16(*src1*) - lsb16(*src2*)) → lsb16(*dst_e*)
sat(msb16(*src1*) - msb16(*src2*)) → msb16(*dst_e*)

Instruction Type Single-cycle

Delay Slots 0

See Also ADDSUB2, SADDSUB

Examples

Example 1

```
SADDSUB2 .L1 A0,A1,A3:A2
```

	Before instruction		1 cycle after instruction
A0	0700 C005h	A2	0701 C004h
A1	FFFF 0001h	A3	06FF C006h
CSR	0001 0100h	CSR[1]	0001 0100h
SSR	0000 0000h	SSR[1]	0000 0000h

(1) CSR.SAT and SSR.L1 unchanged by operation

Example 2

```
SADDSUB2 .L2X B0,A1,B3:B2
```

	Before instruction		1 cycle after instruction
B0	7FFF 8000h	B2	7FFF 8001h
A1	FFFF FFFFh	B3	7FFE 8000h
CSR	0001 0100h	CSR[1]	0001 0100h
SSR	0000 0000h	SSR[1]	0000 0000h

(1) CSR.SAT and SSR.L2 unchanged by operation

SADDSU2 *Add Two Signed and Unsigned 16-Bit Integers on Register Halves With Saturation*

Syntax **SADDSU2** (.unit) *src2, src1, dst*

unit = .S1 or .S2

Compatibility C64x and C64x+ CPU

Opcode

31 29	28	27 23	22 18	17 13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	1	1	0	0	0	1	1	1	0	0	*s*	*p*
3	1	5	5	5	1											1	1

Opcode map field used...	For operand type...	Unit
src1 *src2* *dst*	u2 xs2 u2	.S1, .S2

Description The **SADDSU2** pseudo-operation performs 2s-complement addition between unsigned and signed packed 16-bit quantities. The values in *src1* are treated as unsigned packed 16-bit quantities, and the values in *src2* are treated as signed packed 16-bit quantities. The results are placed in an unsigned packed 16-bit format into *dst*. The assembler uses the **SADDUS2** (.unit) *src1, src2, dst* instruction to perform this operation (see SADDUS2).

For each pair of 16-bit quantities in *src1* and *src2*, the sum between the unsigned 16-bit value from *src1* and the signed 16-bit value from *src2* is calculated and saturated to produce a signed 16-bit result. The result is placed in the corresponding position in *dst*.

Saturation is performed on each 16-bit result independently. For each sum, the following tests are applied:

- If the sum is in the range 0 to 2^{16} - 1, inclusive, then no saturation is performed and the sum is left unchanged.
- If the sum is greater than 2^{16} - 1, then the result is set to 2^{16} - 1.
- If the sum is less than 0, then the result is cleared to 0.

Execution

```
if (cond)    {
             sat(smsb16(src2) + umsb16(src1)) → umsb16(dst);
             sat(slsb16(src2) + ulsb16(src1)) → ulsb16(dst)
             }
else nop
```

Pipeline

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

Instruction Type	Single-cycle
Delay Slots	0
See Also	SADD, SADD2, SADDUS2, SADDU4

SADDUS2 *Add Two Unsigned and Signed 16-Bit Integers on Register Halves With Saturation*

Syntax

SADDUS2 (.unit) *src1, src2, dst*

unit = .S1 or .S2

Compatibility

C64x and C64x+ CPU

Opcode

31 29	28	27 23	22 18	17 13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	1	1	0	0	0	1	1	1	0	0	*s*	*p*
3	1	5	5	5	1											1	1

Opcode map field used...	For operand type...	Unit
src1	u2	.S1, .S2
src2	xs2	
dst	u2	

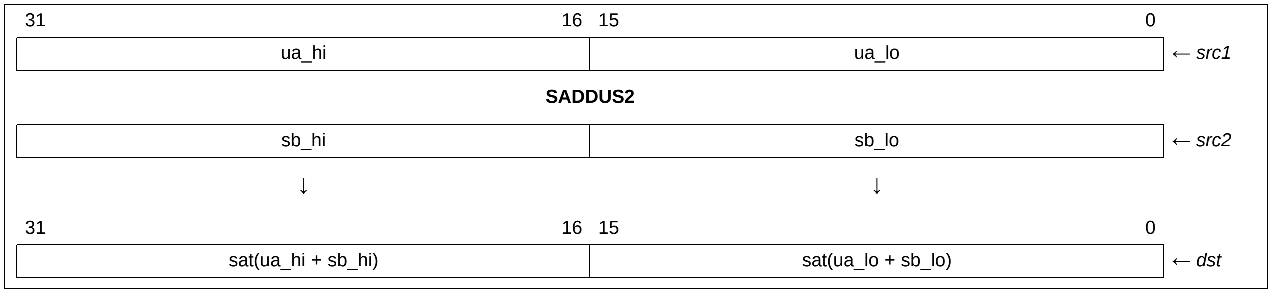
Description

Performs 2s-complement addition between unsigned and signed, packed 16-bit quantities. The values in *src1* are treated as unsigned, packed 16-bit quantities; and the values in *src2* are treated as signed, packed 16-bit quantities. The results are placed in an unsigned, packed 16-bit format into *dst*.

For each pair of 16-bit quantities in *src1* and *src2*, the sum between the unsigned 16-bit value from *src1* and the signed 16-bit value from *src2* is calculated and saturated to produce a signed 16-bit result. The result is placed in the corresponding position in *dst*.

Saturation is performed on each 16-bit result independently. For each sum, the following tests are applied:

- If the sum is in the range 0 to 2^{16} - 1, inclusive, then no saturation is performed and the sum is left unchanged.
- If the sum is greater than 2^{16} - 1, then the result is set to 2^{16} - 1.
- If the sum is less than 0, then the result is cleared to 0.



NOTE: This operation is performed on each halfword separately. This instruction does not affect the SAT bit in CSR.

Execution

```
if (cond)      {
               sat(umsb16(src1) + smsb16(src2)) → umsb16(dst);
               sat(ulsb16(src1) + slsb16(src2)) → ulsb16(dst)
               }
else nop
```

Pipeline

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

Instruction Type Single-cycle

Delay Slots 0

See Also ADD2, SADD, SADD2, SADDU4

Examples

Example 1

```
SADDUS2 .S1 A2, A8, A9
```

	Before instruction			1 cycle after instruction	
A2	5789 F23Ah	22409 62010 unsigned	A2	5789 F23Ah	
A8	74B8 4975h	29880 18805 signed	A8	74B8 4975h	
A9	xxxx xxxxh		A9	CC41 FFFF	52289 65535 unsigned

Example 2

```
SADDUS2 .S2 B2, B8, B12
```

	Before instruction			1 cycle after instruction	
B2	147C 0124h	5244 292 unsigned	B2	147C 0124h	
B8	A051 01A6h	-24495 422 signed	B8	A051 01A6h	
B12	xxxx xxxxh		B12	0000 02CAh	0 714 unsigned

SADDU4 *Add With Saturation, Four Unsigned 8-Bit Pairs for Four 8-Bit Results*

Syntax **SADDU4** (.unit) *src1, src2, dst*

unit = .S1 or .S2

Compatibility C64x and C64x+ CPU

Opcode

31 29	28	27 23	22 18	17 13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	1	1	0	0	1	1	1	1	0	0	*s*	*p*
3	1	5	5	5	1											1	1

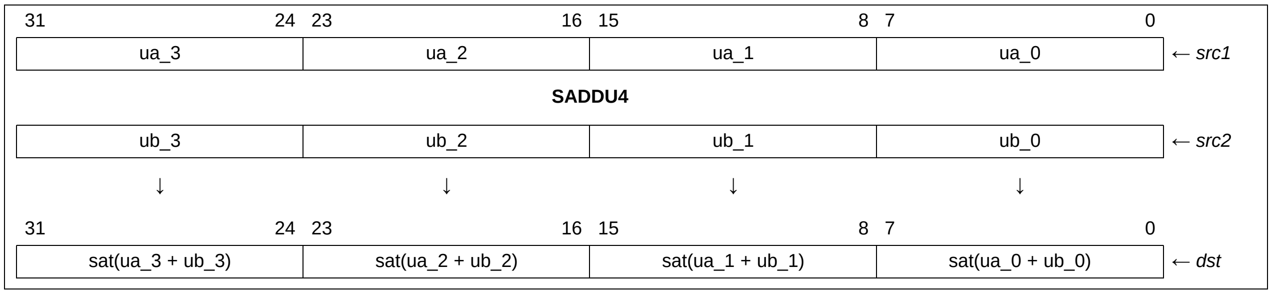
Opcode map field used...	For operand type...	Unit
src1	u4	.S1, .S2
src2	xu4	
dst	u4	

Description Performs 2s-complement addition between unsigned, packed 8-bit quantities. The values in *src1* and *src2* are treated as unsigned, packed 8-bit quantities and the results are written into *dst* in an unsigned, packed 8-bit format.

For each pair of 8-bit quantities in *src1* and *src2*, the sum between the unsigned 8-bit value from *src1* and the unsigned 8-bit value from *src2* is calculated and saturated to produce an unsigned 8-bit result. The result is placed in the corresponding position in *dst*.

Saturation is performed on each 8-bit result independently. For each sum, the following tests are applied:

- If the sum is in the range 0 to 2^8 - 1, inclusive, then no saturation is performed and the sum is left unchanged.
- If the sum is greater than 2^8 - 1, then the result is set to 2^8 - 1.



NOTE: This operation is performed on each 8-bit quantity separately. This instruction does not affect the SAT bit in CSR.

Execution

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

Pipeline

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

Instruction Type Single-cycle

Delay Slots 0

See Also ADD4, SADD, SADD2, SADDUS2, SUB4

Examples

Example 1

```
SADDU4 .S1 A2, A8, A9
```

	Before instruction			1 cycle after instruction	
A2	57 89 F2 3Ah	87 137 242 58 unsigned	A2	57 89 F2 3Ah	
A8	74 B8 49 75h	116 184 73 117 unsigned	A8	74 B8 49 75h	
A9	xxxx xxxxh		A9	CB FF FF AFh	203 255 255 175 unsigned

Example 2

```
SADDU4 .S2 B2, B8, B12
```

	Before instruction			1 cycle after instruction	
B2	14 7C 01 24h	20 124 1 36 unsigned	B2	14 7C 01 24h	
B8	A0 51 01 A6h	160 81 1 166 unsigned	B8	A0 51 01 A6h	
B12	xxxx xxxxh		B12	B4 CD 02 CA	180 205 2 202 unsigned

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

Syntax

SAT (.unit) *src2_h:src2_l, dst*

unit = .L1 or .L2

Compatibility C62x, C64x, and C64x+ CPU

Opcode

31–29	28	27–23	22–18	17	16	15	14	13	12	11	10	9	8	7	6	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) $\rightarrow$ dst
             else if (src2 < -$2^{31}$), -$2^{31}$ $\rightarrow$ dst
             else $src2_{31..0}$ $\rightarrow$ dst
             }
else nop
```

Pipeline

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

Instruction Type Single-cycle

Delay Slots 0

Examples

Example 1

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

	Before instruction			1 cycle after instruction	
B1:B0	0000 001Fh	3413 539Ah	B1:B0	0000 001Fh	3413 539Ah
B5	xxxx xxxxh		B5	7FFF FFFFh	
CSR	0001 0100h		CSR	0001 0100h	
SSR[1]	0000 0000h		SSR	0000 0000h	

	2 cycles after instruction	
B1:B0	0000 001Fh	3413 539Ah
B5	7FFF FFFFh	
CSR	0001 0300h	Saturated
SSR	0000 0002h	

(1) Saturation status register (SSR) is only available on the C64x+ DSP.

Example 2

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

	Before instruction			1 cycle after instruction	
B1:B0	0000 0000h	A190 7321h	B1:B0	0000 0000h	A190 7321h
B5	xxxx xxxxh		B5	7FFF FFFFh	
CSR	0001 0100h		CSR	0001 0100h	
SSR[1]	0000 0000h		SSR	0000 0000h	

	2 cycles after instruction	
B1:B0	0000 0000h	A190 7321h
B5	7FFF FFFFh	
CSR	0001 0300h	Saturated
SSR	0000 0002h	

(1) Saturation status register (SSR) is only available on the C64x+ DSP.

Example 3

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

	Before instruction				1 cycle after instruction	
B1:B0	0000 00FFh	A190 7321h		B1:B0	0000 00FFh	A190 7321h
B5	xxxx xxxxh			B5	A190 7321h	
CSR	0001 0100h			CSR	0001 0100h	
SSR[(1)]	0000 0000h			SSR	0000 0000h	
					2 cycles after instruction	
				B1:B0	0000 00FFh	A190 7321h
				B5	A190 7321h	
				CSR	0001 0100h	Not saturated
				SSR	0000 0000h	

(1) Saturation status register (SSR) is only available on the C64x+ DSP.

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, and C64x+ CPU

Compact Instruction Format

Unit	Opcode Format	Figure
.S	Sc5	Figure F-26

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:

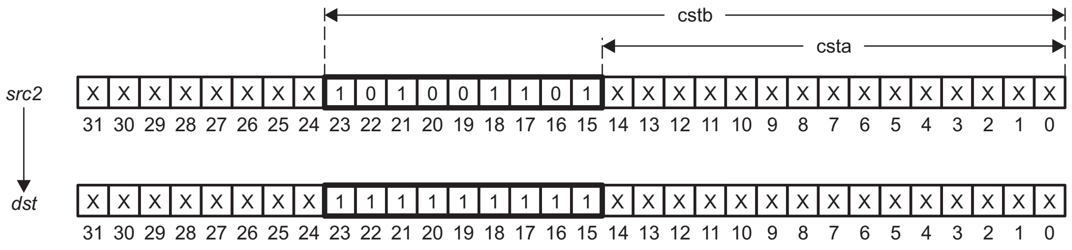
31 29	28	27 23	22 18	17 13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	1	1	1	0	1	1	1	0	0	0	*s*	*p*
3	1	5	5	5	1											1	1

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

Examples

Example 1

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

	Before instruction		1 cycle after instruction
A0	4B13 4A1Eh	A0	4B13 4A1Eh
A1	xxxx xxxxh	A1	4B3F FF9Eh

Example 2

```
SET .S2 B0,B1,B2
```

	Before instruction		1 cycle after instruction
B0	9ED3 1A31h	B0	9ED3 1A31h
B1	0000 C197h	B1	0000 C197h
B2	xxxx xxxxh	B2	9EFF FA31h

SHFL *Shuffle*

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

unit = .M1 or .M2

Compatibility C64x and C64x+ CPU

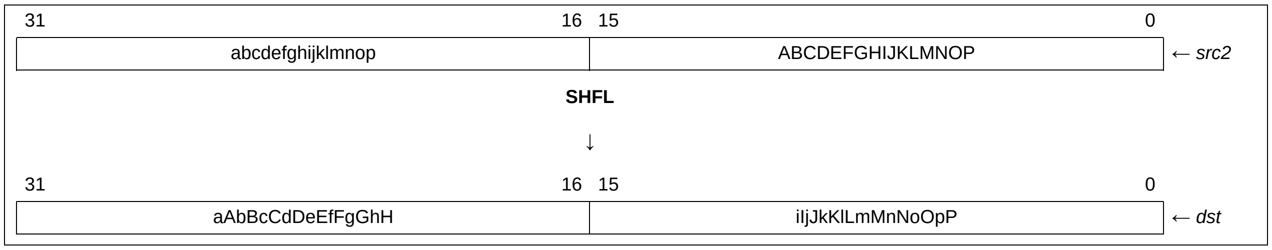
Opcode

31 29	28	27 23	22 18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	1	1	1	0	0	x	0	0	0	0	1	1	1	1	0	0	*s*	*p*
3	1	5	5						1											1	1

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

Description Performs an interleave operation on the two halfwords in *src2*. The bits in the lower halfword of *src2* are placed in the even bit positions in *dst*, and the bits in the upper halfword of *src2* are placed in the odd bit positions in *dst*.

As a result, bits 0, 1, 2, ..., 14, 15 of *src2* are placed in bits 0, 2, 4, ... , 28, 30 of *dst*. Likewise, bits 16, 17, 18, .. 30, 31 of *src2* are placed in bits 1, 3, 5, ..., 29, 31 of *dst*.



NOTE: The **SHFL** instruction is the exact inverse of the **DEAL** instruction (see DEAL).

Execution

```
if (cond)    {
             $src2_{31,30,29...16} \rightarrow dst_{31,29,27...1}$
             $src2_{15,14,13...0} \rightarrow dst_{30,28,26...0}$
             }
else nop
```

Pipeline

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

Instruction Type Two-cycle

Delay Slots 1

See Also DEAL

Example `SHFL .M1 A1,A2`

	Before instruction		2 cycles after instruction
A1	B174 6CA4h	A1	B174 6CA4h
A2	xxxx xxxxh	A2	9E52 6E30h

SHFL3 *3-Way Bit Interleave On Three 16-Bit Values Into a 48-Bit Result*

Syntax **SHFL3** (.unit) *src1, src2, dst_o:dst_e*

unit = .L1 or .L2

Compatibility C64x+ CPU

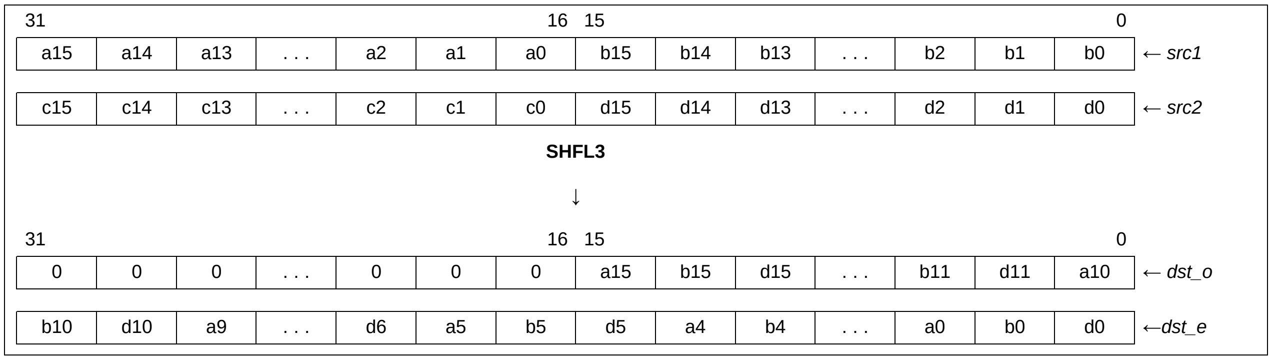
Opcode

31	30	29	28	27 – 23	22 – 18	17 – 13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	*dst*	*src2*	*src1*	x	0	1	1	0	1	1	0	1	1	0	*s*	*p*
				5	5	5	1											1	1

Opcode map field used...	For operand type...	Unit
src1	sint	.L1, .L2
src2	xsint	
dst	dint	

Description Performs a 3-way bit interleave on three 16-bit values and creating a 48-bit result.

This instruction executes unconditionally and cannot be predicated.


31
16 15
0
a15 a14 a13 . . . a2 a1 a0 b15 b14 b13 . . . b2 b1 b0 ← *src1*
c15 c14 c13 . . . c2 c1 c0 d15 d14 d13 . . . d2 d1 d0 ← *src2*
SHFL3
31
16 15
0
0 0 0 . . . 0 0 0 a15 b15 d15 . . . b11 d11 a10 ← *dst_o*
b10 d10 a9 . . . d6 a5 b5 d5 a4 b4 . . . a0 b0 d0 ←*dst_e*

Execution

```
int inp0, inp1, inp2
dword result;
inp0 = src2 & FFFFh;
inp1 = src1 & FFFFh;
inp2 = src1 >> 16 & FFFFh;
result = 0;
for (I = 0; I < 16; I++)
{
        result |= (inp0 >> I & 1) << (I × 3) ;
        result |= (inp1 >> I & 1) << ((I × 3) + 1);
        result |= (inp2 >> I & 1) << I ((I × 3) + 2)
}
```

Instruction Type Single-cycle

Delay Slots 0

Example `SHFL3 .L1 A0,A1,A3:A2`

	Before instruction		1 cycle after instruction
A0	8765 4321h	A2	7E17 9306h
A1	1234 5678h	A3	0000 8C11h

SHL *Arithmetic Shift Left*

Syntax

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

or

SHL (.unit) *src2_h:src2_l, src1, dst_h:dst_l*

unit = .S1 or .S2

Compatibility C62x, C64x, and C64x+ CPU

Compact Instruction Format

Unit	Opcode Format	Figure
.S	S3i	Figure F-22
	Ssh5	Figure F-24
	S2sh	Figure F-25

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 *src2* is a register pair, only the bottom 40 bits of the register pair are shifted. The upper 24 bits of the register pair are unused.

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 & 0xFFFFFF) << 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 ROTL, SHLMB, SHR, SSHL, SSHVL

Examples

Example 1

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

	Before instruction		1 cycle after instruction
A0	29E3 D31Ch	A0	29E3 D31Ch
A1	xxxx xxxxh	A1	9E3D 31C0h

Example 2

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

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

Example 3

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

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

Example 4

```
SHL .S1 A5:A4,0,A1:A0
```

	Before instruction			1 cycle after instruction	
A5:A4	FFFF FFFFh	FFFF FFFFh	A5:A4	FFFF FFFFh	FFFF FFFFh
A1:A0	xxxx xxxxh	xxxx xxxxh	A1:A0	0000 00FFh	FFFF FFFFh

SHLMB *Shift Left and Merge Byte*

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

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

Compatibility C64x and C64x+ CPU

Opcode .L unit

31–29	28	27–23	22–18	17–13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	1	1	0	0	0	0	1	1	1	0	*s*	*p*
3	1	5	5	5	1											1	1

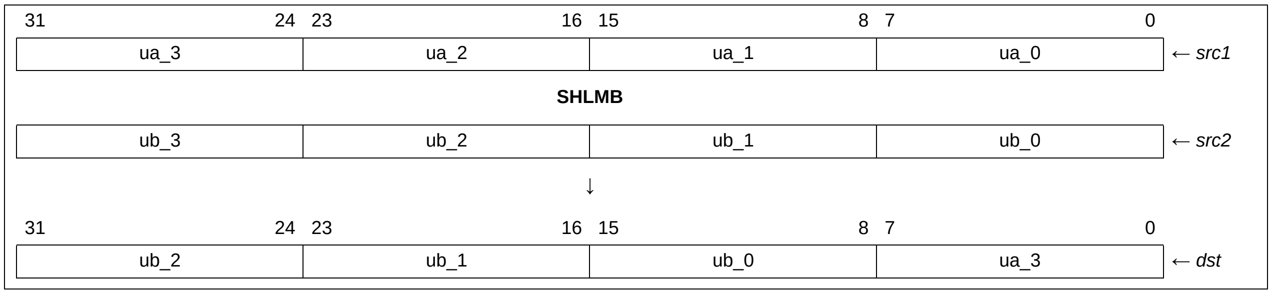
Opcode map field used...	For operand type...	Unit
src1 *src2* *dst*	u4 xu4 u4	.L1, .L2

Opcode .S unit

31–29	28	27–23	22–18	17–13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	1	1	1	0	0	1	1	1	0	0	*s*	*p*
3	1	5	5	5	1											1	1

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

Description Shifts the contents of *src2* left by 1 byte, and then the most-significant byte of *src1* is merged into the least-significant byte position. The result is placed in *dst*.



Execution

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

Pipeline

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

Instruction Type Single-cycle

Delay Slots 0

See Also ROTL, SHL, SHRMB

Examples

Example 1

```
SHLMB .L1 A2, A8, A9
```

	Before instruction		1 cycle after instruction
A2	3789 F23Ah	A2	3789 F23Ah
A8	04B8 4975h	A8	04B8 4975h
A9	xxxx xxxxh	A9	B849 7537h

Example 2

```
SHLMB .S2 B2,B8, B12
```

	Before instruction		1 cycle after instruction
B2	0124 2451h	B2	0124 2451h
B8	01A6 A051h	B8	01A6 A051h
B12	xxxx xxxxh	B12	A6A0 5101h

SHR *Arithmetic Shift Right*

Syntax

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

or

SHR (.unit) *src2_h:src2_l, src1, dst*

unit = .S1 or .S2

Compatibility C62x, C64x, and C64x+ CPU

Compact Instruction Format

Unit	Opcode Format	Figure
.S	S3i	Figure F-22
	Ssh5	Figure F-24
	S2sh	Figure F-25

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 *src2* is a register pair, only the bottom 40 bits of the register pair are shifted. The upper 24 bits of the register pair are unused.

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 & 0xFFFFFF) >>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, SHR2, SHRMB, SHRU, SHRU2, SSHVR

Examples

Example 1

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

	Before instruction		1 cycle after instruction
A0	F123 63D1h	A0	F123 63D1h
A1	xxxx xxxxh	A1	FFF1 2363h

Example 2

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

	Before instruction		1 cycle after instruction
B0	1492 5A41h	B0	1492 5A41h
B1	0000 0012h	B1	0000 0012h
B2	xxxx xxxxh	B2	0000 0524h

Example 3

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

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

Example 4

```
SHR .S1 A5:A4,0,A1:A0
```

	Before instruction			1 cycle after instruction	
A5:A4	FFFF FFFFh	FFFF FFFFh	A5:A4	FFFF FFFFh	FFFF FFFFh
A1:A0	xxxx xxxxh	xxxx xxxxh	A1:A0	0000 00FFh	FFFF FFFFh

SHR2 *Arithmetic Shift Right, Signed, Packed 16-Bit*

Syntax **SHR2** (.unit) *src2, src1, dst*

unit = .S1 or .S2

Compatibility C64x and C64x+ CPU

Opcode .S unit (uint form)

31 29	28	27 23	22 18	17 13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	1	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	uint	.S1, .S2
src2	xs2	
dst	s2	

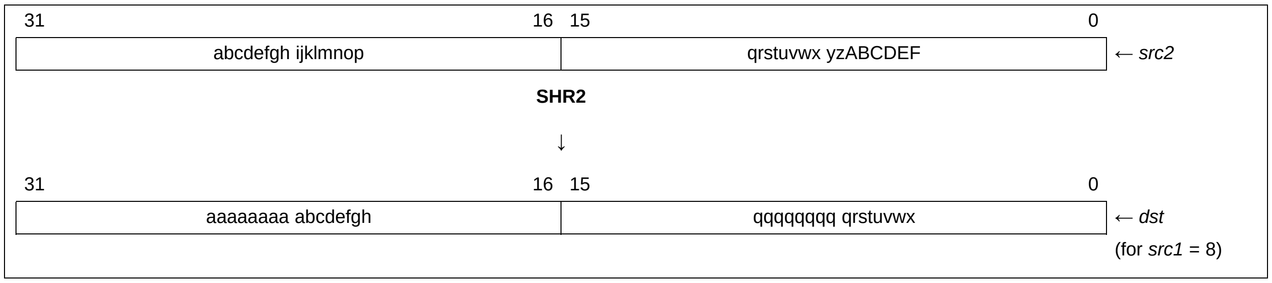
Opcode .S unit (cst form)

31 29	28	27 23	22 18	17 13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	0	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	ucst5	.S1, .S2
src2	xs2	
dst	s2	

Description Performs an arithmetic shift right on signed, packed 16-bit quantities. The values in *src2* are treated as signed, packed 16-bit quantities. The lower 5 bits of *src1* are treated as the shift amount. The results are placed in a signed, packed 16-bit format into *dst*.

For each signed 16-bit quantity in *src2*, the quantity is shifted right by the number of bits specified in the lower 5 bits of *src1*. Bits 5 through 31 of *src1* are ignored and may be non-zero. The shifted quantity is sign-extended, and placed in the corresponding position in *dst*. Bits shifted out of the least-significant bit of the signed 16-bit quantity are discarded.



NOTE: If the shift amount specified in *src1* is in the range 16 to 31, the behavior is identical to a shift value of 15.

Execution

```
if (cond)    {
             smsb16(src2) >> src1 → smsb16(dst);
             slsb16(src2) >> src1 → slsb16(dst)
             }
else nop
```

Pipeline

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

Instruction Type Single-cycle

Delay Slots 0

See Also SHL, SHR, SHRMB, SHRU, SHRU2

Examples

Example 1

```
SHR2 .S2 B2,B4,B5
```

	Before instruction			1 cycle after instruction
B2	A6E2 C179h		B2	A6E2 C179h
B4	1458 3B69h	shift value 9	B4	1458 3B69h
B5	xxxx xxxxh		B5	FFD3 FFE0h

Example 2

```
SHR2 .S1 A4,0fh,A5 ; shift value is 15
```

	Before instruction		1 cycle after instruction
A4	000A 87AFh	A4	000A 87AFh
A5	xxxx xxxxh	A5	0000 FFFFh

SHRMB *Shift Right and Merge Byte*

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

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

Compatibility C64x and C64x+ CPU

Opcode .L unit

31 29	28	27 23	22 18	17 13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	1	1	0	0	0	1	0	1	1	0	*s*	*p*
3	1	5	5	5	1											1	1

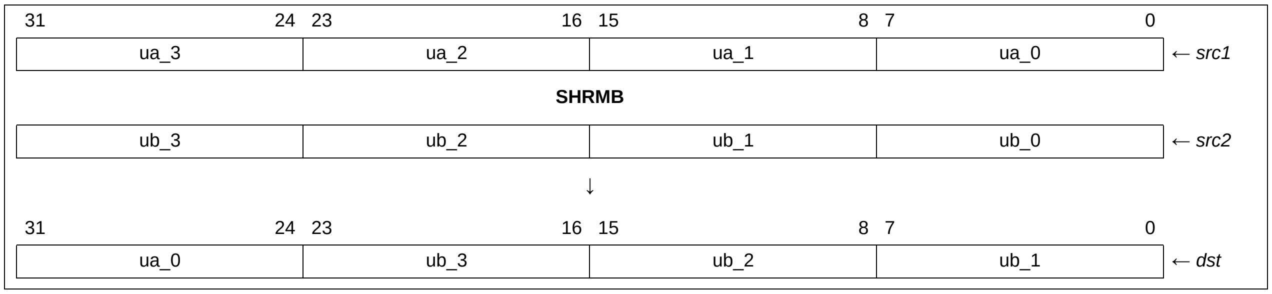
Opcode map field used...	For operand type...	Unit
src1	u4	.L1, .L2
src2	xu4	
dst	u4	

Opcode .S unit

31 29	28	27 23	22 18	17 13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	1	1	1	0	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	u4	.S1, .S2
src2	xu4	
dst	u4	

Description Shifts the contents of *src2* right by 1 byte, and then the least-significant byte of *src1* is merged into the most-significant byte position. The result is placed in *dst*.



Execution

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

Pipeline

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

Instruction Type Single-cycle

Delay Slots 0

See Also SHL, SHLMB, SHR, SHR2, SHRU, SHRU2

Examples

Example 1

```
SHRMB .L1 A2,A8,A9
```

	Before instruction		1 cycle after instruction
A2	3789 F23Ah	A2	3789 F23Ah
A8	04B8 4975h	A8	04B8 4975h
A9	xxxx xxxxh	A9	3A04 B849h

Example 2

```
SHRMB .S2 B2,B8,B12
```

	Before instruction		1 cycle after instruction
B2	0124 2451h	B2	0124 2451h
B8	01A6 A051h	B8	01A6 A051h
B12	xxxx xxxxh	B12	5101 A6A0h

SHRU *Logical Shift Right*

Syntax

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

or

SHRU (.unit) *src2_h:src2_l, src1, dst_h:dst_l*

unit = .S1 or .S2

Compatibility C62x, C64x, and C64x+ CPU

Compact Instruction Format

Unit	Opcode Format	Figure
.S	Ssh5	Figure F-24
	S2sh	Figure F-25

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 *src2* is a register pair, only the bottom 40 bits of the register pair are shifted. The upper 24 bits of the register pair are unused.

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 & 0xFFFFFF) >>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, SHR2, SHRMB, SHRU2

Examples

Example 1

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

	Before instruction		1 cycle after instruction
A0	F123 63D1h	A0	F123 63D1h
A1	xxxx xxxxh	A1	00F1 2363h

Example 2

```
SHRU .S1 A5:A4,0,A1:A0
```

	Before instruction			1 cycle after instruction	
A5:A4	FFFF FFFFh	FFFF FFFFh	A5:A4	FFFF FFFFh	FFFF FFFFh
A1:A0	xxxx xxxxh	xxxx xxxxh	A1:A0	0000 00FFh	FFFF FFFFh

SHRU2 *Arithmetic Shift Right, Unsigned, Packed 16-Bit*

Syntax **SHRU2** (.unit) *src2, src1, dst*

unit = .S1 or .S2

Compatibility C64x and C64x+ CPU

Opcode .S unit (uint form)

31–29	28	27–23	22–18	17–13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	1	1	1	0	0	0	1	1	0	0	*s*	*p*
3	1	5	5	5	1											1	1

Opcode map field used...	For operand type...	Unit
src1	uint	.S1, .S2
src2	xu2	
dst	u2	

Opcode .S unit (cst form)

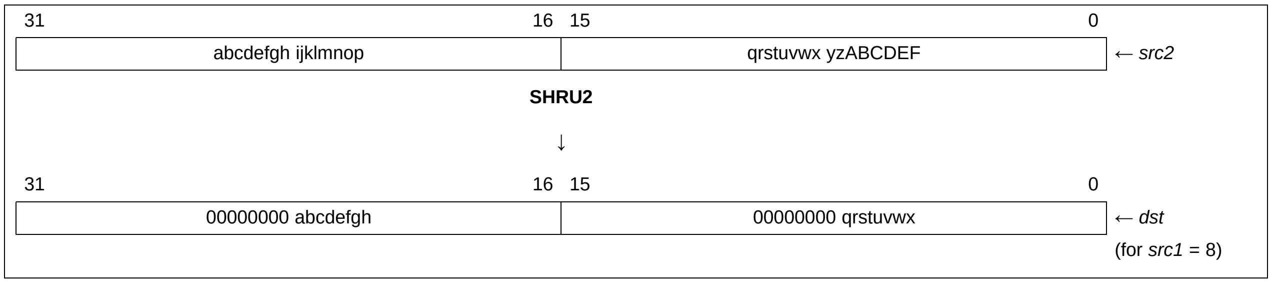
31–29	28	27–23	22–18	17–13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	0	1	1	0	0	1	1	0	0	0	*s*	*p*
3	1	5	5	5	1											1	1

Opcode map field used...	For operand type...	Unit
src1	ucst5	.S1, .S2
src2	xu2	
dst	u2	

Description Performs an arithmetic shift right on unsigned, packed 16-bit quantities. The values in *src2* are treated as unsigned, packed 16-bit quantities. The lower 5 bits of *src1* are treated as the shift amount. The results are placed in an unsigned, packed 16-bit format into *dst*.

For each unsigned 16-bit quantity in *src2*, the quantity is shifted right by the number of bits specified in the lower 5 bits of *src1*. Bits 5 through 31 of *src1* are ignored and may be non-zero. The shifted quantity is zero-extended, and placed in the corresponding position in *dst*. Bits shifted out of the least-significant bit of the signed 16-bit quantity are discarded.

NOTE: If the shift amount specified in *src1* is in the range of 16 to 31, the *dst* will be cleared to all zeros.



Execution

```
if (cond)      {
               umsb16(src2) >> src1 → umsb16(dst);
               ulsb16(src2) >> src1 → ulsb16(dst)
               }
else nop
```

Pipeline

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

Instruction Type Single-cycle

Delay Slots 0

See Also SHL, SHR, SHR2, SHRMB, SHRU

Examples

Example 1

```
SHRU2 .S2 B2,B4,B5
```

	Before instruction			1 cycle after instruction
B2	A6E2 C179h		B2	A6E2 C179h
B4	1458 3B69h	Shift value 9	B4	1458 3B69h
B5	xxxx xxxxh		B5	0053 0060h

Example 2

```
SHRU2 .S1 A4,0Fh,A5 ; Shift value is 15
```

	Before instruction		1 cycle after instruction
A4	000A 87AFh	A4	000A 87AFh
A5	xxxx xxxxh	A5	0000 0001h

SL *Store Linked Word to Buffer*

Syntax **SL** (.unit) *src, *baseR*

unit = .D2

Compatibility C64x+ CPU

NOTE: The atomic operations are not supported on all C64x+ devices, see your device-specific data manual for more information.

Opcode

31–29	28	27–23	22–18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*src*	*baseR*	0	0	0	0	0	0	0	1	1	0	1	1	0	0	0	0	1	*p*
3	1	5	5																		1

Opcode map field used...	For operand type...	Unit
baseR	address	.D2
src	int	

Description The **SL** instruction performs a write of the 32-bit word in *src* to the memory address specified by *baseR*. For linked-operation aware systems, the write request is interpreted as a request to buffer the 32-bit word for use in conjunction with a subsequent **CMTL** operation. When initiating the memory write, if the previously buffered address is not equal to the memory address specified by *baseR* then the link valid flag is cleared. Other than this signaling, the operation of the **SL** instruction from the CPU perspective is identical to that of STW *src, *baseR*.

See Chapter 9 for more details.

Execution

```
if (cond)    src → mem
             signal store-linked operation
else nop
```

Instruction Type Store

Delay Slots 0

See Also **CMTL**, **LL**

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, and C64x+ CPU

Compact Instruction Format

Unit	Opcode Format	Figure
.M	M3	Figure E-5

Opcode

31–29	28	27–23	22–18	17–13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*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 16 least-significant bits of *src1* operand is multiplied by the 16 least-significant bits of 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 (((lsb16(src1) × lsb16(src2)) << 1) != 8000 0000h),
             ((lsb16(src1) × lsb16(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 cycles after instruction	
A1	0000 0123h	291[1]	A1	0000 0123h	
A2	01E0 FA81h	-1407[1]	A2	01E0 FA81h	
A3	xxxx xxxxh		A3	FFF3 8146h	-818,874
CSR	0001 0100h		CSR	0001 0100h	Not saturated
SSR[2]	0000 0000h		SSR	0000 0000h	

(1) Signed 16-LSB integer
(2) Saturation status register (SSR) is only available on the C64x+ DSP.

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, and C64x+ CPU

Compact Instruction Format

Unit	Opcode Format	Figure
.M	M3	Figure E-5

Opcode

31–29	28	27–23	22–18	17–13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*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 16 most-significant bits of *src1* operand is multiplied by the 16 most-significant bits of 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 (((msb16(src1) × msb16(src2)) << 1) != 8000 0000h),
             ((msb16(src1) × msb16(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, and C64x+ CPU

Compact Instruction Format

Unit	Opcode Format	Figure
.M	M3	Figure E-5

Opcode

31–29	28	27–23	22–18	17–13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*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 16 most-significant bits of the *src1* operand is multiplied by the 16 least-significant bits of 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 (((msb16(src1) × lsb16(src2)) << 1) != 8000 0000h),
              ((msb16(src1) × lsb16(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[1]	A1	008A 0000h	
A2	0000 00A7h	167[2]	A2	0000 00A7h	
A3	xxxx xxxxh		A3	0000 B40Ch	46,092
CSR	0001 0100h		CSR	0001 0100h	Not saturated
SSR[3]	0000 0000h		SSR	0000 0000h	

(1) Signed 16-MSB integer
(2) Signed 16-LSB integer
(3) Saturation status register (SSR) is only available on the C64x+ DSP.

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, and C64x+ CPU

Compact Instruction Format

Unit	Opcode Format	Figure
.M	M3	Figure E-5

Opcode

31 29	28	27 23	22 18	17 13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*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 *src2* *dst*	slsb16 xsmsb16 sint	.M1, .M2

Description The 16 least-significant bits of the *src1* operand is multiplied by the 16 most-significant bits of 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 (((lsb16(src1) × msb16(src2)) << 1) != 8000 0000h),
                ((lsb16(src1) × msb16(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	-32,768[1]	A1	0000 8000h	
A2	8000 0000h	-32,768[2]	A2	8000 0000h	
A3	xxxx xxxxh		A3	7FFF FFFFh	2,147,483,647
CSR	0001 0100h		CSR	0001 0300h	Saturated
SSR[3]	0000 0000h		SSR	0000 0010h	

(1) Signed 16-LSB integer
(2) Signed 16-MSB integer
(3) Saturation status register (SSR) is only available on the C64x+ DSP.

SMPY2 *Multiply Signed by Signed, 16 LSB × 16 LSB and 16 MSB × 16 MSB With Left Shift and Saturation*

Syntax **SMPY2** (.unit) *src1, src2, dst_o:dst_e*

unit = .M1 or .M2

Compatibility C64x and C64x+ CPU

Opcode

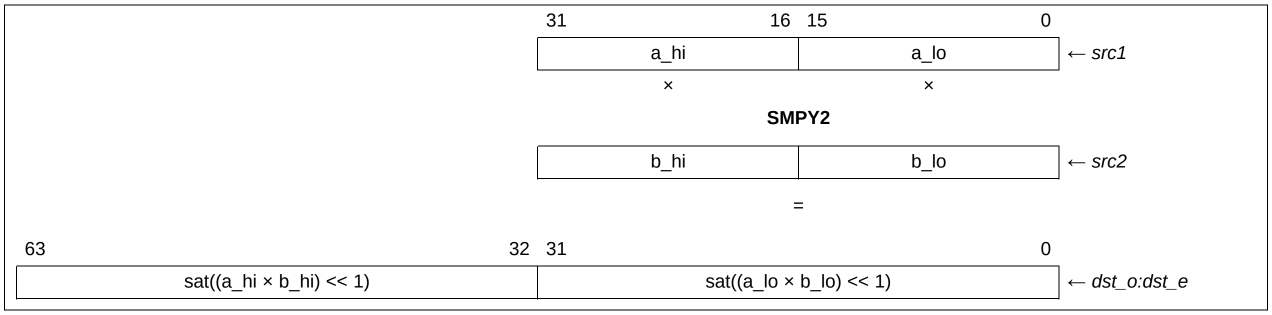
31–29	28	27–23	22–18	17–13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	0	0	0	0	0	1	1	1	0	0	*s*	*p*
3	1	5	5	5	1											1	1

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

Description Performs two 16-bit by 16-bit multiplies between two pairs of signed, packed 16-bit values, with an additional left-shift and saturate. The values in *src1* and *src2* are treated as signed, packed 16-bit quantities. The two 32-bit results are written into a 64-bit register pair.

The **SMPY2** instruction produces two 16 × 16 products. Each product is shifted left by 1. If the left-shifted result is 8000 0000h, the output value is saturated to 7FFF FFFFh.

The saturated product of the lower halfwords of *src1* and *src2* is written to the even destination register, *dst_e*. The saturated product of the upper halfwords of *src1* and *src2* is written to the odd destination register, *dst_o*.



NOTE: If either product saturates, the SAT bit is set in CSR one cycle after the cycle that the result is written to *dst_o:dst_e*. If neither product saturates, the SAT bit in CSR remains unaffected.

The **SMPY2** instruction helps reduce the number of instructions required to perform two 16-bit by 16-bit saturated multiplies on both the lower and upper halves of two registers.

The following code:

```
SMPY    .M1   A0, A1, A2
SMPYH   .M1   A0, A1, A3
```

may be replaced by:

```
SMPY2   .M1   A0, A1, A3:A2
```

Execution

```
if (cond)    {
             sat((lsb16(src1) × lsb16(src2)) << 1) → dst_e;
             sat((msb16(src1) × msb16(src2)) << 1) → dst_o
             }
else nop
```

Pipeline

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

Instruction Type Four-cycle

Delay Slots 3

See Also MPY2, SMPY

Examples

Example 1

```
SMPY2 .M1 A5,A6,A9:A8
```

	Before instruction			4 cycles after instruction	
A5	6A32 1193h	27186 4499	A5	6A32 1193h	
A6	B174 6CA4h	-20108 27812	A6	B174 6CA4h	
A9:A8	xxxx xxxxh	xxxx xxxxh	A9:A8	BED5 6150h	0EEA 8C58h
				-1,093,312,176	250,252,376

Example 2

```
SMPY2 .M2 B2, B5, B9:B8
```

	Before instruction			4 cycles after instruction	
B2	1234 3497h	4660 13463	B2	1234 3497h	
B5	21FF 50A7h	8703 20647	B5	21FF 50A7h	
B9:B8	xxxx xxxxh	xxxx xxxxh	B9:B8	04D5 AB98h	2122 FD02h
				81,111,960	555,941,122

SMPY32 — *Multiply Signed 32-Bit × Signed 32-Bit Into 64-Bit Result With Left Shift and Saturation*

Syntax

SMPY32 (.unit) *src1, src2, dst*

unit = .M1 or .M2

Compatibility

C64x+ CPU

Opcode

31	30	29	28	27 23	22 18	17 13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	*dst*	*src2*	*src1*	x	0	1	1	0	0	1	1	1	0	0	*s*	*p*
				5	5	5	1											1	1

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

Description

Performs a 32-bit by 32-bit multiply. *src1* and *src2* are signed 32-bit values. The 64-bit result is shifted left by 1 with saturation, and the 32 most-significant bits of the shifted value are written to *dst*.

If the result saturates either on the multiply or the shift, the M1 or M2 bit in SSR and the SAT bit in CSR are written one cycle after the results are written to *dst*.

This instruction executes unconditionally and cannot be predicated.

NOTE: When both inputs are 8000 0000h, the shifted result cannot be represented as a 32-bit signed value. In this case, the saturation value 7FFF FFFFh is written into *dst*.

Execution

msb32(sat((*src2* × *src1*) << 1)) → *dst*

Instruction Type Four-cycle

Delay Slots 3

See Also MPY32, SMPY2

Examples

Example 1

```
SMPY32 .M1 A0,A1,A2
```

	Before instruction		4 cycle after instruction
A0	8765 4321h	A2	EED8 ED1Ah
A1	1234 5678h		
CSR	0001 0100h	CSR[1]	0001 0100h
SSR	0000 0000h	SSR[1]	0000 0000h

(1) CSR.SAT and SSR.M1 unchanged by operation

Example 2

```
SMPY32 .L1 A0,A1,A2
```

	Before instruction		4 cycles after instruction
A0	8000 0000h	A2	7FFF FFFFh
A1	8000 0000h		
CSR	0001 0100h	CSR[1]	0001 0300h
SSR	0000 0000h	SSR[1]	0000 0010h

(1) CSR.SAT and SSR.M1 set to 1, 5 cycles after instruction

SPACK2 *Saturate and Pack Two 16 LSBs Into Upper and Lower Register Halves*

Syntax **SPACK2** (.unit) *src1, src2, dst*

unit = .S1 or .S2

Compatibility C64x and C64x+ CPU

Opcode

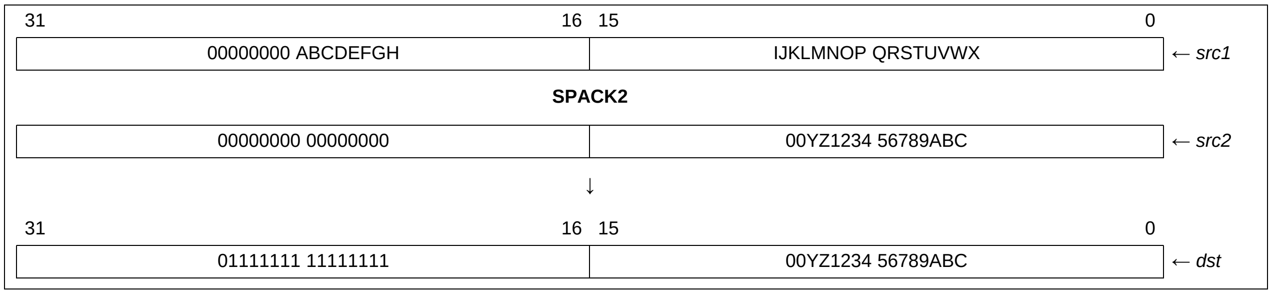
31–29	28	27–23	22–18	17–13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	1	1	0	0	1	0	1	1	0	0	*s*	*p*
3	1	5	5	5	1											1	1

Opcode map field used...	For operand type...	Unit
src1	int	.S1, .S2
src2	xint	
dst	s2	

Description Takes two signed 32-bit quantities in *src1* and *src2* and saturates them to signed 16-bit quantities. The signed 16-bit results are then packed into a signed, packed 16-bit format and written to *dst*. Specifically, the saturated 16-bit signed value of *src1* is written to the upper halfword of *dst,* and the saturated 16-bit signed value of *src2* is written to the lower halfword of *dst*.

Saturation is performed on each input value independently. The input values start as signed 32-bit quantities, and are saturated to 16-bit quantities according to the following rules:

- If the value is in the range - 2^{15} to 2^{15} - 1, inclusive, then no saturation is performed and the value is truncated to 16 bits.
- If the value is greater than 2^{15} - 1, then the result is set to 2^{15} - 1.
- If the value is less than - 2^{15}, then the result is set to - 2^{15}.



The **SPACK2** instruction is useful in code that manipulates 16-bit data at 32-bit precision for its intermediate steps, but that requires the final results to be in a 16-bit representation. The saturate step ensures that any values outside the signed 16-bit range are clamped to the high or low end of the range before being truncated to 16 bits.

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

Execution

```
if (cond)    {
             if (src2 > 0000 7FFFh), 7FFFh → lsb16(dst) or
             if (src2 < FFFF 8000h), 8000h → lsb16(dst)
                 else truncate(src2) → lsb16(dst);
             if (src1 > 0000 7FFFh), 7FFFFh→ msb16(dst) or
             if (src1 < FFFF 8000h), 8000h→ msb16(dst)
                 else truncate(src1) → msb16(dst)
             }
else nop
```

Pipeline

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

Instruction Type Single-cycle

Delay Slots 0

See Also PACK2, PACKH2, PACKHL2, PACKLH2, RPACK2, SPACKU4

Examples

Example 1

```
SPACK2 .S1 A2,A8,A9
```

	Before instruction			1 cycle after instruction	
A2	3789 F23Ah	931,787,322	A2	3789 F23Ah	
A8	04B8 4975h	79,186,293	A8	04B8 4975h	
A9	xxxx xxxxh		A9	7FFF 7FFFh	32767 32767

Example 2

```
SPACK2 .S2 B2,B8,B12
```

	Before instruction			1 cycle after instruction	
B2	A124 2451h	-1,591,466,927	B2	A124 2451h	
B8	01A6 A051h	27,697,233	B8	01A6 A051h	
B12	xxxx xxxxh		B12	8000 7FFFh	-32768 32767

SPACKU4 *Saturate and Pack Four Signed 16-Bit Integers Into Four Unsigned 8-Bit Halfwords*

Syntax **SPACKU4** (.unit) *src1, src2, dst*

unit = .S1 or .S2

Compatibility C64x and C64x+ CPU

Opcode

31 29	28	27 23	22 18	17 13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	1	1	0	1	0	0	1	1	0	0	*s*	*p*
3	1	5	5	5	1											1	1

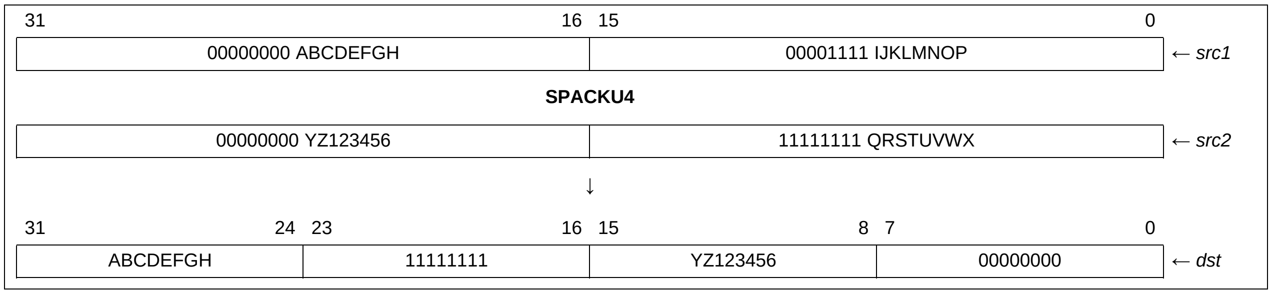
Opcode map field used...	For operand type...	Unit
src1	s2	.S1, .S2
src2	xs2	
dst	u4	

Description Takes four signed 16-bit values and saturates them to unsigned 8-bit quantities. The values in *src1* and *src2* are treated as signed, packed 16-bit quantities. The results are written into *dst* in an unsigned, packed 8-bit format.

Each signed 16-bit quantity in *src1* and *src2* is saturated to an unsigned 8-bit quantity as described below. The resulting quantities are then packed into an unsigned, packed 8-bit format. Specifically, the upper halfword of *src1* is used to produce the most-significant byte of *dst*. The lower halfword of *src1* is used to produce the second most-significant byte (bits 16 to 23) of *dst*. The upper halfword of *src2* is used to produce the third most-significant byte (bits 8 to 15) of *dst*. The lower halfword of *src2* is used to produce the least-significant byte of *dst*.

Saturation is performed on each signed 16-bit input independently, producing separate unsigned 8-bit results. For each value, the following tests are applied:

- If the value is in the range 0 to 2^8 - 1, inclusive, then no saturation is performed and the result is truncated to 8 bits.
- If the value is greater than 2^8 - 1, then the result is set to 2^8 - 1.
- If the value is less than 0, the result is cleared to 0.



The **SPACKU4** instruction is useful in code that manipulates 8-bit data at 16-bit precision for its intermediate steps, but that requires the final results to be in an 8-bit representation. The saturate step ensures that any values outside the unsigned 8-bit range are clamped to the high or low end of the range before being truncated to 8 bits.

NOTE: This operation is performed on each 8-bit quantity separately. This instruction does not affect the SAT bit in CSR.

Execution

```
if (cond)     {
              if (msb16(src1) >> 0000 00FFh), FFh → ubyte3(dst) or
              if (msb16(src1) << 0), 0 → ubyte3(dst)
                  else truncate(msb16(src1)) → ubyte3(dst);
              if (lsb16(src1) >> 0000 00FFh), FFh → ubyte2(dst) or
              if (lsb16(src1) << 0), 0 → ubyte2(dst)
                  else truncate(lsb16(src1)) → ubyte2(dst);
              if (msb16(src2) >> 0000 00FFh), FFh → ubyte1(dst) or
              if (msb16(src2) << 0), 0 → ubyte1(dst)
                  else truncate(msb16(src2)) → ubyte1(dst);
              if (lsb16(src2) >> 0000 00FFh), FFh → ubyte0(dst) or
              if (lsb16(src2) << 0), 0 → ubyte0(dst)
                  else truncate(lsb16(src2)) → ubyte0(dst)
              }
else nop
```

Pipeline

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

Instruction Type Single-cycle

Delay Slots 0

See Also PACKH4, PACKL4, SPACK2

Examples

Example 1

```
SPACKU4 .S1 A2,A8,A9
```

	Before instruction			1 cycle after instruction	
A2	3789 F23Ah	14217 -3526	A2	3789 F23Ah	
A8	04B8 4975h	1208 18805	A8	04B8 4975h	
A9	xxxx xxxxh		A9	FF 00 FF FFh	255 0 255 255

Example 2

```
SPACKU4 .S2 B2,B8,B12
```

	Before instruction			1 cycle after instruction	
B2	A124 2451h	-24284 9297	B2	A124 2451h	
B8	01A6 A051h	422 -24495	B8	01A6 A051h	
B12	xxxx xxxxh		B12	00 FF FF 00h	0 255 255 0

SPKERNEL *Software Pipelined Loop (SPLOOP) Buffer Operation Code Boundary*

Syntax **SPKERNEL** (*fstg, fcyc*)

unit = none

Compatibility C64x+ CPU

Compact Instruction Format

Unit	Opcode Format	Figure
none	Uspk	Figure H-7

Opcode

31	30	29	28	27 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	*fstg/fcyc*	0	0	0	0	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	*s*	*p*
				6																					1	1

Description The **SPKERNEL** instruction is placed in parallel with the last execute packet of the SPLOOP code body indicating there are no more instructions to load into the loop buffer. The **SPKERNEL** instruction also controls at what point in the epilog the execution of post-SPLOOP instructions begins. This point is specified in terms of stage and cycle counts, and is derived from the *fstg/fcyc* field.

The stage and cycle values for both the post-SPLOOP fetch and reload cases are derived from the *fstg/fcyc* field. The 6-bit field is interpreted as a function of the *ii* value from the associated **SPLOOP(D)** instruction. The number of bits allocated to stage and cycle vary according to *ii*. The value for cycle starts from the least-significant end; the value for stage starts from the most-significant end, and they grow together. The number of epilog stages and the number of cycles within those stages are shown in Table 3-23. The exact bit allocation to stage and cycle is shown in Table 3-24.

The following restrictions apply to the use of the **SPKERNEL** instruction:

- The **SPKERNEL** instruction must be the first instruction in the execute packet containing it.
- The **SPKERNEL** instruction cannot be placed in the same execute packet as any instruction that initiates multicycle NOPs. This includes **BNOP**, **CALLP**, **NOP** *n* (*n* > 1), and protected loads (see compact instruction discussion in Section 3.9).
- The **SPKERNEL** instruction cannot be placed in the execute packet immediately following an execute packet containing any instruction that initiates multicycle NOPs. This includes **BNOP**, **CALLP**, **NOP** *n* (*n* > 1), and protected loads (see compact instruction discussion in Section 3.9).
- The **SPKERNEL** instruction cannot be placed in parallel with **DINT** or **RINT** instructions.
- The **SPKERNEL** instruction cannot be placed in parallel with **SPMASK, SPMASKR, SPLOOP, SPLOOPD,** or **SPLOOPW** instructions.
- When the **SPKERNEL** instruction is used with the **SPLOOPW** instruction, *fstg* and *fcyc* should both be zero.

NOTE: The delay specified by the **SPKERNEL** *fstg/fcyc* parameters will not extend beyond the end of the kernel epilog. If the end of the kernel epilog is reached prior to the end of the delay specified by *fstg/fcyc* parameters due to either an excessively large value specified for parameters or due to an early exit from the loop, program fetch will begin immediately and the value specified by the *fstg/fcyc* will be ignored.

Table 3-23. Field Allocation in stg/cyc Field

ii	Number of Bits for Stage	Number of Bits for Cycle
1	6	0
2	5	1
3-4	4	2
5-8	3	3
9-14	2	4

Table 3-24. Bit Allocations to Stage and Cycle in stg/cyc Field

ii	stg/cyc[5]	stg/cyc[4]	stg/cyc[3]	stg/cyc[2]	stg/cyc[1]	stg/cyc[0]
1	stage[0]	stage[1]	stage[2]	stage[3]	stage[4]	stage[5]
2	stage[0]	stage[1]	stage[2]	stage[3]	stage[4]	cycle[0]
3-4	stage[0]	stage[1]	stage[2]	stage[3]	cycle[1]	cycle[0]
5-8	stage[0]	stage[1]	stage[2]	cycle[2]	cycle[1]	cycle[0]
9-14	stage[0]	stage[1]	cycle[3]	cycle[2]	cycle[1]	cycle[0]

Execution See Chapter 7 for more information

See Also SPKERNELR

SPKERNELR *Software Pipelined Loop (SPLOOP) Buffer Operation Code Boundary*

Syntax **SPKERNELR**

unit = none

Compatibility C64x+ CPU

Opcode

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	0	0	0	0	0	0	0	0	0	0	1	1	0	1	1	0	0	0	0	0	0	0	0	0	0	0	*s*	*p*
																														1	1

Description The **SPKERNELR** instruction is placed in parallel with the last execute packet of the SPLOOP code body indicating there are no more instructions to load into the loop buffer. The **SPKERNELR** instruction also indicates that the execution of both post-SPLOOP instructions and instructions reloaded from the buffer begin in the first cycle of the epilog.

The following restrictions apply to the use of the **SPKERNELR** instruction:

- The **SPKERNELR** instruction must be the first instruction in the execute packet containing it.
- The **SPKERNELR** instruction cannot be placed in the same execute packet as any instruction that initiates multicycle NOPs. This includes **BNOP**, **CALLP**, **NOP** *n* (*n* > 1), and protected loads (see compact instruction discussion in Section 3.9).
- The **SPKERNELR** instruction cannot be placed in the execute packet immediately following an execute packet containing any instruction that initiates multicycle NOPs. This includes **BNOP**, **CALLP**, **NOP** *n* (*n* > 1), and protected loads (see compact instruction discussion in Section 3.9).
- The **SPKERNELR** instruction cannot be placed in parallel with **DINT** or **RINT** instructions.
- The **SPKERNELR** instruction cannot be placed in parallel with **SPMASK, SPMASKR, SPLOOP, SPLOOPD,** or **SPLOOPW** instructions.
- The **SPKERNELR** instruction can only be used when the **SPLOOP** instruction that began the SPLOOP buffer operation was predicated.
- The **SPKERNELR** instruction cannot be paired with an **SPLOOPW** instruction.

This instruction executes unconditionally and cannot be predicated.

Execution See Chapter 7 for more information.

See Also SPKERNEL

SPLOOP *Software Pipelined Loop (SPLOOP) Buffer Operation*

Syntax **SPLOOP** *ii*

unit = none

Compatibility C64x+ CPU

Compact Instruction Format

Unit	Opcode Format	Figure
none	Uspl	Figure H-5

Opcode

31 29	28	27 23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*ii - 1*	0	0	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	*s*	*p*
3	1	5																						1	1

Description The **SPLOOP** instruction invokes the loop buffer mechanism. See Chapter 7 for more details.

When the **SPLOOP** instruction is predicated, it indicates that the loop is a nested loop using the SPLOOP reload capability. The decision of whether to reload is determined by the predicate register selected by the *creg* and *z* fields.

The following restrictions apply to the use of the **SPLOOP** instruction:

- The **SPLOOP** instruction must be the first instruction in the execute packet containing it.
- The **SPLOOP** instruction cannot be placed in the same execute packet as any instruction that initiates multicycle NOPs. This includes **BNOP, CALLP**, **NOP** *n* (*n* > 1), and protected loads (see compact instruction discussion in Section 3.9).
- The **SPLOOP** instruction cannot be placed in parallel with **DINT** or **RINT** instructions.
- The **SPLOOP** instruction cannot be placed in parallel with **SPMASK**, **SPMASKR, SPKERNEL,** or **SPKERNELR** instructions.

Execution See Chapter 7 for more information.

See Also SPLOOPD, SPLOOPW

SPLOOPD *Software Pipelined Loop (SPLOOP) Buffer Operation With Delayed Testing*

Syntax **SPLOOPD** *ii*

unit = none

Compatibility C64x+ CPU

Compact Instruction Format

Unit	Opcode Format	Figure
none	Uspl	Figure H-5
	Uspldr	Figure H-6

Opcode

31 29	28	27 23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*ii - 1*	0	0	0	0	0	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	*s*	*p*
3	1	5																						1	1

Description The **SPLOOPD** instruction invokes the loop buffer mechanism. The testing of the termination condition is delayed for four cycles. See Chapter 7 for more details.

When the **SPLOOPD** instruction is predicated, it indicates that the loop is a nested loop using the SPLOOP reload capability. The decision of whether to reload is determined by the predicate register selected by the *creg* and *z* fields.

The following restrictions apply to the use of the **SPLOOPD** instruction:

- The **SPLOOPD** instruction must be the first instruction in the execute packet containing it.
- The **SPLOOPD** instruction cannot be placed in the same execute packet as any instruction that initiates multicycle NOPs. This includes **BNOP**, **CALLP**, **NOP** *n* (*n* > 1), and protected loads (see compact instruction discussion in Section 3.9).
- The **SPLOOPD** instruction cannot be placed in parallel with **DINT** or **RINT** instructions.
- The **SPLOOPD** instruction cannot be placed in parallel with **SPMASK**, **SPMASKR, SPKERNEL,** or **SPKERNELR** instructions.

Execution See Chapter 7 for more information.

See Also SPLOOP, SPLOOPW

SPLOOPW *Software Pipelined Loop (SPLOOP) Buffer Operation With Delayed Testing and No Epilog*

Syntax **SPLOOPW** *ii*

unit = none

Compatibility C64x+ CPU

Opcode

31–29	28	27–23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*ii - 1*	0	0	0	0	0	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	*s*	*p*
3	1	5																						1	1

Description The **SPLOOPW** instruction invokes the loop buffer mechanism. The testing of the termination condition is delayed for four cycles. See Chapter 7 for more details.

The **SPLOOPW** instruction is always predicated. The termination condition is the value of the predicate register selected by the *creg* and *z* fields.

The following restrictions apply to the use of the **SPLOOPW** instruction:

- The **SPLOOPW** instruction must be the first instruction in the execute packet containing it.
- The **SPLOOPW** instruction cannot be placed in the same execute packet as any instruction that initiates multicycle NOPs. This includes **BNOP**, **NOP** *n* (*n* > 1), and protected loads (see compact instruction discussion in Section 3.9).
- The **SPLOOPW** instruction cannot be placed in parallel with **DINT** or **RINT** instructions.
- The **SPLOOPW** instruction cannot be placed in parallel with **SPMASK**, **SPMASKR, SPKERNEL,** or **SPKERNELR** instructions.

Execution See Chapter 7 for more information.

See Also SPLOOP, SPLOOPD

SPMASK *Software Pipelined Loop (SPLOOP) Buffer Operation Load/Execution Control*

Syntax **SPMASK** *unitmask*

unit = none

Compatibility C64x+ CPU

Compact Instruction Format

Unit	Opcode Format	Figure
none	Uspm	Figure H-8

Opcode

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	0	0	M2	M1	D2	D1	S2	S1	L2	L1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	*s*	*p*
						1	1	1	1	1	1	1	1																	1	1

Description The **SPMASK** instruction serves two purposes within the SPLOOP mechanism:

1. The **SPMASK** instruction inhibits the execution of specified instructions from the buffer within the current execute packet.
2. The **SPMASK** inhibits the loading of specified instructions into the buffer during loading phase, although the instruction will execute normally.

If the SPLOOP is reloading after returning from an interrupt, the **SPMASK**ed instructions coming from the buffer execute, but the **SPMASK**ed instructions from program memory do not execute and are not loaded into the buffer.

An **SPMASK**ed instruction encountered outside of the SPLOOP mechanism shall be treated as a NOP.

The **SPMASK**ed instruction must be the first instruction in the execute packet containing it.

The **SPMASK** instruction cannot be placed in parallel with **SPLOOP**, **SPLOOPD**, **SPKERNEL**, or **SPKERNELR** instructions.

The **SPMASK** instruction executes unconditionally and cannot be predicated.

There are two ways to specify which instructions within the current execute packet will be masked:

1. The functional units of the instruction can be specified as the SPMASK argument.
2. The instruction to be masked can be marked with a caret (^) in the instruction code. The following three examples are equivalent:

```
     SPMASK  D2,L1
||   MV      .D2  B0,B1
||   MV      .L1  A0,A1

     SPMASK  D2
||   MV      .D2  B0,B1
||^  MV      .L1  A0,A1

     SPMASK
||^  MV      .D2  B0,B1
||^  MV      .L1  A0,A1
```

The following two examples mask two **MV** instructions, but do not mask the **MPY** instruction.

```
     SPMASK  D1, D2
||   MV      .D1   A0,A1          ;This unit is SPMASKed
||   MV      .D2   B0,B1          ;This unit is SPMASKed
||   MPY     .L1   A0,B1          ;This unit is Not SPMASKed

     SPMASK
||^  MV      .D1   A0,A1          ;This unit is SPMASKed
||^  MV      .D2   B0,B1          ;This unit is SPMASKed
||   MPY     .L1   A0,B1          ;This unit is Not SPMASKed
```

Execution See Chapter 7

See Also SPMASKR

SPMASKR *Software Pipelined Loop (SPLOOP) Buffer Operation Load/Execution Control*

Syntax **SPMASKR** *unitmask*

unit = none

Compatibility C64x+ CPU

Compact Instruction Format

Unit	Opcode Format	Figure
none	Uspm	Figure H-8

Opcode

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	0	0	M2	M1	D2	D1	S2	S1	L2	L1	1	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	*s*	*p*
						1	1	1	1	1	1	1	1																	1	1

Description The **SPMASKR** instruction serves three purposes within the SPLOOP mechanism. Similar to the **SPMASK** instruction:

1. The **SPMASKR** instruction inhibits the execution of specified instructions from the buffer within the current execute packet.
2. The **SPMASKR** instruction inhibits the loading of specified instructions into the buffer during loading phase, although the instruction will execute normally.

   In addition to the functionality of the **SPMASK** instruction:
3. The **SPMASKR** instruction controls the reload point for nested loops.

The **SPMASKR** instruction is placed in the execute packet (in the post-SPKERNEL code) preceding the execute packet that will overlap with the first cycle of the reload operation.

The **SPKERNELR** and the **SPMASKR** instructions cannot coexist in the same SPLOOP operation. In the case where reload is intended to start in the first epilog cycle, the **SPKERNELR** instruction is used and the **SPMASKR** instruction is not used for that nested loop.

The **SPMASKR** instruction cannot be used in a loop using the **SPLOOPW** instruction.

An **SPMASKR** instruction encountered outside of the SPLOOP mechanism shall be treated as a NOP.

The **SPMASKR** instruction executes unconditionally and cannot be predicated.

The **SPMASKR** instruction must be the first instruction in the execute packet containing it.

The **SPMASKR** instruction cannot be placed in parallel with **SPLOOP**, **SPLOOPD**, **SPKERNEL**, or **SPKERNELR** instructions.

There are two ways to specify which instructions within the current execute packet will be masked:

1. The functional units of the instruction can be specified as the SPMASKR argument.
2. The instruction to be masked can be marked with a caret (^) in the instruction code. The following three examples are equivalent:

```
     SPMASKR  D2,L1
||   MV       .D2  B0,B1
||   MV       .L1  A0,A1

     SPMASKR
||   MV       .D2  B0,B1
||^  MV       .L1  A0,A1

     SPMASKR
||^  MV       .D2  B0,B1
||^  MV       .L1  A0,A1
```

The following two examples mask two **MV** instructions, but do not mask the **MPY** instruction. The presence of a caret (^) in the instruction code specifies which instructions are **SPMASK**ed.

```
     SPMASKR  D1,D2
||   MV       .D1  A0,A1           ;This unit is SPMASKed
||   MV       .D2  B0,B1           ;This unit is SPMASKed
||   MPY      .L1  A0,B1           ;This unit is Ned SPMASKed

     SPMASKR
||^  MV       .D1  A0,A1           ;This unit is SPMASKED
||^  MV       .D2  B0,B1           ;This unit is SPMASKED
||   MPY      .L1  A0,B1           ;This unit is Not SPMASKed
```

Execution See Chapter 7

See Also SPMASK

Example

```
     SPMASKR
||^  LDW      .D1  *A0,A1          ;This unit is SPMASKed
||^  LDW      .D2  *B0,B1          ;This unit is SPMASKed
||   MPY      .M1  A3,A4,A5        ;This unit is Not SPMASKed
```

SSHL *Shift Left With Saturation*

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

unit = .S1 or .S2

Compatibility C62x, C64x, and C64x+ CPU

Compact Instruction Format

Unit	Opcode Format	Figure
.S	Ssh5	Figure F-24
	S2sh	Figure F-25

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 5 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.

NOTE: For the C64x and C64x+ DSP, when a register is used to specify the shift, the 6 least-significant bits specify the shift amount. Valid values are 0 through 63. If the shift count value is greater than 32, then the result is saturated to 32 bits when *src2* is non-zero.

Execution

```
if (cond)      {
               if (bit(31) through bit(31 - src1) of src2 are all 1s or all 0s),
               dst = src2 << src1;
               else if (src2 > 0), saturate dst to 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 ROTL, SHL, SHLMB, SHR, SSHVL

Examples

Example 1

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

	Before instruction		1 cycle after instruction		2 cycles after instruction	
A0	02E3 031Ch	A0	02E3 031Ch	A0	02E3 031Ch	
A1	xxxx xxxxh	A1	0B8C 0C70h	A1	0B8C 0C70h	
CSR	0001 0100h	CSR	0001 0100h	CSR	0001 0100h	Not saturated
SSR[1]	0000 0000h	SSR	0000 0000h	SSR	0000 0000h	

[1] Saturation status register (SSR) is only available on the C64x+ DSP.

Example 2

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

	Before instruction		1 cycle after instruction		2 cycles after instruction	
A0	4719 1925h	A0	4719 1925h	A0	4719 1925h	
A1	0000 0006h	A1	0000 0006h	A1	0000 0006h	
A2	xxxx xxxxh	A2	7FFF FFFFh	A2	7FFF FFFFh	
CSR	0001 0100h	CSR	0001 0100h	CSR	0001 0300h	Saturated
SSR[1]	0000 0000h	SSR	0000 0000h	SSR	0000 0004h	

[1] Saturation status register (SSR) is only available on the C64x+ DSP.

SSHVL *Variable Shift Left*

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

unit = .M1 or .M2

Compatibility C64x and C64x+ CPU

Opcode

31–29	28	27–23	22–18	17–13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	0	1	1	1	0	0	1	1	0	0	*s*	*p*
3	1	5	5	5	1											1	1

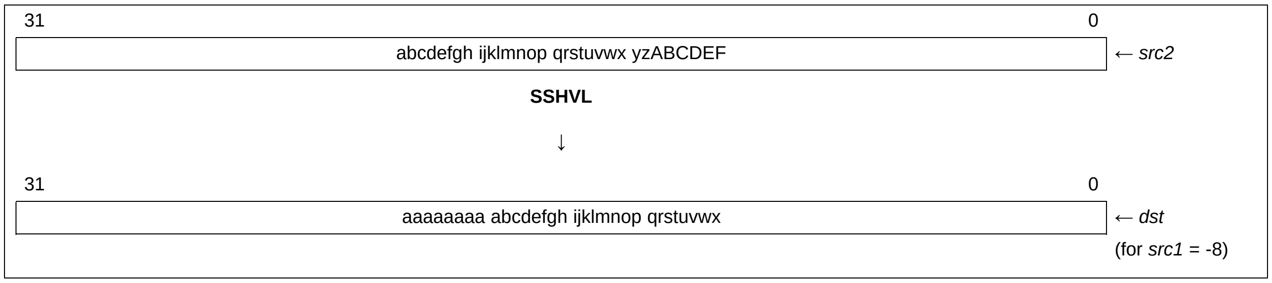
Opcode map field used...	For operand type...	Unit
src1	int	.M1, .M2
src2	xint	
dst	int	

Description Shifts the signed 32-bit value in *src2* to the left or right by the number of bits specified by *src1,* and places the result in *dst*.

The *src1* argument is treated as a 2s-complement shift value which is automatically limited to the range -31 to 31. If *src1* is positive, *src2* is shifted to the left. If *src1* is negative, *src2* is shifted to the right by the absolute value of the shift amount, with the sign-extended shifted value being placed in *dst*. It should also be noted that when *src1* is negative, the bits shifted right past bit 0 are lost.

Saturation is performed when the value is shifted left under the following conditions:

- If the shifted value is in the range -2^{31} to 2^{31} - 1, inclusive, then no saturation is performed, and the result is truncated to 32 bits.
- If the shifted value is greater than 2^{31} - 1, then the result is saturated to 2^{31} - 1.
- If the shifted value is less than - 2^{31}, then the result is saturated to - 2^{31}.


31
0
abcdefgh ijklmnop qrstuvwx yzABCDEF
← src2
SSHVL
↓
31
0
aaaaaaaa abcdefgh ijklmnop qrstuvwx
← dst
(for src1 = -8)

NOTE: If the shifted value is saturated, then the SAT bit is set in CSR one cycle after the result is written to dst. If the shifted value is not saturated, then the SAT bit is unaffected.

Execution

```
if (cond)    {
             if (0 <= src1 <= 31), sat(src2 << src1) → dst ;
             if (-31 <= src1 < 0), (src2 >> abs(src1)) → dst;
             if (src1 > 31), sat(src2 << 31) → dst;
             if (src1 < -31), (src2 >> 31) → dst
             }
else nop
```

Pipeline

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

Instruction Type Two-cycle

Delay Slots 1

See Also SHL, SHLMB, SSHL, SSHVR

Examples

Example 1

```
SSHVL .M2 B2, B4, B5
```

	Before instruction			2 cycles after instruction
B2	FFFF F000h		B2	FFFF F000h
B4	FFFF FFE1h	-31	B4	FFFF FFE1h
B5	xxxx xxxxh		B5	FFFF FFFFh

Example 2

```
SSHVL .M1 A2,A4,A5
```

	Before instruction			2 cycles after instruction	
A2	F14C 2108h		A2	F14C 2108h	
A4	0000 0001Fh	31	A4	0000 0001Fh	
A5	xxxx xxxxh		A5	8000 0000h	Saturated to most negative value

Example 3

```
SSHVL .M2 B12, B24, B25
```

	Before instruction			2 cycles after instruction
B12	187A 65FCh		B12	187A 65FCh
B24	FFFF FFFFh	-1	B24	FFFF FFFFh
B25	xxxx xxxxh		B25	03CD 32FEh

SSHVR *Variable Shift Right*

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

unit = .M1 or .M2

Compatibility C64x and C64x+ CPU

Opcode

31–29	28	27–23	22–18	17–13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	0	1	1	0	1	0	1	1	0	0	*s*	*p*
3	1	5	5	5	1											1	1

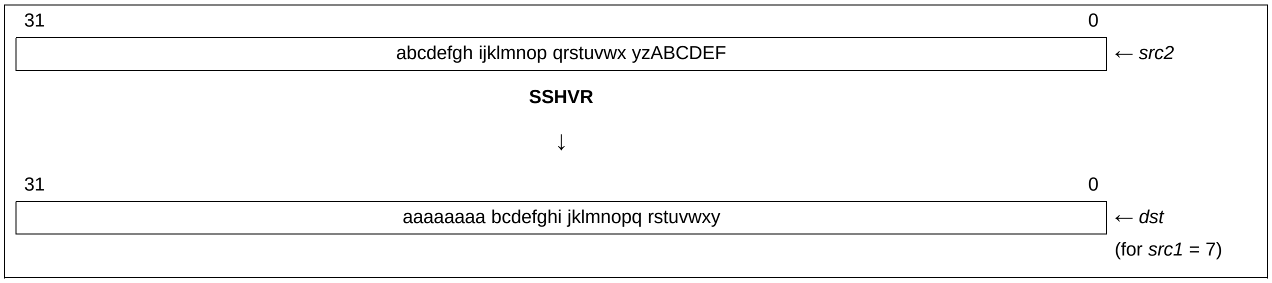
Opcode map field used...	For operand type...	Unit
src1	int	.M1, .M2
src2	xint	
dst	int	

Description Shifts the signed 32-bit value in *src2* to the left or right by the number of bits specified by *src1,* and places the result in *dst*.

The *src1* argument is treated as a 2s-complement shift value that is automatically limited to the range -31 to 31. If *src1* is positive, *src2* is shifted to the right by the value specified with the sign-extended shifted value being placed in *dst*. It should also be noted that when *src1* is positive, the bits shifted right past bit 0 are lost. If *src1* is negative, *src2* is shifted to the left by the absolute value of the shift amount value and the result is placed in *dst*.

Saturation is performed when the value is shifted left under the following conditions:

- If the shifted value is in the range -2^{31} to 2^{31} - 1, inclusive, then no saturation is performed, and the result is truncated to 32 bits.
- If the shifted value is greater than 2^{31} - 1, then the result is saturated to 2^{31} - 1.
- If the shifted value is less than - 2^{31}, then the result is saturated to - 2^{31}.


31
0
abcdefgh ijklmnop qrstuvwx yzABCDEF
← src2
SSHVR
↓
31
0
aaaaaaaa bcdefghi jklmnopq rstuvwxy
← dst
(for src1 = 7)

NOTE: If the shifted value is saturated, then the SAT bit is set in CSR one cycle after the result is written to dst. If the shifted value is not saturated, then the SAT bit is unaffected.

Execution

```
if (cond)     {
              if (0 <= src1 <= 31), (src2 >> src1) → dst;
              if (-31 <= src1 < 0), sat(src2 << abs(src1)) → dst;
              if (src1 > 31), (src2 >> 31) → dst;
              if (src1 < -31), sat(src2 << 31) → dst
              }
else nop
```

Pipeline

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

Instruction Type Two-cycle

Delay Slots 1

See Also SHR, SHR2, SHRMB, SHRU, SHRU2, SSHVL

Examples

Example 1

`SSHVR .M2 B2,B4,B5`

	Before instruction			2 cycles after instruction	
B2	FFFF F000h		B2	FFFF F000h	
B4	FFFF FFE1h	-31	B4	FFFF FFE1h	
B5	xxxx xxxxh		B5	8000 0000h	Saturated to most negative value

Example 2

`SSHVR .M1 A2,A4,A5`

	Before instruction			2 cycles after instruction
A2	F14C 2108h		A2	F14C 2108h
A4	0000 0001Fh	31	A4	0000 0001Fh
A5	xxxx xxxxh		A5	FFFF FFFFh

Example 3

`SSHVR .M2 B12, B24, B25`

	Before instruction			2 cycles after instruction
B12	187A 65FCh		B12	187A 65FCh
B24	FFFF FFFFh	-1	B24	FFFF FFFFh
B25	xxxx xxxxh		B25	30F4 CBF8h

SSUB *Subtract Two Signed Integers With Saturation*

Syntax

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

or

SSUB (.unit) *src1, src2_h:src2_l, dst_h:dst_l*

unit = .L1 or .L2

Compatibility C62x, C64x, and C64x+ CPU

Compact Instruction Format

Unit	Opcode Format	Figure
.L	L3	Figure D-4

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, SSUB2

Examples

Example 1

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

	Before instruction			**1 cycle after instruction**	
B1	5A2E 51A3h	1,512,984,995	B1	5A2E 51A3h	
B2	802A 3FA2h	-2,144,714,846	B2	802A 3FA2h	
B3	xxxx xxxxh		B3	7FFF FFFFh	2,147,483,647
CSR	0001 0100h		CSR	0001 0100h	
SSR[1]	0000 0000h		SSR	0000 0000h	
				2 cycles after instruction	
			B1	5A2E 51A3h	
			B2	802A 3FA2h	
			B3	7FFF FFFFh	
			CSR	0001 0300h	Saturated
			SSR	0000 0002h	

(1) Saturation status register (SSR) is only available on the C64x+ DSP.

Example 2

```
SSUB .L1 A0,A1,A2
```

	Before instruction			1 cycle after instruction	
A0	4367 71F2h	1,130,852,850	A0	4367 71F2h	
A1	5A2E 51A3h	1,512,984,995	A1	5A2E 51A3h	
A2	xxxx xxxxh		A2	E939 204Fh	-382,132,145
CSR	0001 0100h		CSR	0001 0100h	
SSR[1]	0000 0000h		SSR	0000 0000h	
				2 cycles after instruction	
			A0	4367 71F2h	
			A1	5A2E 51A3h	
			A2	E939 204Fh	
			CSR	0001 0100h	Not saturated
			SSR	0000 0000h	

[1] Saturation status register (SSR) is only available on the C64x+ DSP.

SSUB2 *Subtract Two Signed 16-Bit Integers on Upper and Lower Register Halves With Saturation*

Syntax

SSUB2 (.unit) *src1, src2, dst*

unit = .L1 or .L2

Compatibility

C64x+ CPU

Opcode

31–29	28	27–23	22–18	17–13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	1	1	0	0	1	0	0	1	1	0	*s*	*p*
3	1	5	5	5	1											1	1

Opcode map field used...	For operand type...	Unit
src1	s2	.L1, .L2
src2	xs2	
dst	s2	

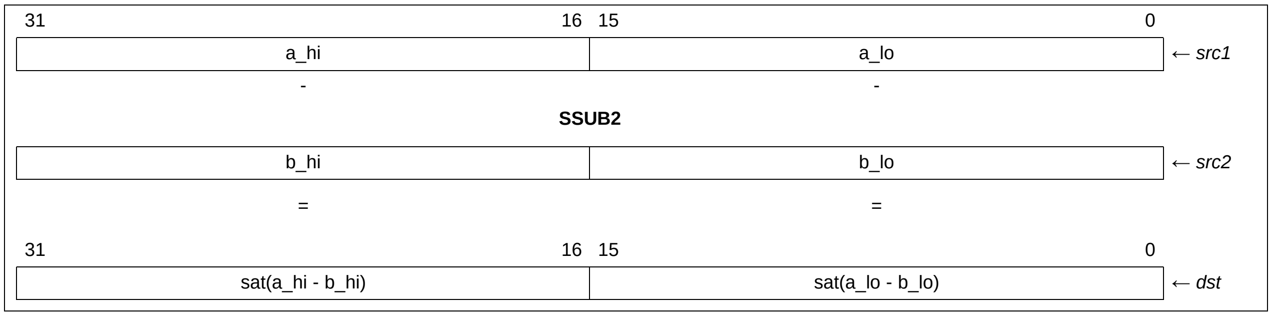
Description

Performs 2s-complement subtraction between signed, packed 16-bit quantities in *src1* and *src2*. The results are placed in a signed, packed 16-bit format into *dst*.

For each pair of 16-bit quantities in *src1* and *src2*, the difference between the signed 16-bit value from *src1* and the signed 16-bit value from *src2* is calculated and saturated to produce a signed 16-bit result. The result is placed in the corresponding position in *dst*.

Saturation is performed on each 16-bit result independently. For each sum, the following tests are applied:

- If the difference is in the range - 2^{15} to 2^{15} - 1, inclusive, then no saturation is performed and the sum is left unchanged.
- If the difference is greater than 2^{15} - 1, then the result is set to 2^{15} - 1.
- If the difference is less than - 2^{15}, then the result is set to - 2^{15}.



NOTE: This operation is performed on each halfword separately. This instruction does not affect the SAT bit in CSR or the L1 or L2 bit in SSR.

Execution

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

Instruction Type Single-cycle

Delay Slots 0

See Also ADD2, SUB, SUB4, SSUB2

Examples

Example 1

```
SSUB2 .L1 A0,A1,A2
```

	Before instruction		1 cycle after instruction
A0	0007 0005h	A2	0008 0006h
A1	FFFF FFFFh		

Example 2

```
SSUB2 .L1 A0,A1,A2
```

	Before instruction		1 cycle after instruction
A0	0007 0005h	A2	7FFF 0006h
A1	8000 FFFFh		

STB *Store Byte to Memory With a 5-Bit Unsigned Constant Offset or Register Offset*

Syntax

Register Offset
STB (.unit) *src*, **+baseR[offsetR]*

Unsigned Constant Offset
STB (.unit) *src*, **+baseR[ucst5]*

unit = .D1 or .D2

Compatibility

C62x, C64x, and C64x+ CPU

Compact Instruction Format

Unit	Opcode Format	Figure
.D	Doff4	Figure C-9
	Dind	Figure C-11
	Dinc	Figure C-13
	Ddec	Figure C-15

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*	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 3-6 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 2.8.3).

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.

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

Examples

Example 1

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

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

Example 2

```
STB .D1 A8,*++A4[5]
```

	Before instruction		1 cycle after instruction		3 cycles after instruction
A4	0000 4020h	A4	0000 4025h	A4	0000 4025h
A8	0123 4567h	A8	0123 4567h	A8	0123 4567h
mem 4024:27h	xxxx xxxxh	mem 4024:27h	xxxx xxxxh	mem 4024:27h	xxxx 67xxh

Example 3

```
STB .D1 A8,*A4++[5]
```

	Before instruction		1 cycle after instruction		3 cycles after instruction
A4	0000 4020h	A4	0000 4025h	A4	0000 4025h
A8	0123 4567h	A8	0123 4567h	A8	0123 4567h
mem 4020:23h	xxxx xxxxh	mem 4020:23h	xxxx xxxxh	mem 4020:23h	xxxx xx67h

Example 4

```
STB .D1 A8,*++A4[A12]
```

	Before instruction		1 cycle after instruction		3 cycles after instruction
A4	0000 4020h	A4	0000 4026h	A4	0000 4026h
A8	0123 4567h	A8	0123 4567h	A8	0123 4567h
A12	0000 0006h	A12	0000 0006h	A12	0000 0006h
mem 4024:27h	xxxx xxxxh	mem 4024:27h	xxxx xxxxh	mem 4024:27h	xx67 xxxxh

STB — *Store Byte to Memory With a 15-Bit Unsigned Constant Offset*

Syntax

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

unit = .D2

Compatibility

C62x, C64x, and C64x+ CPU

Opcode

31–29	28	27–23	22–8	7	6	5	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	B1	1234 5678h	B1	1234 5678h
B14	0000 1000h	B14	0000 1000h	B14	0000 1000h
mem 1028h	42h	mem 1028h	42h	mem 1028h	78h

STDW *Store Doubleword to Memory With a 5-Bit Unsigned Constant Offset or Register Offset*

Syntax

Register Offset
STDW (.unit) *src*, **+baseR[offsetR]*

Unsigned Constant Offset
STDW (.unit) *src*, **+baseR[ucst5]*

unit = .D1 or .D2

Compatibility

C64x and C64x+ CPU

Compact Instruction Format

Unit	Opcode Format	Figure
.D	Doff4DW	Figure C-10
	DindDW	Figure C-12
	DincDW	Figure C-14
	DdecDW	Figure C-16
	Dpp	Figure C-22

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

Opcode map field used...	For operand type...	Unit
src *baseR* *offsetR*	ullong uint uint	.D1, .D2
src *baseR* *offsetR*	ullong uint ucst5	.D1, .D2

Description

Stores a 64-bit quantity to memory from a 64-bit register, *src*. Table 3-6 describes the addressing generator options. Alignment to a 64-bit boundary is required. 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.

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

The *offsetR/ucst5* is scaled by a left shift of 3 bits. After scaling, *offsetR/ucst5* is added to, or subtracted from, *baseR*. For the 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 from memory.

The addressing arithmetic that performs the additions and subtractions defaults to linear mode. However, for A4-A7 and for B4-B7, the mode can be changed to circular mode by writing the appropriate value to the AMR (see Section 2.8.3).

The *src* pair can be in either register file, regardless of the .D unit or *baseR* or *offsetR* used. The *s* bit determines which file *src* will be loaded from: *s* = 0 indicates *src* will be in the A register file and *s* = 1 indicates *src* will be in the B register file.

Assembler Notes

When no bracketed register or constant is specified, the assembler defaults increments and decrements to 1 and offsets to 0. Stores that do no modification to the *baseR* can use the assembler syntax *R. Square brackets, [ ], indicate that the *ucst5* offset is left-shifted by 3 for doubleword stores.

Parentheses, ( ), can be used to tell the assembler that the offset is a non-scaled, constant offset. The assembler right shifts the constant by 3 bits for doubleword stores before using it for the *ucst5* field. After scaling by the **STDW** instruction, this results in the same constant offset as the assembler source if the least-significant three bits are zeros.

For example, **STDW** (.unit) *src*, *+*baseR* (16) represents an offset of 16 bytes (2 doublewords), and the assembler writes out the instruction with *ucst5* = 2. **STDW** (.unit) *src*, *+*baseR* [16] represents an offset of 16 doublewords, or 128 bytes, and the assembler writes out the instruction with *ucst5* = 16.

Either brackets or parentheses must be typed around the specified offset if the optional offset parameter is used. The register pair syntax always places the odd-numbered register first, a colon, followed by the even-numbered register (that is, A1:A0, B1:B0, A3:A2, B3:B2, etc.).

Execution

if (cond)	*src* → mem
else nop	

Pipeline

Pipeline Stage	E1
Read	*baseR, offsetR*, *src*
Written	*baseR*
Unit in use	.D

Instruction Type Store

Delay Slots 0

See Also LDDW, STW

Examples

Example 1

```
STDW .D1 A3:A2,*A0++
```

	Before instruction			1 cycle after instruction	
A0	0000 1000h		A0	0000 1008h	
A3:A2	A176 3B28h	6041 AD65h	A3:A2	A176 3B28h	6041 AD65h

Byte Memory Address	1009	1008	1007	1006	1005	1004	1003	1002	1001	1000
Data Value Before Store	00	00	00	00	00	00	00	00	00	00
Data Value After Store	00	00	A1	76	3B	28	60	41	AD	65

Example 2

```
STDW .D1 A3:A2, *A0++
```

	Before instruction			1 cycle after instruction	
A0	0000 1004h		A0	0000 100Ch	
A3:A2	A176 3B28h	6041 AD65h	A3:A2	A176 3B28h	6041 AD65h

Byte Memory Address	100D	100C	100B	100A	1009	1008	1007	1006	1005	1004	1003
Data Value Before Store	00	00	00	00	00	00	00	00	00	00	00
Data Value After Store	00	00	A1	76	3B	28	60	41	AD	65	00

Example 3

```
STDW .D1 A9:A8, *++A4[5]
```

	Before instruction			1 cycle after instruction	
A4	0000 4020h		A4	0000 4048h	
A9:A8	ABCD EF98h	0123 4567h	A9:A8	ABCD EF98h	0123 4567h

Byte Memory Address	4051	4050	404F	404E	404D	404C	404B	404A	4049	4048	4047
Data Value Before Store	00	00	00	00	00	00	00	00	00	00	00
Data Value After Store	00	00	AB	CD	EF	98	01	23	45	67	00

Example 4

```
STDW .D1 A9:A8, *++A4(16)
```

	Before instruction			1 cycle after instruction	
A4	0000 4020h		A4	0000 4030h	
A9:A8	ABCD EF98h	0123 4567h	A9:A8	ABCD EF98h	0123 4567h

Byte Memory Address	**4039**	**4038**	**4037**	**4036**	**4035**	**4034**	**4033**	**4032**	**4031**	**4030**	**402F**
Data Value Before Store	00	00	00	00	00	00	00	00	00	00	00
Data Value After Store	00	00	AB	CD	EF	98	01	23	45	67	00

Example 5

```
STDW .D1 A9:A8, *++A4[A12]
```

	Before instruction			1 cycle after instruction	
A4	0000 4020h		A4	0000 4030h	
A9:A8	ABCD EF98h	0123 4567h	A9:A8	ABCD EF98h	0123 4567h
A12	0000 0006h		A12	0000 0006h	

Byte Memory Address	**4059**	**4058**	**4057**	**4056**	**4055**	**4054**	**4053**	**4052**	**4051**	**4050**	**404F**
Data Value Before Store	00	00	00	00	00	00	00	00	00	00	00
Data Value After Store	00	00	AB	CD	EF	98	01	23	45	67	00

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, and C64x+ CPU

Compact Instruction Format

Unit	Opcode Format	Figure
.D	Doff4	Figure C-9
	Dind	Figure C-11
	Dinc	Figure C-13
	Ddec	Figure C-15

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	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 3-6 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 2.8.3).

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.

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

Examples

Example 1

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

	Before instruction		1 cycle after instruction		3 cycles after instruction
A1	9A32 7634h	A1	9A32 7634h	A1	9A32 7634h
B10	0000 1000h	A10	0000 1000h	A10	0000 1000h
mem 104h	1134h	mem 104h	1134h	mem 104h	7634h

Example 2

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

	Before instruction		1 cycle after instruction		3 cycles after instruction
A1	9A32 2634h	A1	9A32 2634h	A1	9A32 2634h
A10	0000 0100h	A10	0000 00F8h	A10	0000 00F8h
A11	0000 0004h	A11	0000 0004h	A11	0000 0004h
mem F8h	0000h	mem F8h	0000h	mem F8h	0000h
mem 100h	0000h	mem 100h	0000h	mem 100h	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, and C64x+ CPU

Opcode

31 29	28	27 23	22 8	7	6	5	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

STNDW *Store Nonaligned Doubleword to Memory With a 5-Bit Unsigned Constant Offset or Register Offset*

Syntax

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

unit = .D1 or .D2

Compatibility

C64x and C64x+ CPU

Compact Instruction Format

Unit	Opcode Format	Figure
.D	Doff4DW	Figure C-10
	DindDW	Figure C-12
	DincDW	Figure C-14
	DdecDW	Figure C-16

Opcode

31 29	28	27 24	23	22 18	17 13	12 9	8	7	6	5	4	3	2	1	0
creg	*z*	*src*	*sc*	*baseR*	*offsetR/ucst5*	*mode*	1	*y*	1	1	1	0	1	*s*	*p*
3	1	4	1	5	5	4		1						1	1

Opcode map field used...	For operand type...	Unit
src *baseR* *offsetR*	ullong uint uint	.D1, .D2
src *baseR* *offsetR*	ullong uint ucst5	.D1, .D2

Description

Stores a 64-bit quantity to memory from a 64-bit register pair, *src*. Table 3-6 describes the addressing generator options. The **STNDW** instruction may write a 64-bit value to any byte boundary. Thus alignment to a 64-bit boundary is not required. The memory address is formed from a base address register (*baseR*) and an optional offset that is either a register (*offsetR*) or a 5-bit unsigned constant (*ucst5*).

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

The **STNDW** instruction supports both scaled offsets and non-scaled offsets. The *sc* field is used to indicate whether the *offsetR/ucst5* is scaled or not. If *sc* is 1 (scaled), the *offsetR/ucst5* is shifted left 3 bits before adding or subtracting from the *baseR*. If *sc* is 0 (nonscaled), the *offsetR/ucst5* is not shifted before adding to or subtracting from the *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 post-decrement addressing, the value of *baseR* before the addition or subtraction is the address to be accessed from memory.

The addressing arithmetic that performs the additions and subtractions defaults to linear mode. However, for A4-A7 and for B4-B7, the mode can be changed to circular mode by writing the appropriate value to the AMR (see Section 2.8.3).

The *src* pair can be in either register file, regardless of the .D unit or *baseR* or *offsetR* used. The s bit determines which file *src* will be loaded from: *s* = 0 indicates *src* will be in the A register file and *s* = 1 indicates *src* will be in the B register file.

NOTE: No other memory access may be issued in parallel with a nonaligned memory access. The other .D unit can be used in parallel, as long as it is not performing a memory access.

Assembler Notes

When no bracketed register or constant is specified, the assembler defaults increments and decrements to 1, and offsets to 0. Loads that do no modification to the *baseR* can use the assembler syntax *R. Square brackets, [ ], indicate that the *ucst5* offset is left-shifted by 3 for doubleword stores.

Parentheses, ( ), can be used to indicate to the assembler that the offset is a nonscaled offset.

For example, **STNDW** (.unit) *src*, *+*baseR* (12) represents an offset of 12 bytes and the assembler writes out the instruction with *offsetC* = 12 and *sc* = 0.

STNDW (.unit) *src*, *+*baseR* [16] represents an offset of 16 doublewords, or 128 bytes, and the assembler writes out the instruction with *offsetC* = 16 and *sc* = 1.

Either brackets or parentheses must be typed around the specified offset if the optional offset parameter is used.

Execution

```
if (cond)    src → mem
else nop
```

Pipeline

Pipeline Stage	E1
Read	*baseR*, *offsetR*, *src*
Written	*baseR*
Unit in use	.D

Instruction Type Store

Delay Slots 0

See Also LDNW, LDNDW, STNW

Examples

Example 1

```
STNDW .D1 A3:A2, *A0++
```

	Before instruction			1 cycle after instruction	
A0	0000 1001h		A0	0000 1009h	
A3	A176 3B28h	6041 AD65h	A3:A2	A176 3B28h	6041 AD65h

Byte Memory Address	**1009**	**1008**	**1007**	**1006**	**1005**	**1004**	**1003**	**1002**	**1001**	**1000**
Data Value Before Store	00	00	00	00	00	00	00	00	00	00
Data Value After Store	00	A1	76	3B	28	60	41	AD	65	00

Example 2

```
STNDW .D1 A3:A2, *A0++
```

	Before instruction			1 cycle after instruction	
A0	0000 1003h		A0	0000 100Bh	
A3:A2	A176 3B28h	6041 AD65h	A3:A2	A176 3B28h	6041 AD65h

Byte Memory Address	**100B**	**100A**	**1009**	**1008**	**1007**	**1006**	**1005**	**1004**	**1003**	**1002**	**1001**	**1000**
Data Value Before Store	00	00	00	00	00	00	00	00	00	00	00	00
Data Value After Store	00	A1	76	3B	28	60	41	AD	65	00	00	00

STNW *Store Nonaligned Word to Memory With a 5-Bit Unsigned Constant Offset or Register Offset*

Syntax

Register Offset	**Unsigned Constant Offset**
STNW (.unit) *src*, *+*baseR[offsetR]*	**STNW** (.unit) *src*, *+*baseR[ucst5]*
unit = .D1 or .D2	

Compatibility

C64x and C64x+ CPU

Compact Instruction Format

Unit	Opcode Format	Figure
.D	Doff4	Figure C-9
	Dind	Figure C-11
	Dinc	Figure C-13
	Ddec	Figure C-15

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

Opcode map field used...	For operand type...	Unit
src *baseR* *offsetR*	uint uint uint	.D1, .D2
src *baseR* *offsetR*	uint uint ucst5	.D1, .D2

Description

Stores a 32-bit quantity to memory from a 32-bit register, *src*. Table 3-6 describes the addressing generator options. The **STNW** instruction may write a 32-bit value to any byte boundary. Thus alignment to a 32-bit boundary is not required. The memory address is formed from a base address register (*baseR*) and an optional offset that is either a register (*offsetR*) or a 5-bit unsigned constant (*ucst5*).

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

The *offsetR/ucst5* is scaled by a left shift of 2 bits. After scaling, *offsetR/ucst5* is added to, or subtracted from, *baseR*. For the 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 from memory.

The addressing arithmetic that performs the additions and subtractions defaults to linear mode. However, for A4-A7 and for B4-B7, the mode can be changed to circular mode by writing the appropriate value to the AMR (see Section 2.8.3).

The *src* can be in either register file, regardless of the .D unit or *baseR* or *offsetR* used. The *s* bit determines which file *src* will be loaded from: *s* = 0 indicates *src* will be in the A register file and *s* = 1 indicates *src* will be in the B register file.

NOTE: No other memory access may be issued in parallel with a nonaligned memory access. The other .D unit can be used in parallel as long as it is not performing memory access.

Assembler Notes

When no bracketed register or constant is specified, the assembler defaults increments and decrements to 1 and offsets to 0. Loads that do no modification to the *baseR* can use the assembler syntax *R. Square brackets, [ ], indicate that the *ucst5* offset is left-shifted by 2 for word stores.

Parentheses, ( ), can be used to tell the assembler that the offset is a non-scaled, constant offset. The assembler right shifts the constant by 2 bits for word stores before using it for the *ucst5* field. After scaling by the **STNW** instruction, this results in the same constant offset as the assembler source if the least-significant two bits are zeros.

For example, **STNW** (.unit) *src*,*+*baseR* (12) represents an offset of 12 bytes (3 words), and the assembler writes out the instruction with *ucst5* = 3.

STNW (.unit) *src*,*+*baseR* [12] represents an offset of 12 words, or 48 bytes, and the assembler writes out the instruction with *ucst5* = 12.

Either brackets or parentheses must be typed around the specified offset if the optional offset parameter is used.

Execution

if (cond)	*src* → mem
else nop	

Pipeline

Pipeline Stage	E1
Read	*baseR*, *offsetR*, *src*
Written	*baseR*
Unit in use	.D

Instruction Type Store

Delay Slots 0

See Also LDNW, LDNDW, STNDW

Examples

Example 1

```
STNW .D1 A3, *A0++
```

	Before instruction		1 cycle after instruction
A0	0000 1001h	A0	0000 1005h
A3	A176 3B28h	A3	A176 3B28h

Byte Memory Address	1007	1006	1005	1004	1003	1002	1001	1000
Data Value Before Store	00	00	00	00	00	00	00	00
Data Value After Store	00	00	00	A1	76	3B	28	00

Example 2

```
STNW .D1 A3, *A0++
```

	Before instruction		1 cycle after instruction
A0	0000 1003h	A0	0000 1007h
A3	A176 3B28h	A3	A176 3B28h

Byte Memory Address	1007	1006	1005	1004	1003	1002	1001	1000
Data Value Before Store	00	00	00	00	00	00	00	00
Data Value After Store	00	A1	76	3B	28	00	00	00

STW *Store Word to Memory With a 5-Bit Unsigned Constant Offset or Register Offset*

Syntax

Register Offset
STW (.unit) *src*, **+baseR[offsetR]*

Unsigned Constant Offset
STW (.unit) *src*, **+baseR[ucst5]*

unit = .D1 or .D2

Compatibility

C62x, C64x, and C64x+ CPU

Compact Instruction Format

Unit	Opcode Format	Figure
.D	Doff4	Figure C-9
	Dind	Figure C-11
	Dinc	Figure C-13
	Ddec	Figure C-15

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 3-6 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 2.8.3).

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.

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* → 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

Examples

Example 1

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

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

Example 2

```
STW .D1 A8,*++A4[5]
```

	Before instruction		1 cycle after instruction		3 cycles after instruction
A4	0000 4020h	A4	0000 4034h	A4	0000 4034h
A8	0123 4567h	A8	0123 4567h	A8	0123 4567h
mem 4020h	xxxx xxxxh	mem 4020h	xxxx xxxxh	mem 4020h	xxxx xxxxh
mem 4034h	xxxx xxxxh	mem 4034h	xxxx xxxxh	mem 4034h	0123 4567h

Example 3

```
STW .D1 A8,*++A4(8)
```

	Before instruction		1 cycle after instruction		3 cycles after instruction
A4	0000 4020h	A4	0000 4028h	A4	0000 4028h
A8	0123 4567h	A8	0123 4567h	A8	0123 4567h
mem 4020h	xxxx xxxxh	mem 4020h	xxxx xxxxh	mem 4020h	xxxx xxxxh
mem 4028h	xxxx xxxxh	mem 4028h	xxxx xxxxh	mem 4028h	0123 4567h

Example 4

```
STW .D1 A8,*++A4[A12]
```

	Before instruction		1 cycle after instruction		3 cycles after instruction
A4	0000 4020h	A4	0000 4038h	A4	0000 4038h
A8	0123 4567h	A8	0123 4567h	A8	0123 4567h
A12	0000 0006h	A12	0000 0006h	A12	0000 0006h
mem 4020h	xxxx xxxxh	mem 4020h	xxxx xxxxh	mem 4020h	xxxx xxxxh
mem 4038h	xxxx xxxxh	mem 4038h	xxxx xxxxh	mem 4038h	0123 4567h

STW *Store Word to Memory With a 15-Bit Unsigned Constant Offset*

Syntax **STW**(.unit) *src*, *+B14/B15[*ucst15*]

unit = .D2

Compatibility C62x, C64x, and C64x+ CPU

Compact Instruction Format

Unit	Opcode Format	Figure
.D	Dstk	Figure C-17
	Dpp	Figure C-22

Opcode

31–29	28	27–23	22–8	7	6	5	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 (.L1 or .L2) *src1, src2, dst_h:dst_l*

or

SUB (.D1 or .D2) *src2, src1, dst* (if the cross path form is not used)

or

SUB (.D1 or .D2) *src1, src2, dst* (if the cross path form is used)

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

Compatibility C62x, C64x, and C64x+ CPU

Compact Instruction Format

Unit	Opcode Format	Figure
.L	L3	Figure D-4
	Lx1	Figure D-11
.S	S3	Figure F-21
	Sx2op	Figure F-28
	Sx1	Figure F-30
.D	Dx2op	Figure C-18
	Dx1	Figure C-21

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*.

The .D unit, when the cross path form is not used, 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.

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

src2 - src1:

31 29	28	27 23	22 18	17 13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	1	1	0	1	0	1	1	1	0	0	*s*	*p*
3	1	5	5	5	1											1	1

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

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 (if the cross path form is not used)

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

Execution

```
if (cond)     src2 - src1 → dst
else nop
```

Opcode .D unit (if the cross path form is used)

31 29	28	27 23	22 18	17 13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	1	0	1	1	0	0	1	1	0	0	*s*	*p*
3	1	5	5	5	1											1	1

Opcode map field used...	For operand type...	Unit
src1 *src2* *dst*	sint xsint sint	.D1, .D2

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

Execution

```
if (cond)     src1 - src2 → dst
else nop
```

Pipeline

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

Instruction Type Single-cycle

Delay Slots 0

See Also ADD, NEG, SUBC, SUBU, SSUB, SUB2

Example `SUB .L1 A1,A2,A3`

	Before instruction			1 cycle after instruction	
A1	0000 325Ah	12,890	A1	0000 325Ah	
A2	FFFF FF12h	-238	A2	FFFF FF12h	
A3	xxxx xxxxh		A3	0000 3348h	13,128

SUBAB *Subtract Using Byte Addressing Mode*

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

unit = .D1 or .D2

Compatibility C62x, C64x, and C64x+ 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 2.8.3).The result is placed in *dst*.

Execution

```
if (cond)    src2 - 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]		1 cycle after instruction
A0	0000 0004h	A0	0000 0004h
A5	0000 4000h	A5	0000 400Ch
AMR	0003 0004h	AMR	0003 0004h

[1] BK0 = 3 → size = 16
A5 in circular addressing mode using BK0

SUBABS4 *Subtract With Absolute Value, Four 8-Bit Pairs for Four 8-Bit Results*

Syntax **SUBABS4** (.unit) *src1, src2, dst*

unit = .L1 or .L2

Compatibility C64x and C64x+ CPU

Opcode

31 29	28	27 23	22 18	17 13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	1	0	1	1	0	1	0	1	1	0	*s*	*p*
3	1	5	5	5	1											1	1

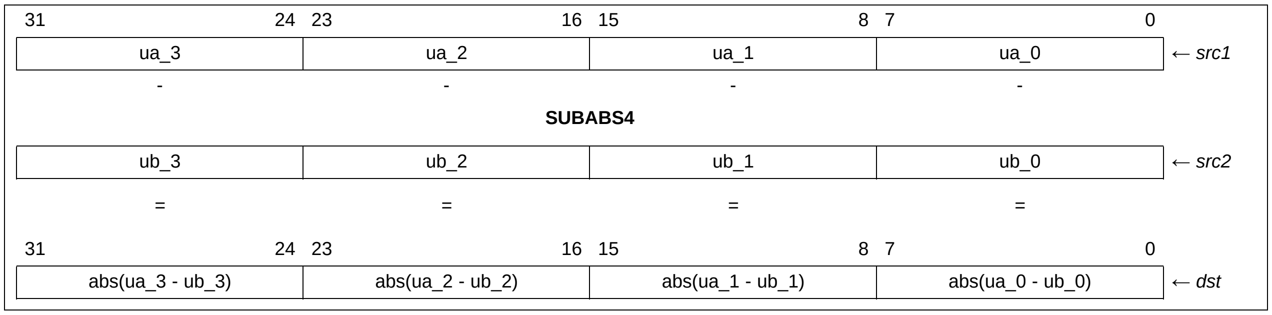
Opcode map field used...	For operand type...	Unit
src1	u4	.L1, .L2
src2	xu4	
dst	u4	

Description Calculates the absolute value of the differences between the packed 8-bit data contained in the source registers. The values in *src1* and *src2* are treated as unsigned, packed 8-bit quantities. The result is written into *dst* in an unsigned, packed 8-bit format.

For each pair of unsigned 8-bit values in *src1* and *src2*, the absolute value of the difference is calculated. This result is then placed in the corresponding position in *dst*.

- The absolute value of the difference between *src1* byte0 and *src2* byte0 is placed in byte0 of *dst*.
- The absolute value of the difference between *src1* byte1 and *src2* byte1 is placed in byte1 of *dst*.
- The absolute value of the difference between *src1* byte2 and *src2* byte2 is placed in byte2 of *dst*.
- The absolute value of the difference between *src1* byte3 and *src2* byte3 is placed in byte3 of *dst*.

The **SUBABS4** instruction aids in motion-estimation algorithms, and other algorithms, that compute the "best match" between two sets of 8-bit quantities.


31 24 23 16 15 8 7 0
ua_3 ua_2 ua_1 ua_0 ← src1
- - - -
SUBABS4
ub_3 ub_2 ub_1 ub_0 ← src2
= = = =
31 24 23 16 15 8 7 0
abs(ua_3 - ub_3) abs(ua_2 - ub_2) abs(ua_1 - ub_1) abs(ua_0 - ub_0) ← dst

Execution

```
if (cond)    {
             abs(ubyte0(src1) - ubyte0(src2)) → ubyte0(dst);
             abs(ubyte1(src1) - ubyte1(src2)) → ubyte1(dst);
             abs(ubyte2(src1) - ubyte2(src2)) → ubyte2(dst);
             abs(ubyte3(src1) - ubyte3(src2)) → ubyte3(dst)
             }
else nop
```

Pipeline

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

Instruction Type Single-cycle

Delay Slots 0

See Also ABS, SUB, SUB4

Example `SUBABS4 .L1 A2, A8, A9`

	Before instruction			1 cycle after instruction	
A2	37 89 F2 3Ah	55 137 242 58 unsigned	A2	37 89 F2 3Ah	
A8	04 B8 49 75h	4 184 73 117 unsigned	A8	04 B8 49 75h	
A9	xxxx xxxxh		A9	33 2F A9 3Bh	51 47 169 59 unsigned

SUBAH *Subtract Using Halfword Addressing Mode*

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

unit = .D1 or .D2

Compatibility C62x, C64x, and C64x+ 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 2.8.3). *src1* is left shifted by 1. The result is placed in *dst*.

Execution

```
if (cond)    src2 - src1<<1 → 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, and C64x+ CPU

Compact Instruction Format

Unit	Opcode Format	Figure
.D	Dx5p	Figure C-20

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 2.8.3). *src1* is left shifted by 2. The result is placed in *dst*.

Execution

```
if (cond)    src2 - src1<<2 → 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]		1 cycle after instruction
A3	xxxx xxxxh	A3	0000 0108h
A5	0000 0100h	A5	0000 0100h
AMR	0003 0004h	AMR	0003 0004h

[1] 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, and C64x+ CPU

Opcode

31–29	28	27–23	22–18	17–13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*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	uint	.L1, .L2
src2	xuint	
dst	uint	

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, SUBU, SUB2

Examples

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	137,777	A0	0000 47E5h	18,405
A1	0001 F63Fh	128,575	A1	0001 F63Fh	

SUBU *Subtract Two Unsigned Integers Without Saturation*

Syntax

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

or

SUBU (.unit) *src1, src2, dst_h:dst_l*

unit = .L1 or .L2

Compatibility C62x, C64x, and C64x+ 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 → 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, SUB2

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

	Before instruction				1 cycle after instruction		
A1	0000 325Ah	12,890[1]		A1	0000 325Ah		
A2	FFFF FF12h	4,294,967,058[1]		A2	FFFF FF12h		
A5:A4	xxxx xxxxh	xxxx xxxxh		A5:A4	0000 00FFh	0000 3348h	-4,294,954,168[2]

(1) Unsigned 32-bit integer
(2) Signed 40-bit (long) integer

SUB2 *Subtract Two 16-Bit Integers on Upper and Lower Register Halves*

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

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

Compatibility C62x, C64x, and C64x+ CPU

Opcode .L unit

31–29	28	27–23	22–18	17–13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	0	0	0	0	1	0	0	1	1	0	*s*	*p*
3	1	5	5	5	1											1	1

Opcode map field used...	For operand type...	Unit
src1	i2	.L1, .L2
src2	xi2	
dst	i2	

Opcode .S unit

31–29	28	27–23	22–18	17–13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	0	1	0	0	0	1	1	0	0	0	*s*	*p*
3	1	5	5	5	1											1	1

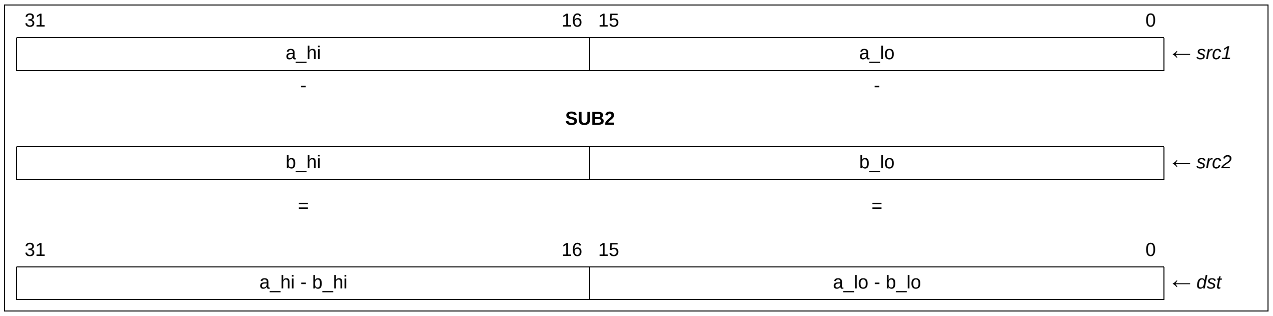
Opcode map field used...	For operand type...	Unit
src1	i2	.S1, .S2
src2	xi2	
dst	i2	

Opcode .D unit

31–29	28	27–23	22–18	17–13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	1	0	0	1	0	1	1	1	0	0	*s*	*p*
3	1	5	5	5	1											1	1

Opcode map field used...	For operand type...	Unit
src1	i2	.D1, .D2
src2	xi2	
dst	i2	

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*.



NOTE: Unlike the **SUB** instruction, the argument ordering on the .D unit form of .S2 is consistent with the argument ordering for the .L and .S unit forms.

Execution

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

Pipeline

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

Instruction Type Single-cycle

Delay Slots 0

See Also ADD2, SUB, SUBU, SUB4, SSUB2

Examples

Example 1

```
SUB2 .S1 A3, A4, A5
```

	Before instruction			1 cycle after instruction	
A3	1105 6E30h	4357 28208	A3	1105 6E30h	
A4	1105 6980h	4357 27008	A4	1105 6980h	
A5	xxxx xxxxh		A5	0000 04B0h	0 1200

Example 2

```
SUB2 .D2 B2, B8, B15
```

	Before instruction			1 cycle after instruction	
B2	F23A 3789h	-3526 14217	B2	F23A 3789h	
B8	04B8 6732h	1208 26418	B8	04B8 6732h	
B15	xxxx xxxxh		B15	ED82 D057h	-4734 -12201

Example 3

```
SUB2 .S2X B1,A0,B2
```

	Before instruction			1 cycle after instruction	
A0	0021 3271h	33[1] 12913[2]	A0	0021 3271h	
B1	003A 1B48h	58[1] 6984[2]	B1	003A 1B48h	
B2	xxxx xxxxh		B2	0019 E8D7h	25[1] -5929[2]

(1) Signed 16-MSB integer
(2) Signed 16-LSB integer

SUB4 *Subtract Without Saturation, Four 8-Bit Pairs for Four 8-Bit Results*

Syntax

SUB4 (.unit) *src1, src2, dst*

unit = .L1 or .L2

Compatibility C64x and C64x+ CPU

Opcode

31–29	28	27–23	22–18	17–13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	1	1	0	0	1	1	0	1	1	0	*s*	*p*
3	1	5	5	5	1											1	1

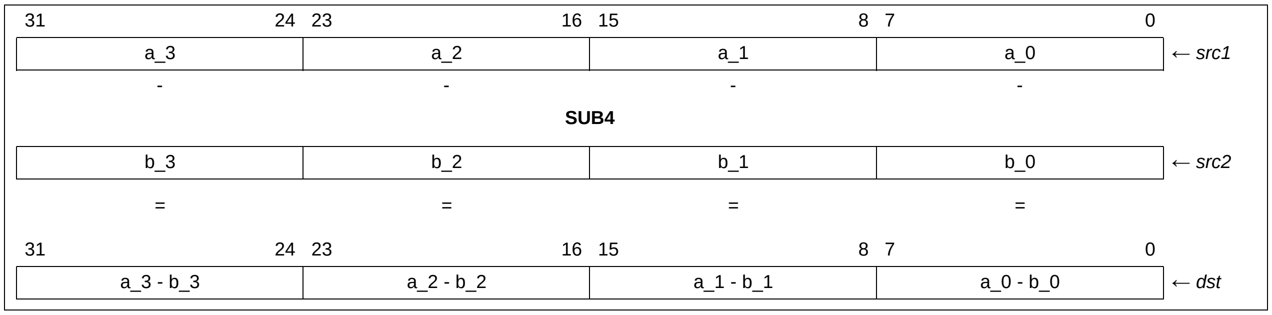
Opcode map field used...	For operand type...	Unit
src1	i4	.L1, .L2
src2	xi4	
dst	i4	

Description

Performs 2s-complement subtraction between packed 8-bit quantities. The values in *src1* and *src2* are treated as packed 8-bit data and the results are written into *dst* in a packed 8-bit format.

For each pair of 8-bit values in *src1* and *src2*, the difference between the 8-bit value from *src1* and the 8-bit value from *src2* is calculated to produce an 8-bit result. No saturation is performed. The result is placed in the corresponding position in *dst*:

- The difference between *src1* byte0 and *src2* byte0 is placed in byte0 of *dst*.
- The difference between *src1* byte1 and *src2* byte1 is placed in byte1 of *dst*.
- The difference between *src1* byte2 and *src2* byte2 is placed in byte2 of *dst*.
- The difference between *src1* byte3 and *src2* byte3 is placed in byte3 of *dst*.



Execution

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

Pipeline

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

Instruction Type Single-cycle

Delay Slots 0

See Also ADD4, SUB, SUB2

Example `SUB4 .L1 A2, A8, A9`

	Before instruction			1 cycle after instruction	
A2	37 89 F2 3Ah	55 137 242 58	A2	37 89 F2 3Ah	
A8	04 B8 49 75h	04 184 73 117	A8	04 B8 49 75h	
A9	xxxx xxxxh		A9	33 D1 A9 C5h	51 -47 169 -59

SWAP2　　*Swap Bytes in Upper and Lower Register Halves*

Syntax　　**SWAP2** (.unit) *src2, dst*

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

Compatibility　　C64x and C64x+ CPU

Opcode　　.L unit

31 29	28	27 23	22 18	17 13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	0	0	1	1	0	1	1	1	1	0	*s*	*p*
3	1	5	5	5	1											1	1

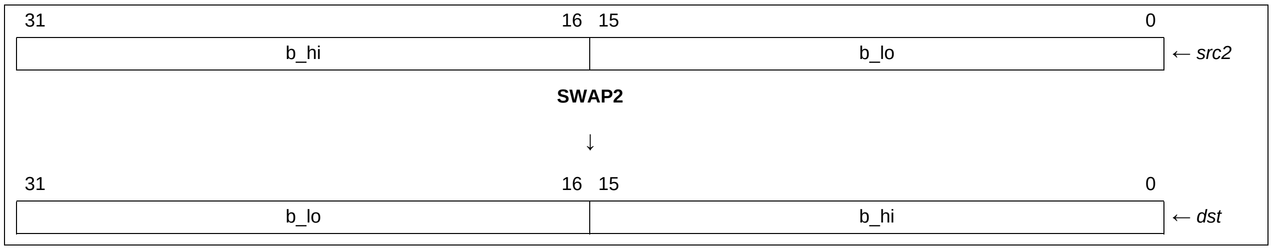
Opcode map field used...	For operand type...	Unit
src2 *dst*	s2 s2	.L1, .L2

Opcode　　.S unit

31 29	28	27 23	22 18	17 13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	*src1*	x	0	1	0	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
src2 *dst*	s2 s2	.S1, .S2

Description　　The **SWAP2** pseudo-operation takes the lower halfword from *src2* and places it in the upper halfword of *dst*, while the upper halfword from *src2* is placed in the lower halfword of *dst*. The assembler uses the **PACKLH2** (.unit) *src1, src2, dst* instruction to perform this operation (see PACKLH2).



The **SWAP2** instruction can be used in conjunction with the **SWAP4** instruction (see SWAP4) to change the byte ordering (and therefore, the endianess) of 32-bit data.

Execution

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

Pipeline

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

Instruction Type Single-cycle

Delay Slots 0

See Also SWAP4

Examples

Example 1

```
SWAP2 .L1 A2,A9
```

	Before instruction			1 cycle after instruction	
A2	3789 F23Ah	14217 -3526	A2	3789 F23Ah	
A9	xxxx xxxxh		A9	F23A 3789h	-3526 14217

Example 2

```
SWAP2 .S2 B2,B12
```

	Before instruction			1 cycle after instruction	
B2	0124 2451h	292 9297	B2	0124 2451h	
B12	xxxx xxxxh		B12	2451 0124h	9297 292

SWAP4 *Swap Byte Pairs in Upper and Lower Register Halves*

Syntax **SWAP4** (.unit) *src2, dst*

unit = .L1 or .L2

Compatibility C64x and C64x+ CPU

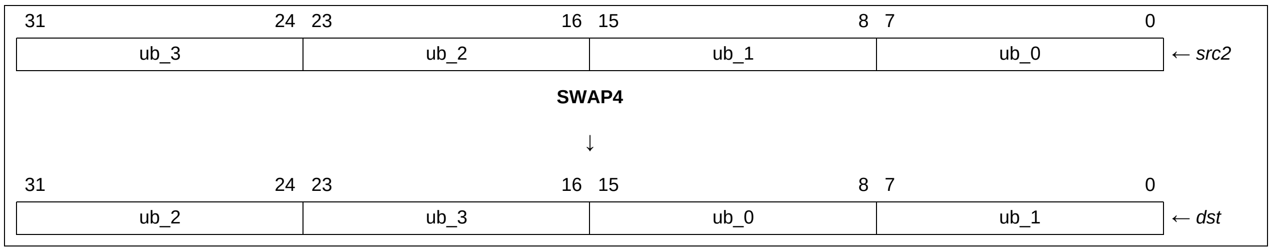
Opcode

31 29	28	27 23	22 18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src*	0	0	0	0	1	x	0	0	1	1	0	1	0	1	1	0	*s*	*p*
3	1	5	5						1											1	1

Opcode map field used...	For operand type...	Unit
*src*2 *dst*	xu4 u4	.L1, .L2

Description Exchanges pairs of bytes within each halfword of *src2*, placing the result in *dst*. The values in *src2* are treated as unsigned, packed 8-bit values.

Specifically the upper byte in the upper halfword is placed in the lower byte in the upper halfword, while the lower byte of the upper halfword is placed in the upper byte of the upper halfword. Also the upper byte in the lower halfword is placed in the lower byte of the lower halfword, while the lower byte in the lower halfword is placed in the upper byte of the lower halfword.



By itself, this instruction changes the ordering of bytes within halfwords. This effectively changes the endianess of 16-bit data packed in 32-bit words. The endianess of full 32-bit quantities can be changed by using the **SWAP4** instruction in conjunction with the **SWAP2** instruction (see SWAP2).

Execution

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

Pipeline

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

Instruction Type Single-cycle

Delay Slots 0

See Also SWAP2

Example `SWAP4 .L1 A1,A2`

	Before instruction			1 cycle after instruction	
A1	9E 52 6E 30h	158 82 110 48	A1	9E 52 6E 30h	
A2	xxxx xxxxh		A2	52 9E 30 6Eh	82 158 48 110

SWE *Software Exception*

Syntax **SWE**

unit = none

Compatibility C64x+ CPU

Opcode

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	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	*p*
																															1

Description Causes an internal exception to be taken. It can be used as a mechanism for User mode programs to request Supervisor mode services. Execution of the **SWE** instruction results in an exception being recognized in the E1 pipeline phase containing the **SWE** instruction. The SXF bit in EFR is set to 1. The HWE bit in NTSR is cleared to 0. If exceptions have been globally enabled, this causes an exception to be recognized before execution of the next execute packet. The address of that next execute packet is placed in NRP.

Execution

$1 \rightarrow$ SXF bit in EFR
$0 \rightarrow$ HWE bit in TSR

Instruction Type Single-cycle

Delay Slots 0

See Also SWENR

SWENR *Software Exception—No Return*

Syntax **SWENR**

unit = none

Compatibility C64x+ CPU

Opcode

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	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	*p*
																															1

Description Causes an internal exception to be taken. It is intended for use in systems supporting a secure operating mode. It can be used as a mechanism for User mode programs to request Supervisor mode services. It differs from the **SWE** instruction in four ways:

1. TSR is not copied into NTSR.
2. No return address is placed in NRP (it remains unmodified).
3. The IB bit in TSR is set to 1. This will be observable only in the case where another exception is recognized simultaneously.
4. A branch to REP (restricted entry point register) is forced in the context switch rather than the ISTP-based exception (NMI) vector.

This instruction executes unconditionally.

If another exception (internal or external) is recognized simultaneously with the SWENR-raised exception then the other exception(s) takes priority and normal exception behavior occurs; that is, NTSR and NRP are used and execution is directed to the NMI vector.

Execution

1 → SXF bit in EFR
0 → HWE bit in TSR

Instruction Type Single-cycle

Delay Slots 0

See Also SWE

UNPKHU4 *Unpack 16 MSB Into Two Lower 8-Bit Halfwords of Upper and Lower Register Halves*

Syntax **UNPKHU4** (.unit) *src2, dst*

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

Compatibility C64x and C64x+ CPU

Opcode .L unit

31 29	28	27 23	22 18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	0	0	0	1	1	x	0	0	1	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*	xu4 u2	.L1, .L2

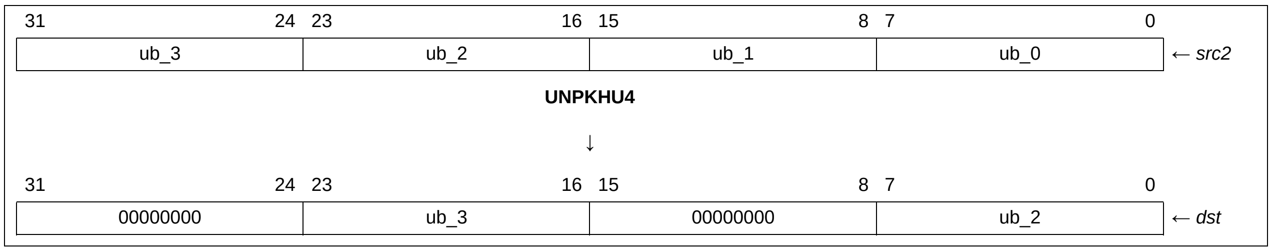
Opcode .S unit

31 29	28	27 23	22 18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	0	0	0	1	1	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*	xu4 u2	.S1, .S2

Description Moves the two most-significant bytes of *src2* into the two low bytes of the two halfwords of *dst*.

Specifically the upper byte in the upper halfword is placed in the lower byte in the upper halfword, while the lower byte of the upper halfword is placed in the lower byte of the lower halfword. The *src2* bytes are zero-extended when unpacked, filling the two high bytes of the two halfwords of *dst* with zeros.



Execution

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

Pipeline

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

Instruction Type Single cycle

Delay Slots 0

See Also UNPKLU4

Examples

Example 1

```
UNPKHU4 .L1 A1,A2
```

	Before instruction		1 cycle after instruction
A1	9E 52 6E 30h	A1	9E 52 6E 30h
A2	xxxx xxxxh	A2	00 9E 00 52h

Example 2

```
UNPKHU4 .L2 B17,B18
```

	Before instruction		1 cycle after instruction
B17	11 05 69 34h	B17	11 05 69 34h
B18	xxxx xxxxh	B18	00 11 00 05h

UNPKLU4 *Unpack 16 LSB Into Two Lower 8-Bit Halfwords of Upper and Lower Register Halves*

Syntax **UNPKLU4** (.unit) *src2, dst*

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

Compatibility C64x and C64x+ CPU

Opcode .L unit

31 29	28	27 23	22 18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	0	0	0	1	0	x	0	0	1	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*	xu4 u2	.L1, .L2

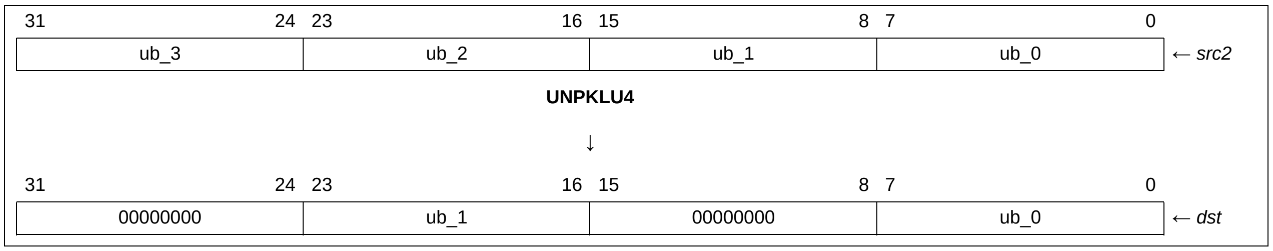
Opcode .S unit

31 29	28	27 23	22 18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	0	0	0	1	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*	xu4 u2	.S1, .S2

Description Moves the two least-significant bytes of *src2* into the two low bytes of the two halfwords of *dst*.

Specifically, the upper byte in the lower halfword is placed in the lower byte in the upper halfword, while the lower byte of the lower halfword is kept in the lower byte of the lower halfword. The *src2* bytes are zero-extended when unpacked, filling the two high bytes of the two halfwords of *dst* with zeros.



Execution

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

Pipeline

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

Instruction Type Single cycle

Delay Slots 0

See Also UNPKHU4

Examples

Example 1

```
UNPKLU4 .L1 A1,A2
```

	Before instruction		1 cycle after instruction
A1	9E 52 6E 30h	A1	9E 52 6E 30h
A2	xxxx xxxxh	A2	00 6E 00 30h

Example 2

```
UNPKLU4 .L2 B17,B18
```

	Before instruction		1 cycle after instruction
B17	11 05 69 34h	B17	11 05 69 34h
B18	xxxx xxxxh	B18	00 69 00 34h

XOR *Bitwise Exclusive OR*

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

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

Compatibility C62x, C64x, and C64x+ CPU

Compact Instruction Format

Unit	Opcode Format	Figure
.L	L2c	Figure D-7
.L, .S, .D	LSDx1	Figure G-4

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

Opcode .D unit

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

Opcode map field used...	For operand type...	Unit	Opfield
src1 *src2* *dst*	uint xuint uint	.D1, .D2	1110
src1 *src2* *dst*	scst5 xuint uint	.D1, .D2	1111

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, .S, or .D

Instruction Type Single-cycle

Delay Slots 0

See Also AND, ANDN, NOT, OR

Examples **Example 1**

```
XOR .S1 A3, A4, A5
```

	Before instruction		1 cycle after instruction
A3	0721 325Ah	A3	0721 325Ah
A4	0019 0F12h	A4	0019 0F12h
A5	xxxx xxxxh	A5	0738 3D48h

Example 2

```
XOR .D2 B1, 0Dh, B8
```

	Before instruction		1 cycle after instruction
B1	0000 1023h	B1	0000 1023h
B8	xxxx xxxxh	B8	0000 102Eh

XORMPY *Galois Field Multiply With Zero Polynomial*

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

unit = .M1 or .M2

Compatibility C64x+ CPU

Opcode

31	30	29	28	27–23	22–18	17–13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	*dst*	*src2*	*src1*	x	0	1	1	0	1	1	1	1	0	0	*s*	*p*
				5	5	5	1											1	1

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

Description Performs a Galois field multiply, where *src1* is 32 bits and *src2* is limited to 9 bits. This multiply connects all levels of the gmpy4 together and only extends out by 8 bits. The **XORMPY** instruction is identical to a **GMPY** instruction executed with a zero-value polynomial.

```
uword xormpy(uword src1,uword src2)
{
  // the multiply is always between GF(2^9) and GF(2^32)
  // so no size information is needed

  uint pp;
  uint mask, tpp;
  uint I;

      pp = 0;
      mask = 0x00000100; // multiply by computing
                         // partial products.
      for ( I=0; i<8; I++ ){
        if ( src2 & mask )   pp ^= src1;
        mask >>= 1;
        pp <<= 1;
      }
if ( src2 & 0x1 ) pp ^= src1;

return (pp) ;   // leave it asserted left.
}
```

Execution

GMPY_poly = 0
(lsb9(*src2*) gmpy uint(*src1*)) → uint(*dst*)

Instruction Type Four-cycle

Delay Slots 3

See Also GMPY, GMPY4, XOR

Example `XORMPY .M1 A0,A1,A2 GPLYA = FFFFFFFF (ignored)`

	Before instruction		1 cycle after instruction
A0	1234 5678h	A2	1E65 4210h
A1	0000 0126h		

XPND2 *Expand Bits to Packed 16-Bit Masks*

Syntax **XPND2** (.unit) *src2, dst*

unit = .M1 or .M2

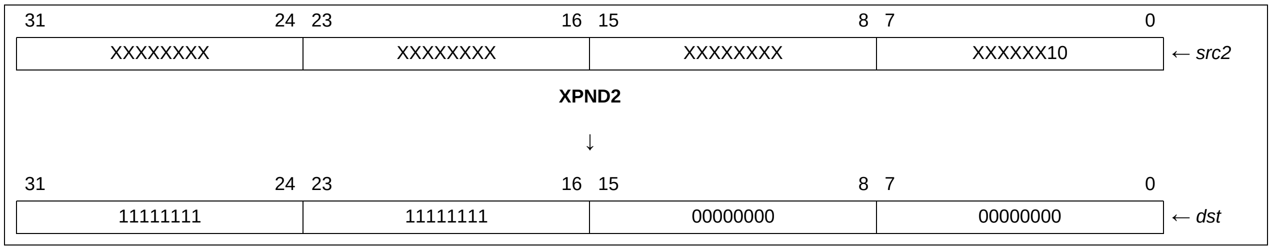
Compatibility C64x and C64x+ CPU

Opcode

31–29	28	27–23	22–18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	1	1	0	0	1	x	0	0	0	0	1	1	1	1	0	0	*s*	*p*
3	1	5	5						1											1	1

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

Description Reads the two least-significant bits of *src2* and expands them into two halfword masks written to *dst*. Bit 1 of *src2* is replicated and placed in the upper halfword of *dst*. Bit 0 of *src2* is replicated and placed in the lower halfword of *dst*. Bits 2 through 31 of *src2* are ignored.



The **XPND2** instruction is useful, when combined with the output of the **CMPGT2** or **CMPEQ2** instruction, for generating a mask that corresponds to the individual halfword positions that were compared. That mask may then be used with **ANDN**, **AND**, or **OR** instructions to perform other operations like compositing. This is an example:

```
CMPGT2   .S1   A3, A4, A5          ; Compare two registers, both upper
                                   ; and lower halves.
XPND2    .M1   A5, A2              ; Expand the compare results into
                                   ; two 16-bit masks.
NOP
AND      .D1   A2, A7, A8          ; Apply the mask to a value to create result.
```

Because the **XPND2** instruction only examines the two least-significant bits of *src2*, it is possible to store a large bit mask in a single 32-bit word and expand it using multiple **SHR** and **XPND2** instruction pairs. This can be useful for expanding a packed 1-bit-per-pixel bitmap into full 16-bit pixels in imaging applications.

Execution

```
if (cond)    {
             XPND2(src2 & 1) → lsb16(dst);
             XPND2(src2 & 2) → msb16(dst)
             }
else nop
```

Pipeline

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

Instruction Type Two-cycle

Delay Slots 1

See Also CMPEQ2, CMPGT2, XPND4

Examples

Example 1

```
XPND2 .M1 A1,A2
```

	Before instruction			2 cycles after instruction
A1	B174 6CA1h	2 LSBs are 01	A1	B174 6CA1h
A2	xxxx xxxxh		A2	0000 FFFFh

Example 2

```
XPND2 .M2 B1,B2
```

	Before instruction			2 cycles after instruction
B1	0000 0003h	2 LSBs are 11	B1	0000 0003h
B2	xxxx xxxxh		B2	FFFF FFFFh

XPND4 *Expand Bits to Packed 8-Bit Masks*

Syntax **XPND4** (.unit) *src2, dst*

unit = .M1 or .M2

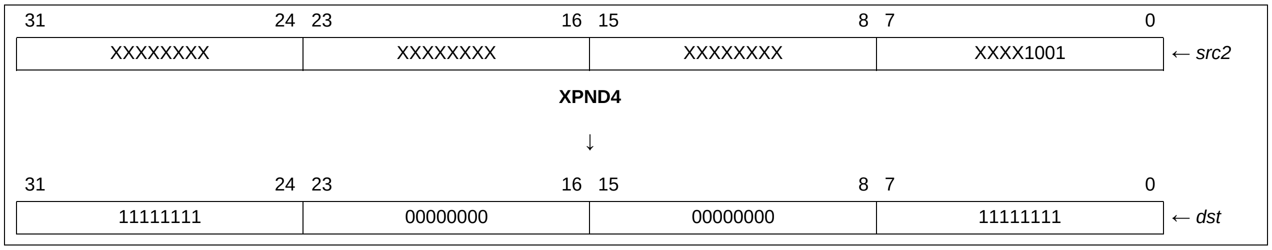
Compatibility C64x and C64x+ CPU

Opcode

31 29	28	27 23	22 18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src2*	1	1	0	0	0	x	0	0	0	0	1	1	1	1	0	0	*s*	*p*
3	1	5	5						1											1	1

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

Description Reads the four least-significant bits of *src2* and expands them into four-byte masks written to *dst*. Bit 0 of *src2* is replicated and placed in the least-significant byte of *dst*. Bit 1 of *src2* is replicated and placed in second least-significant byte of *dst*. Bit 2 of *src2* is replicated and placed in second most-significant byte of *dst*. Bit 3 of *src2* is replicated and placed in most-significant byte of *dst*. Bits 4 through 31 of *src2* are ignored.



The **XPND4** instruction is useful, when combined with the output of the **CMPGT4** or **CMPEQ4** instruction, for generating a mask that corresponds to the individual byte positions that were compared. That mask may then be used with **ANDN**, **AND**, or **OR** instructions to perform other operations like compositing.

This is an example:

```
CMPEQ4   .S1   A3, A4, A5        ; Compare two 32-bit registers all four bytes.
XPND4    .M1   A5, A2            ; Expand the compare results into
                                 ; four 8-bit masks.
NOP
AND      .D1   A2, A7, A8        ; Apply the mask to a value to create result.
```

Because the **XPND4** instruction only examines the four least-significant bits of *src2*, it is possible to store a large bit mask in a single 32-bit word and expand it using multiple **SHR** and **XPND4** instruction pairs. This can be useful for expanding a packed, 1-bit-per-pixel bitmap into full 8-bit pixels in imaging applications.

Execution

```
if (cond)    {
             XPND4(src2 & 1) → byte0(dst);
             XPND4(src2 & 2) → byte1(dst):
             XPND4(src2 & 4) → byte2(dst);
             XPND4(src2 & 8) → byte3(dst)
             }
else nop
```

Pipeline

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

Instruction Type Two-cycle

Delay Slots 1

See Also CMPEQ4, CMPGTU4, XPND2

Examples

Example 1

```
XPND4 .M1 A1,A2
```

	Before instruction			2 cycles after instruction
A1	B174 6CA4h	4 LSBs are 0100	A1	B174 6CA4h
A2	xxxx xxxxh		A2	00 FF 00 00h

Example 2

```
XPND4 .M2 B1,B2
```

	Before instruction			2 cycles after instruction
B1	0000 000Ah	4 LSBs are 1010	B1	00 00 00 0Ah
B2	xxxx xxxxh		B2	FF 00 FF 00h

ZERO *Zero a Register*

Syntax

ZERO (.unit) *dst*

or

ZERO (.unit) *dst_o:dst_e*

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

Compatibility C62x, C64x, and C64x+ 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

This is a pseudo-operation used to fill the destination register or register pair with 0s.

When the destination is a single register, the assembler uses the **MVK** instruction to load it with zeros: **MVK** (.unit) 0, *dst* (see MVK).

When the destination is a register pair, the assembler uses the **SUB** instruction (see SUB) to subtract a value from itself and store the result in the destination pair.

Execution

```
if (cond)     0 → dst
else nop
```

or

```
if (cond)     src − src → dst_o:dst_e
else nop
```

Instruction Type Single-cycle

Delay Slots 0

See Also MVK, SUB

Examples

Example 1

```
ZERO .D1 A1
```

	Before instruction		1 cycle after instruction
A1	B174 6CA1h	A1	0000 0000h

Example 2

```
ZERO .L1 A1:A0
```

	Before instruction		1 cycle after instruction
A0	B174 6CA1h	A0	0000 0000h
A1	1234 5678h	A1	0000 0000h

Pipeline

The C64x/C64x+ DSP pipeline provides flexibility to simplify programming and improve performance. These two factors provide this flexibility:

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

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

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

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

Finally, this 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**

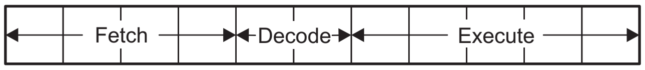
4.1 Pipeline Operation Overview

The pipeline phases are divided into three stages:

- Fetch
- Decode
- Execute

All instructions in the C64x/C64x+ 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 C64x/C64x+ DSP pipeline are shown in Figure 4-1.

Figure 4-1. Pipeline Stages



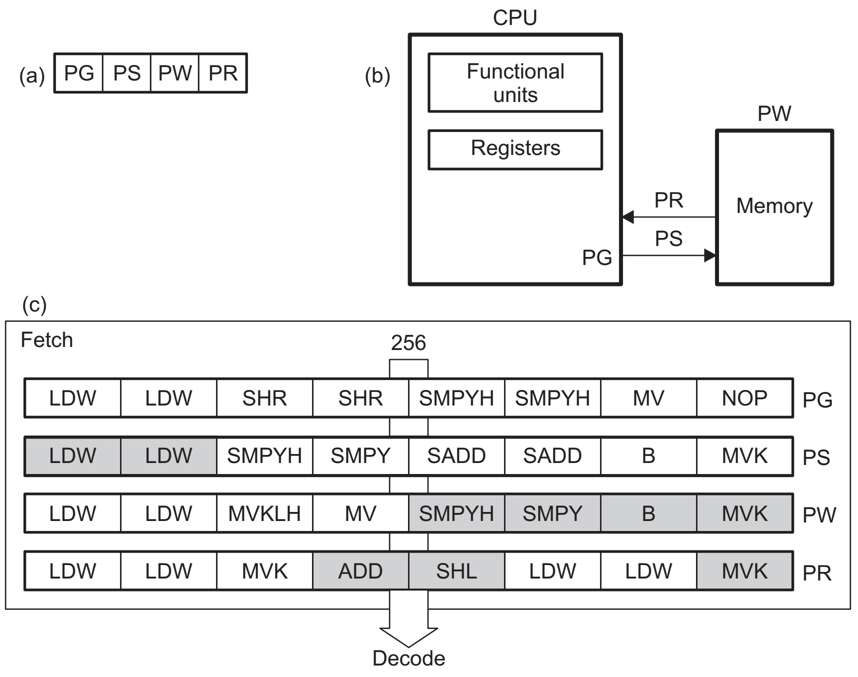
4.1.1 Fetch

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 C64x/C64x+ DSP uses a fetch packet (FP) of eight words. All eight of the words 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 instructions.

Figure 4-2. Fetch Phases of the Pipeline


(a)
PG
PS
PW
PR
(b)
CPU
Functional units
Registers
PG
PR
PS
PW
Memory
(c)
Fetch
256
LDW LDW SHR SHR SMPYH SMPYH MV NOP
PG
LDW LDW SMPYH SMPY SADD SADD B MVK
PS
LDW LDW MVKLH MV SMPYH SMPY B MVK
PW
LDW LDW MVK ADD SHL LDW LDW MVK
PR
Decode

4.1.2 Decode

The decode phases of the pipeline are:

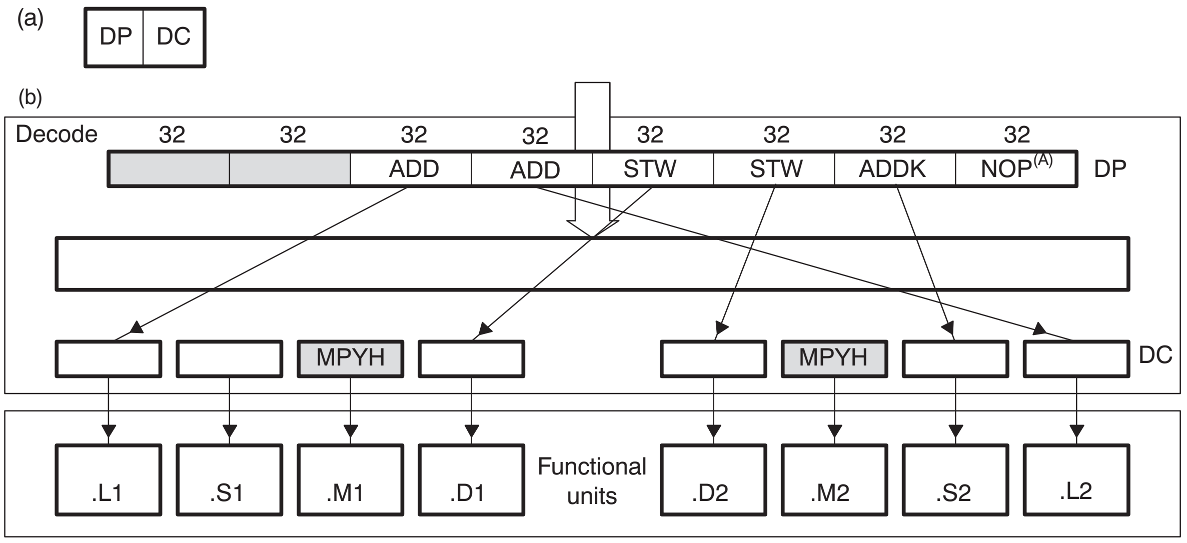
- **DP:** Instruction dispatch
- **DC:** Instruction decode

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 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

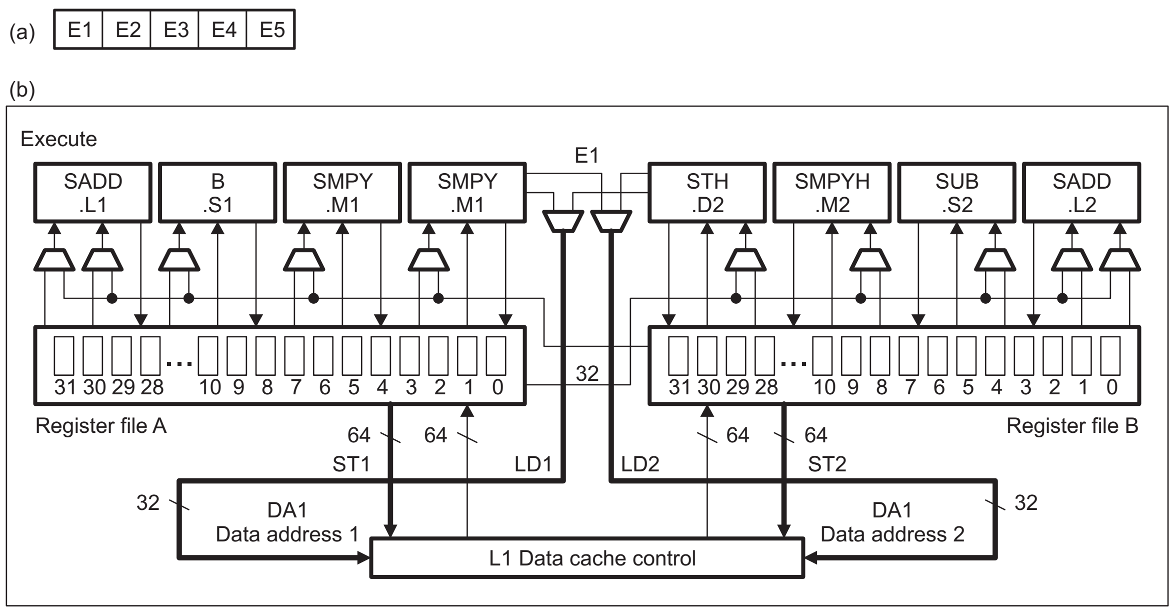


A NOP is not dispatched to a functional unit.

4.1.3 Execute

The execute portion of the pipeline is subdivided into five phases (E1-E5). Different types of instructions require different numbers of these phases to complete their execution. These phases of the pipeline play an important role in your understanding the device state at CPU cycle boundaries. The execution of different types of instructions in the pipeline is described in Section 4.2. 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



4.1.4 *Pipeline Operation Summary*

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

Figure 4-5. Pipeline Phases

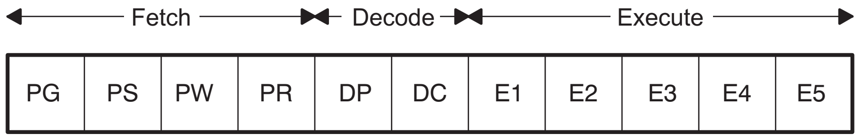


Figure 4-6 shows an example of the pipeline flow of consecutive fetch packets that contain eight parallel instructions. In this case, where the pipeline is full, all instructions in a fetch packet are in parallel and split into one execute packet per fetch packet. The fetch packets flow in lockstep fashion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

For example, examine cycle 7 in Figure 4-6. When the instructions from FPn reach E1, the instructions in the execute packet from FP n +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 4.3 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**
n	PG	PS	PW	PR	DP	DC	E1	E2	E3	E4	E5		
n+1		PG	PS	PW	PR	DP	DC	E1	E2	E3	E4	E5	
n+2			PG	PS	PW	PR	DP	DC	E1	E2	E3	E4	E5
n+3				PG	PS	PW	PR	DP	DC	E1	E2	E3	E4
n+4					PG	PS	PW	PR	DP	DC	E1	E2	E3
n+5						PG	PS	PW	PR	DP	DC	E1	E2
n+6							PG	PS	PW	PR	DP	DC	E1
n+7								PG	PS	PW	PR	DP	DC
n+8									PG	PS	PW	PR	DP
n+9										PG	PS	PW	PR
n+10											PG	PS	PW

Table 4-1. Operations Occurring During Pipeline Phases

Stage	Phase	Symbol	During This Phase
Program fetch	Program address generate	PG	The address of the fetch packet is determined.
	Program address send	PS	The address of the fetch packet is sent to memory.
	Program wait	PW	A program memory access is performed.
	Program data receive	PR	The fetch packet is at the CPU boundary.
Program decode	Dispatch	DP	The next execute packet in the fetch packet is determined and sent to the appropriate functional units to be decoded.
	Decode	DC	Instructions are decoded in functional units.
Execute	Execute 1	E1	For all instruction types, the conditions for the instructions are evaluated and operands are read.
			For load and store instructions, address generation is performed and address modifications are written to a register file.(1)
			For branch instructions, branch fetch packet in PG phase is affected.(1)
			For single-cycle instructions, results are written to a register file.(1)
	Execute 2	E2	For load instructions, the address is sent to memory. For store instructions, the address and data are sent to memory.(1)
			Single-cycle instructions that saturate results set the SAT bit in the control status register (CSR) if saturation occurs.(1)
			For single 16 × 16 multiply instructions, results are written to a register file.(1)
			For multiply unit, nonmultiply instructions, results are written to a register file.(2)
	Execute 3	E3	Data memory accesses are performed. Any multiply instructions that saturate results set the SAT bit in the control status register (CSR) if saturation occurs.(1)
	Execute 4	E4	For load instructions, data is brought to the CPU boundary.(1)
			For multiply extensions, results are written to a register file.(3)
	Execute 5	E5	For load instructions, data is written into a register.(1)

(1) This assumes that the conditions for the instructions are evaluated as true. If the condition is evaluated as false, the instruction does not write any results or have any pipeline operation after E1.
(2) Multiply unit, nonmultiply instructions are **AVG2, AVGU4, BITC4, BITR, DEAL, ROT, SHFL, SSHVL,** and **SSHVR.**
(3) Multiply extensions include **MPY2, MPY4, DOTPx2, DOTPU4, MPYHIx, MPYLIx,** and **MVD.**

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.

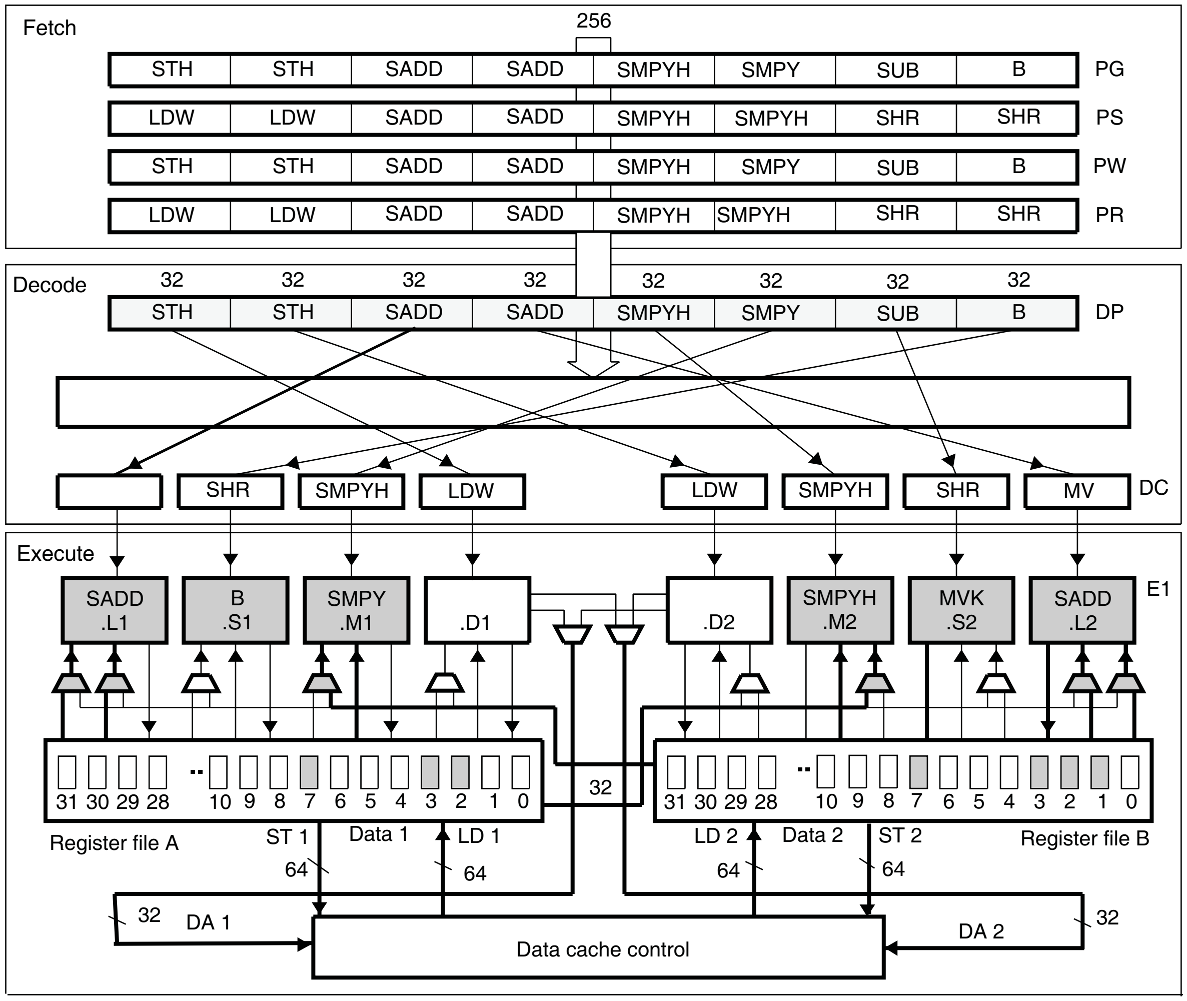
As code flows through the pipeline phases, it is processed by different parts of the DSP. Figure 4-7 shows a full pipeline with a fetch packet in every phase of fetch. One execute packet of eight instructions is being dispatched at the same time that a 7-instruction execute packet is in decode. The arrows 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 cross paths are used by the **MPY** instructions.

Most DSP instructions are single-cycle instructions, which means they have only one execution phase (E1). A small number of instructions require more than one execute phase. The types of instructions, each of which require different numbers of execute phases, are described in Section 4.2.

Figure 4-7. Pipeline Phases Block Diagram



Example 4-1. Execute Packet in Figure 4-7

```
        SADD    .L1     A2,A7,A2         ; E1 Phase
||      SADD    .L2     B2,B7,B2
||      SMPYH   .M2X    B3,A3,B2
||      SMPY    .M1X    B3,A3,A2
||      B       .S1     LOOP1
||      MVK     .S2     117,B1

        LDW     .D2     *B4++,B3         ; DC Phase
||      LDW     .D1     *A4++,A3
||      MV      .L2X    A1,B0
||      SMPYH   .M1     A2,A2,A0
||      SMPYH   .M2     B2,B2,B10
||      SHR     .S1     A2,16,A5
||      SHR     .S2     B2,16,B5

LOOP1:

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

        LDW     .D2     *B4++,B3         ; PR and PS Phases
||      LDW     .D1     *A4++,A3
||      SADD    .L1     A0,A1,A1
||      SADD    .L2     B10,B0,B0
||      SMPYH   .M1     A2,A2,A0
||      SMPYH   .M2     B2,B2,B10
||      SHR     .S1     A2,16,A5
||      SHR     .S2     B2,16,B5
```

4.2 Pipeline Execution of Instruction Types

The pipeline operation of the C64x/C64x+ DSP instructions can be categorized into seven instruction types. Six of these (**NOP** is not included) are shown in Table 4-2, which is a mapping of operations occurring in each execution phase for the different instruction types. The delay slots associated with each instruction type are also listed.

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

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

	Instruction Type					
Execution Phase [1] [2]	**Single Cycle**	**16 × 16 Single Multiply/.M Unit Nonmultiply**	**Store**	**C64x Multiply Extensions**	**Load**	**Branch**
E1	Compute result and write to register	Read operands and start computations	Compute address	Reads operands and start computations	Compute address	Target code in PG [3]
E2		Compute result and write to register	Send address and data to memory		Send address to memory	
E3			Access memory		Access memory	
E4				Write results to register	Send data back to CPU	
E5					Write data into register	
Delay slots	0	1	0 [4]	3	4 [4]	5 [3]
Functional unit latency	1	1	1	1	1	1

(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.
(3) See Section 4.2.6 for more information on branches.
(4) See Section 4.2.3 and Section 4.2.5 for more information on execution and delay slots for stores and loads.

4.2.1 Single-Cycle Instructions

Single-cycle instructions complete execution during the E1 phase of the pipeline (Table 4-3). Figure 4-8 shows the fetch, decode, and execute phases of the pipeline that the 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.

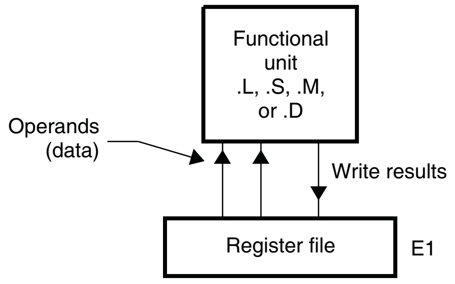
Table 4-3. Single-Cycle Instruction Execution

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 *Two-Cycle Instructions and .M Unit Nonmultiply Operations*

Two-cycle or multiply instructions use both the E1 and E2 phases of the pipeline to complete their operations (Table 4-4). Figure 4-10 shows the fetch, decode, and execute phases of the pipeline that the two-cycle instructions use.

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

Table 4-4. Multiply Instruction Execution

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

Figure 4-10. Two-Cycle Instruction Phases

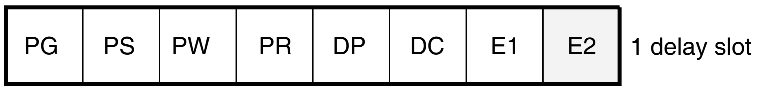
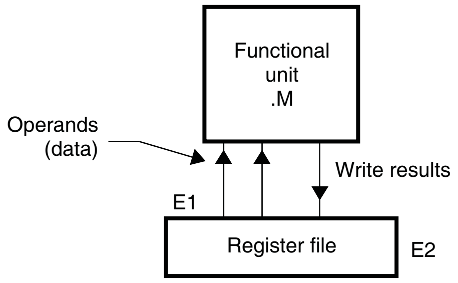


Figure 4-11. Single 16 × 16 Multiply Instruction Execution Block Diagram



4.2.3 Store Instructions

Store instructions require phases E1 through E3 of the pipeline to complete their operations (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 4.2.5.

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

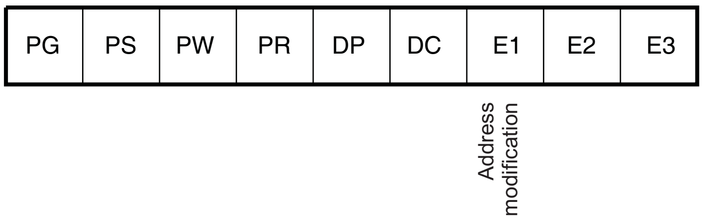
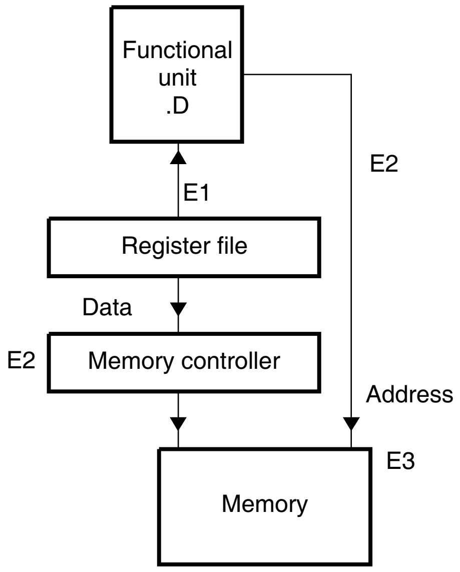


Figure 4-13. Store Instruction Execution Block Diagram



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 Extended Multiply Instructions

The extended multiply instructions use phases E1 through E4 to complete their operations (Table 4-6). Figure 4-14 shows the fetch, decode, and execute phases of the pipeline that the extended multiply instructions.

Figure 4-15 shows the operations occurring in the pipeline for the multiply extensions. In the E1 phase, the operands are read and the multiplies begin. In the E4 phase, the multiplies finish, and the results are written to the destination register. Extended multiply instructions have three delay slots.

Table 4-6. Extended Multiply Instruction Execution

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

Figure 4-14. Extended Multiply Instruction Phases

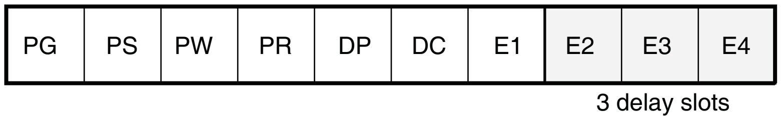
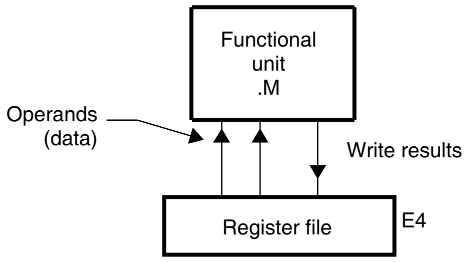


Figure 4-15. Extended Multiply Instruction Execution Block Diagram



4.2.5 Load Instructions

Data loads require all five, E1 through E5, of the pipeline execute phases to complete their operations (Table 4-7). Figure 4-16 shows the fetch, decode, and execute phases of the pipeline that the load instructions use.

Figure 4-17 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 4-7. Load Instruction Execution

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

Figure 4-16. Load Instruction Phases

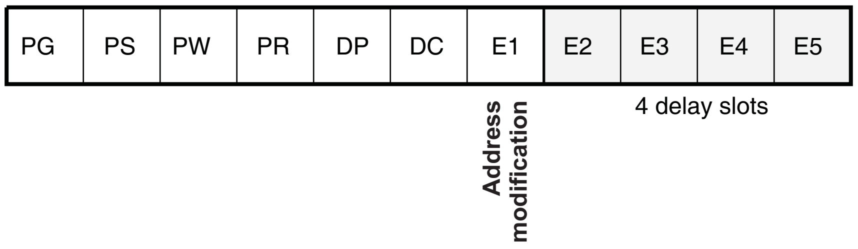
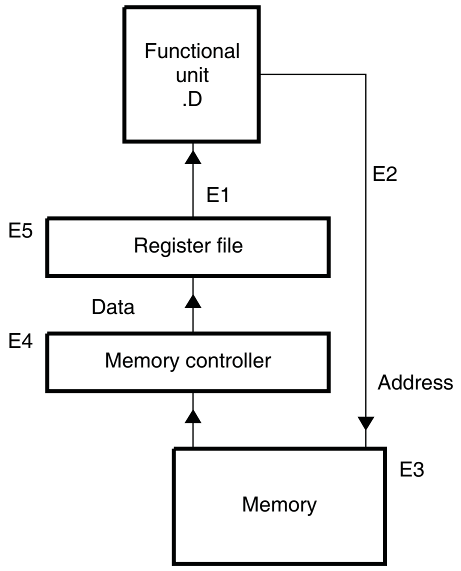


Figure 4-17. Load Instruction Execution Block Diagram



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.

4.2.6 Branch Instructions

Although branch instructions take one execute phase, there are five delay slots between the execution of the branch and execution of the target code (Table 4-8). Figure 4-18 shows the pipeline phases used by the branch instruction and branch target code. The delay slots are shaded.

Figure 4-19 shows a branch instruction execution block diagram. If a branch is in the E1 phase of the pipeline (in the .S2 unit in Figure 4-19), 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.

On the C64x+ DSP, a stall is inserted if a branch is taken to an execute packet that spans fetch packets to give time to fetch the second packet. Normally the assembler compensates for this by preventing branch targets from spanning fetch packets. The one case in which this cannot be done is in the case that an interrupt or exception occurred and the return target is a fetch packet spanning execute packet.

Table 4-8. Branch Instruction Execution

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

Figure 4-18. Branch Instruction Phases

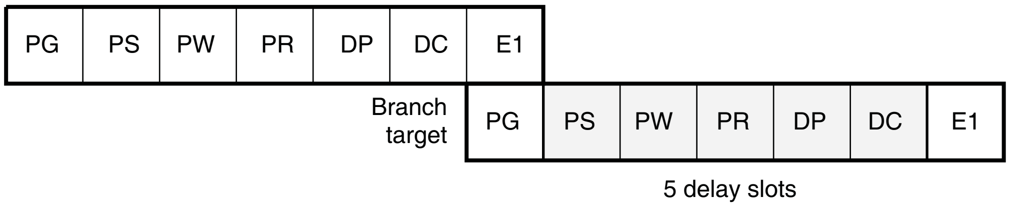
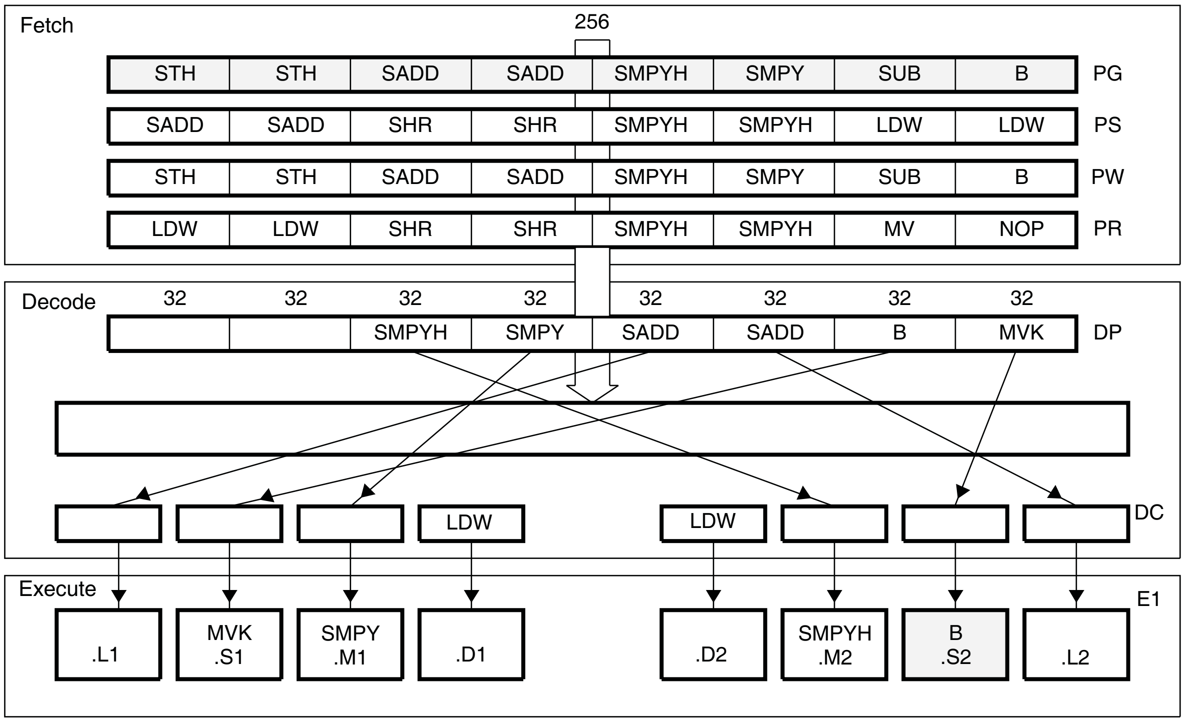


Figure 4-19. Branch Instruction Execution Block Diagram


Fetch
256
STH STH SADD SADD SMPYH SMPY SUB B PG
SADD SADD SHR SHR SMPYH SMPYH LDW LDW PS
STH STH SADD SADD SMPYH SMPY SUB B PW
LDW LDW SHR SHR SMPYH SMPYH MV NOP PR
Decode
32 32 32 32 32 32 32 32
SMPYH SMPY SADD SADD B MVK DP
LDW LDW DC
Execute
E1
.L1
MVK .S1
SMPY .M1
.D1
.D2
SMPYH .M2
B .S2
.L2

4.3 Performance Considerations

The C64x/C64x+ 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 4.3.2 covers the effect of including a multicycle **NOP** in an individual EP.

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

4.3.1 Pipeline Operation With Multiple Execute Packets in a Fetch Packet

Referring to Figure 4-6, pipeline operation is shown with eight instructions in every fetch packet. Figure 4-20, however, shows the pipeline operation with a fetch packet that contains multiple execute packets. Code for Figure 4-20 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 4-20. 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**
n	k	PG	PS	PW	PR	DP	DC	E1	E2	E3	E4	E5		
n	k+1						DP	DC	E1	E2	E3	E4	E5	
n	k+2							DP	DC	E1	E2	E3	E4	E5
n+1	k+3		PG	PS	PW	PR			DP	DC	E1	E2	E3	E4
n+2	k+4			PG	PS	PW	Pipeline		PR	DP	DC	E1	E2	E3
n+3	k+5				PG	PS	stall		PW	PR	DP	DC	E1	E2
n+4	k+6					PG			PS	PW	PR	DP	DC	E1
n+5	k+7								PG	PS	PW	PR	DP	DC
n+6	k+8									PG	PS	PW	PR	DP

In Figure 4-20, fetch packet n, which contains three execute packets, is shown followed by six fetch packets (n + 1 through n + 6), each with one execute packet (containing eight parallel instructions). The first fetch packet (n) goes through the program fetch phases during cycles 1-4. During these cycles, a program fetch phase is started for each of the fetch packets that follow.

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

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

4.3.2 Multicycle NOPs

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

Figure 4-21 shows how a multicycle **NOP** drives the execution of other instructions in the same execute packet. Figure 4-21(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 4-21(b) shows the replacement of the single-cycle **NOP** with a multicycle **NOP** (**NOP 5**) in the same execute packet. The **NOP5** 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 **NOP5** period has completed.

Figure 4-21. Multicycle NOP in an Execute Packet

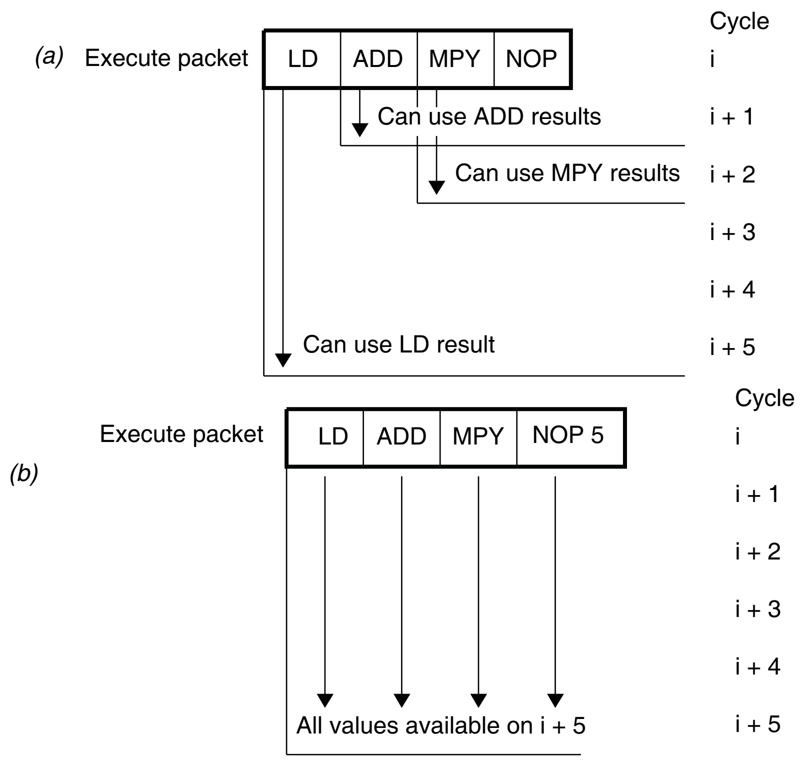
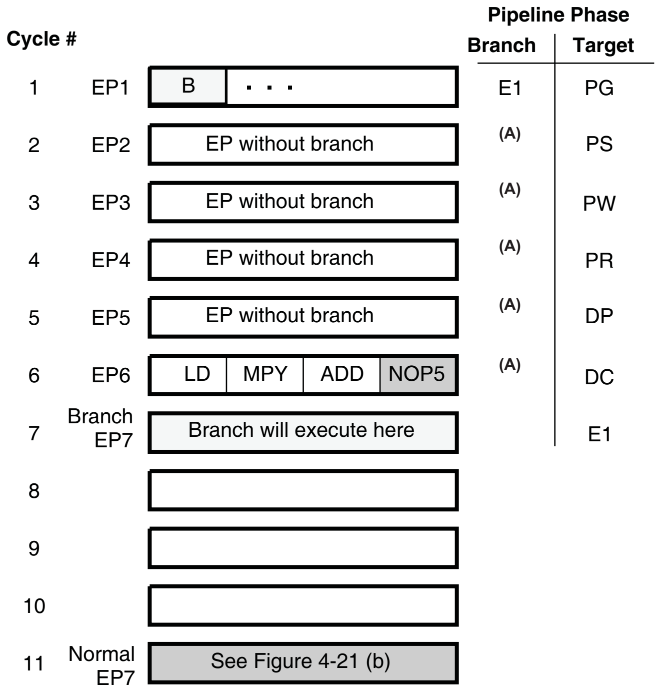


Figure 4-22 shows how a multicycle **NOP** can be affected by a branch. If the delay slots of a branch finish while a multicycle **NOP** is still dispatching **NOP**s into the pipeline, the branch overrides the multicycle **NOP** and the branch target begins execution five delay slots after the branch was issued.

Figure 4-22. Branching and Multicycle NOPs


Cycle #
Pipeline Phase
Branch
Target
1 EP1 B . . . E1 PG
2 EP2 EP without branch (A) PS
3 EP3 EP without branch (A) PW
4 EP4 EP without branch (A) PR
5 EP5 EP without branch (A) DP
6 EP6 LD MPY ADD NOP5 (A) DC
7 Branch EP7 Branch will execute here E1
8
9
10
11 Normal EP7 See Figure 4-21 (b)

A Delay slots of the branch

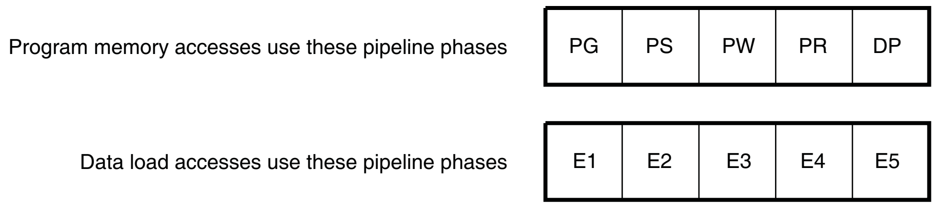
In one case, execute packet 1 (EP1) does not have a branch. The **NOP 5** in EP6 forces the CPU to wait until cycle 11 to execute EP7.

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.

4.3.3 Memory Considerations

The C64x/C64x+ 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 C64x/C64x+ DSP to access memory at a high speed. These phases are shown in Figure 4-23.

Figure 4-23. Pipeline Phases Used During Memory Accesses


Program memory accesses use these pipeline phases
PG PS PW PR DP
Data load accesses use these pipeline phases
E1 E2 E3 E4 E5

To understand the memory accesses, compare data loads and instruction fetches/dispatches. The comparison is valid because data loads and program fetches operate on internal memories of the same speed on the C64x/C64x+ DSP and perform the same types of operations (listed in Table 4-9) to accommodate those memories. Table 4-9 shows the operation of program fetches pipeline versus the operation of a data load.

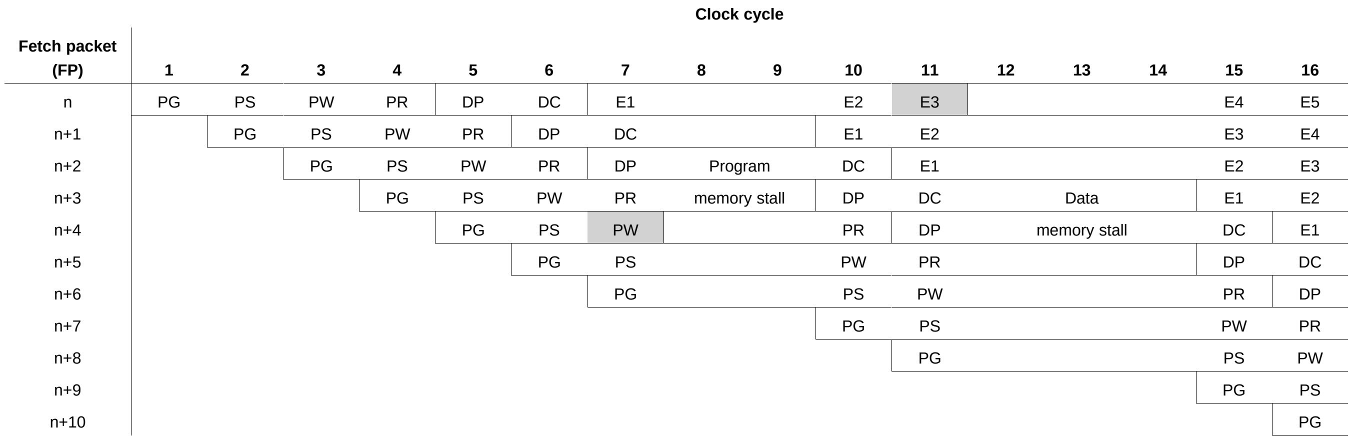
Table 4-9. Program Memory Accesses Versus Data Load Accesses

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.

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 4-24 illustrates this point.

Figure 4-24. 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 C64x+ DSP Differences

The pipeline in the C64x+ DSP has the following differences:

- On the C64x DSP, the fetch packet is 8 words which is the same as 8 instructions. On the C64x+ DSP, the fetch packet is still 8 words, but due to the possibility of compact instructions in a header-based packet, the fetch packet may contain as many as 14 instructions.
- There is an additional 2-cycle delay in processing interrupts on the C64x+ DSP as compared to the C64x DSP.
- On the C64x+ DSP, if a branch is taken to an execute packet that spans fetch packets, a stall is inserted to provide time to fetch the second fetch packet.
- On the C64x+ DSP, if an interrupt or exception is taken and the return target is a fetch packet spanning an execute packet, a stall is inserted to provide time to fetch the second fetch packet.

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, this 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 four types of interrupts on the CPU.

- Reset
- Maskable
- Nonmaskable
- Exception (C64x+ core only)

NOTE: The nonmaskable interrupt (NMI) is not supported on all C6000 devices, see your device-specific data manual for more information.

These first 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 (NMI) 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 datasheet to see your interrupt specifications.

The C64x+ CPU supports exceptions as another type of interrupt. When exceptions are enabled, the NMI input behaves as an exception. This chapter does not deal in depth with exceptions, as it assumes for discussion of NMI as an interrupt that they are disabled. Chapter 6 discusses exceptions including NMI behavior as an exception.

CAUTION

Code Compatibility

The C64x+ CPU code compatibility with existing code compiled for the C64x CPU using NMI as an interrupt is only assured when exceptions are not enabled. Any additional or modified code requiring the use of NMI as an exception to ensure correct behavior will likely require changes to the pre-existing code to adjust for the additional functionality added by enabling exceptions.

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 ($\overline{\text{RESET}}$)

Reset is the highest priority interrupt and is used to halt the CPU and return it to a known state. The reset interrupt is unique in a number of ways:

- $\overline{\text{RESET}}$ is an active-low signal. All other interrupts are active-high signals.
- $\overline{\text{RESET}}$ must be held low for 10 clock cycles before it goes high again to reinitialize the CPU properly.
- The instruction execution in progress is aborted and all registers are returned to their default states.
- The reset interrupt service fetch packet must be located at a specific address which is specific to the specific device. See the device datasheet for more information.
- $\overline{\text{RESET}}$ is not affected by branches.

5.1.1.2 Nonmaskable Interrupt (NMI)

NOTE: The nonmaskable interrupt (NMI) is not supported on all C6000 devices, see your device-specific data manual for more information.

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 (IER) 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.

On the C64x+ CPU, if an NMI is recognized within an SPLOOP operation, the behavior is the same as for an NMI with exceptions enabled. The SPLOOP operation terminates immediately (loop does not wind down as it does in case of an interrupt). The SPLX bit in the NMI/exception task state register (NTSR) is set for status purposes. The NMI service routine must look at this as one of the factors on whether a return to the interrupted code is possible. If the SPLX bit in NTSR is set, then a return to the interrupted code results in incorrect operation. See Section 7.13 for more information.

5.1.1.3 Maskable Interrupts (INT4-INT15)

The CPUs have 12 interrupts that are maskable. These have lower priority than the NMI and reset interrupts as well as all exceptions. 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:

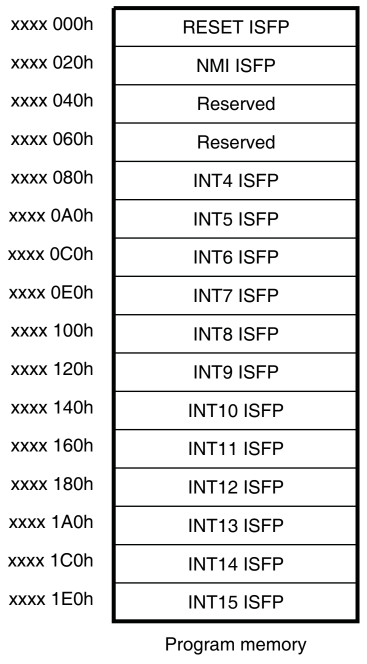
- The global interrupt enable bit (GIE) bit in the control status register (CSR) is set to1.
- The NMIE bit in the interrupt enable register (IER) is set to1.
- The corresponding interrupt enable (IE) bit in the IER is set to1.
- The corresponding interrupt occurs, which sets the corresponding bit in the interrupt flags register (IFR) to 1 and there are no higher priority interrupt flag (IF) bits set in the IFR.

5.1.2 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 up to 14 instructions (either 8 32-bit instructions in a nonheader-based fetch packet or up to 14 instructions in a compact header-based fetch packet). 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



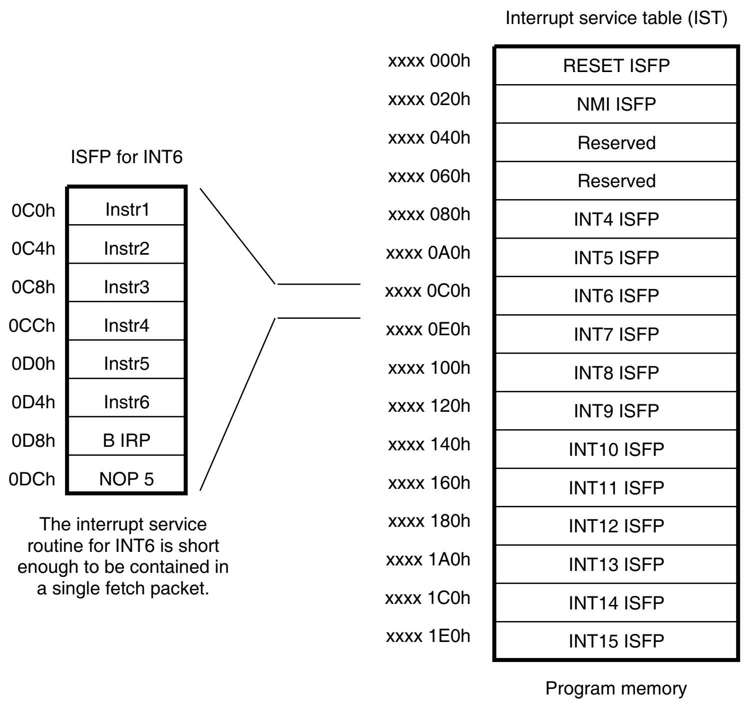
5.1.2.1 Interrupt Service Fetch Packet (ISFP)

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: The ISFP should be exactly 8 words long. To prevent the compiler from using compact instructions (see Section 3.9), the interrupt service table should be preceded by a .nocmp directive. See the *TMS320C6000 Assembly Language Tools User's Guide* (SPRU186).

If the **NOP 5** was not in the routine, the CPU would execute the next five execute packets (some of which are likely to be associated with the next ISFP) because of the delay slots associated with the **B IRP** instruction. See Section 4.2.6 for more information.

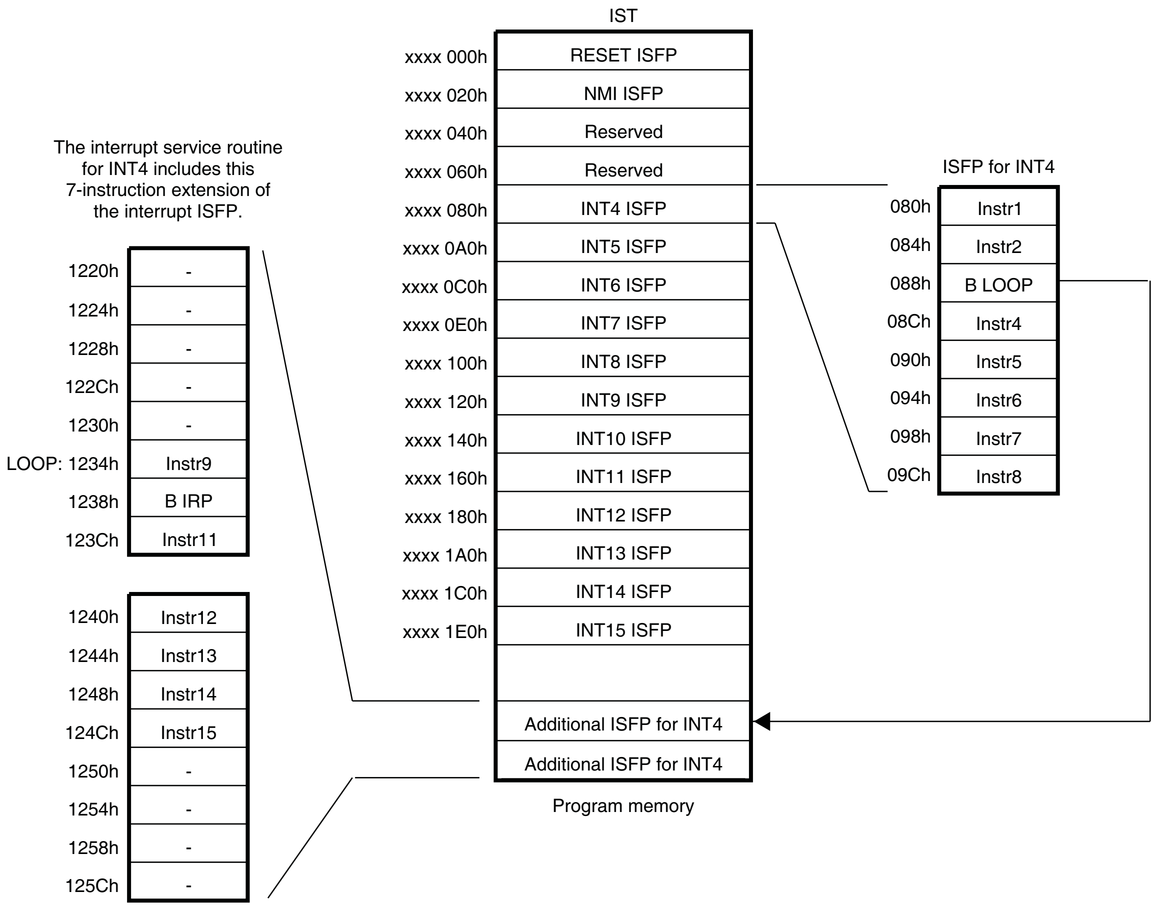
Figure 5-2. Interrupt Service Fetch Packet



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 the default location (see device data manual for more information), but the rest of the IST can be at any program memory location that is on a 256-word boundary (that is, any 1K byte 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 2-12 and described in Table 2-15. Example 5-1 shows the relationship of the ISTB to the table location.

Since the HPEINT field in ISTP gives the value of the highest priority interrupt that is both pending and enabled, the whole of ISTP gives the address of the highest priority interrupt that is both pending and enabled

Example 5-1. Relocation of Interrupt Service Table

(a) Relocating the IST to 800h

1) Copy 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

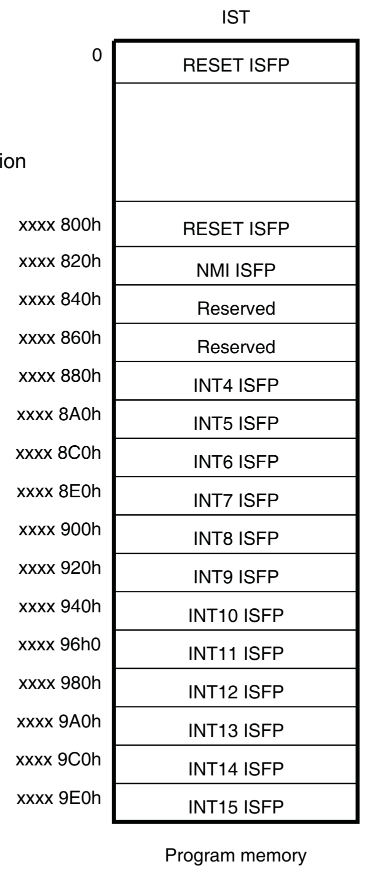
(b) How the ISTP directs the CPU to the appropriate ISFP in the relocated IST

Assume: IFR = BBC0h = 1011 1011 1100 0000b
IER = 1230h = 0001 0010 0011 0001b

2 enabled interrupts pending: INT9 and INT12

The 1s in IFR indicate pending interrupts; the 1s in 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 C64x/C64x+ CPU. The TSR, ITSR, and NTSR exist only on the C64x+ CPU.

Table 5-2. Interrupt Control Registers

Acronym	Register Name	Description	Section
CSR	Control status register	Allows you to globally set or disable interrupts	Section 2.8.4
ICR	Interrupt clear register	Allows you to clear flags in the IFR manually	Section 2.8.6
IER	Interrupt enable register	Allows you to enable interrupts	Section 2.8.7
IFR	Interrupt flag register	Shows the status of interrupts	Section 2.8.8
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.	Section 2.8.9
ISR	Interrupt set register	Allows you to set flags in the IFR manually	Section 2.8.10
ISTP	Interrupt service table pointer register	Pointer to the beginning of the interrupt service table	Section 2.8.11
ITSR	Interrupt task state register	Interrupted (non-NMI) machine state. (C64x+ CPU only)	Section 2.9.9
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.	Section 2.8.12
NTSR	Nonmaskable interrupt task state register	Interrupted (NMI) machine state. (C64x+ CPU only)	Section 2.9.10
TSR	Task state register	Allows you to globally set or disable interrupts. Contains status of current machine state. (C64x+ CPU only)	Section 2.9.15

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 and described in Table 2-9.

On the C64x+ CPU, there is one physical GIE bit that is mapped to bit 0 of both CSR and TSR. Similarly, there is one physical PGIE bit. It is mapped as CSR.PGIE (bit 1) and ITSR.GIE (bit 0). Modification to either of these bits is reflected in both of the mappings. In the following discussion, references to the GIE bit in CSR also refer to the GIE bit in TSR, and references to the PGIE bit in CSR also refer to the GIE bit in ITSR.

The global interrupt enable (GIE) allows you to enable or disable all maskable interrupts by controlling the value of a single bit. GIE is bit 0 of both the control status register (CSR) and the task state register (TSR).

- GIE = 1 enables the maskable interrupts so that they are processed.
- GIE = 0 disables the maskable interrupts so that they are not processed.

The CPU detects interrupts in parallel with instruction execution. As a result, the CPU may begin interrupt processing in the same cycle that an **MVC** instruction writes 0 to GIE to disable interrupts. The PGIE bit (bit 1 of CSR) records the value of GIE after the CPU begins interrupt processing, recording whether the program was in the process of disabling interrupts.

During maskable interrupt processing, the CPU finishes executing the current execute packet. The CPU then copies the current value of GIE to PGIE, overwriting the previous value of PGIE. The CPU then clears GIE to prevent another maskable interrupt from occurring before the handler saves the machine's state. (Section 5.7.2 discusses nesting interrupts.)

When the interrupt handler returns to the interrupted code with the **B IRP** instruction, the CPU copies PGIE back to GIE. When the interrupted code resumes, GIE reflects the last value written by the interrupted code.

Because interrupt detection occurs in parallel with CPU execution, the CPU can take an interrupt in the cycle immediately following an **MVC** instruction that clears GIE. The behavior of PGIE and the **B IRP** instruction ensures, however, that interrupts do not occur after subsequent execute packets. Consider the code in Example 5-2.

Example 5-2. Interrupts Versus Writes to GIE

```
                    ;Assume GIE = 1
MVC   CSR,B0        ; (1) Get CSR
AND   -2,B0,B0      ; (2) Get ready to clear GIE
MVC   B0,CSR        ; (3) Clear GIE
ADD   A0,A1,A2      ; (4)
ADD   A3,A4,A5      ; (5)
```

In Example 5-2, the CPU may service an interrupt between instructions 1 and 2, between instructions 2 and 3, or between instructions 3 and 4. The CPU will not service an interrupt between instructions 4 and 5.

If the CPU services an interrupt between instructions 1 and 2 or between instructions 2 and 3, the PGIE bit will hold the value 1 when arriving at the interrupt service routine. If the CPU services an interrupt between instructions 3 and 4, the PGIE bit will hold the value 0. Thus, when the interrupt service routine resumes the interrupted code, it will resume with GIE set as the interrupted code intended.

On the C64x CPU, programs must directly manipulate the GIE bit in CSR to disable and enable interrupts. Example 5-3 and Example 5-4 show code examples for disabling and enabling maskable interrupts globally, respectively.

Example 5-3. TMS320C64x 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 5-4. TMS320C64x 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
```

The C64x+ CPU handles this process differently, in a manner that is backward compatible with the techniques that the C64x CPU requires. When it begins processing of a maskable interrupt, the C64x+ CPU copies TSR to ITSR, thereby, saving the old value of GIE. It then clears TSR.GIE. (ITSR.GIE is physically the same bit as CSR.PGIE and TSR.GIE is physically the same bit as CSR.GIE.) When returning from an interrupt with the **B IRP** instruction, the CPU restores the TSR state by copying ITSR back to TSR.

The C64x+ CPU provides two new instructions that allow for simpler and safer manipulation of the GIE bit.

- The **DINT** instruction disables interrupts by:
  - Copies the value of CSR.GIE (and TSR.GIE) to TSR.SGIE
  - Clears CSR.GIE and TSR.GIE to 0 (disabling interrupts immediately)

  The CPU will not service an interrupt between the execute packet containing **DINT** and the execute packet that follows it.
- The **RINT** instruction restores interrupts to the previous state by:
  - Copies the value of TSR.SGIE to CSR.GIE (and TSR.GIE)
  - Clears TSR.SGIE to 0

If SGIE bit in TSR when **RINT** executes, interrupts are enabled immediately and the CPU may service an interrupt in the cycle immediately following the execute packet containing **RINT**.

Example 5-5 illustrates the use and timing of the **DINT** instruction in disabling maskable interrupts globally and Example 5-6 shows how to enable maskable interrupts globally using the complementary **RINT** instruction.

Example 5-5. Code Sequence with Disable Global Interrupt Enable (C64x+ CPU only)

```
                    ;Assume GIE = 1
ADD   B0,1,B0       ; Interrupt possible between ADD and DINT
DINT                ; No interrupt between DINT and SUB
SUB   B0,1,B0       ;
```

Example 5-6. Code Sequence with Restore Global Interrupt Enable (C64x+ CPU only)

```
                    ;Assume SGIE == 1, GIE = 0
ADD   B0,1,B0       ; No Interrupt between ADD and RINT
RINT                ; Interrupt possible between RINT and SUB
SUB   B0,1,B0       ;
```

Example 5-7 shows a code fragment in which a load/modify/store is executed with interrupts disabled so that the register cannot be modified by an interrupt between the read and write operation. Since the **DINT** instruction saves the CSR.GIE bit to the TSR.SGIE bit and the **RINT** instruction copies the TSR.SGIE bit back to the CSR.GIE bit, if interrupts were disabled before the **DINT** instruction, they will still be disabled after the **RINT** instruction. If they were enabled before the **DINT** instruction, they will be enabled after the **RINT** instruction.

Example 5-7. Code Sequence with Disable Reenable Interrupt Sequence (C64x+ CPU only)

```
DINT                ; Disable interrupts
LDW   *B0,B1        ; Load data
NOP   3             ; Wait for data to reach register
OR    B1,1,B1       ; Set bit in word
STW   B1,*B0        ; Store modified data back to original location
RINT                ; Re−enable interrupts
```

NOTE: The use of **DINT** and **RINT** instructions in a nested manner, like the following code:

```
DINT
DINT
RINT
RINT
```

leaves interrupts disabled after the second **RINT** instruction. The successive use of the **DINT** instruction leaves the TSR.SGIE bit cleared to 0, so the **RINT** instructions copy zero to the GIE bit.

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 2-8 and described in Table 2-12.

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 5-8 and Example 5-9 show code for enabling and disabling individual interrupts, respectively.

Example 5-8. 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 5-9. 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 2-9 and described in Table 2-13.

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 2-11 and described in Table 2-14. The ICR is shown in Figure 2-7 and described in Table 2-11.

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 5-10 and Example 5-11 show code examples to set and clear individual interrupts.

Example 5-10. Code to Set an Individual Interrupt (INT6) and Read the Flag Register

```
MVK   40h,B3
MVC   B3,ISR
NOP
MVC   IFR,B4
```

Example 5-11. 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

Returning control from interrupts is handled differently for all three types of interrupts: reset, nonmaskable, and maskable.

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, and IFR = 0
- ISTP = Default value varies by device (See data manual for correct value)
- IER = 1
- IRP and NRP = undefined
- CSR bits 15-0
  = 100h in little-endian mode
  = 000h in big-endian mode
- TSR = 0 (C64x+ CPU only)
- ITSR = 0 (C64x+ CPU only)
- NTSR = 0 (C64x+ CPU only)

The program execution begins at the address specified by the ISTB field in ISTP.

5.3.4.2 Returning From Nonmaskable Interrupts

The NMI return pointer register (NRP), shown in Figure 2-13, 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 5-12 shows how to return from an NMI.

The NTSR register will be copied back into the TSR register during the transfer of control out of the interrupt.

Example 5-12. 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 2-10, 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 5-13 shows how to return from a maskable interrupt.

The ITSR will be copied back into the TSR during the transfer of control out of the interrupt.

Example 5-13. Code to Return from a Maskable Interrupt

```
B      IRP            ; return, moves PGIE to GIE
NOP    5              ; delay slots
```

5.4 Interrupt Detection and Processing on the C64x CPU

This section describes interrupts on the C64x CPU. For information about interrupts on the C64x+ CPU, see Section 5.5.

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 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.

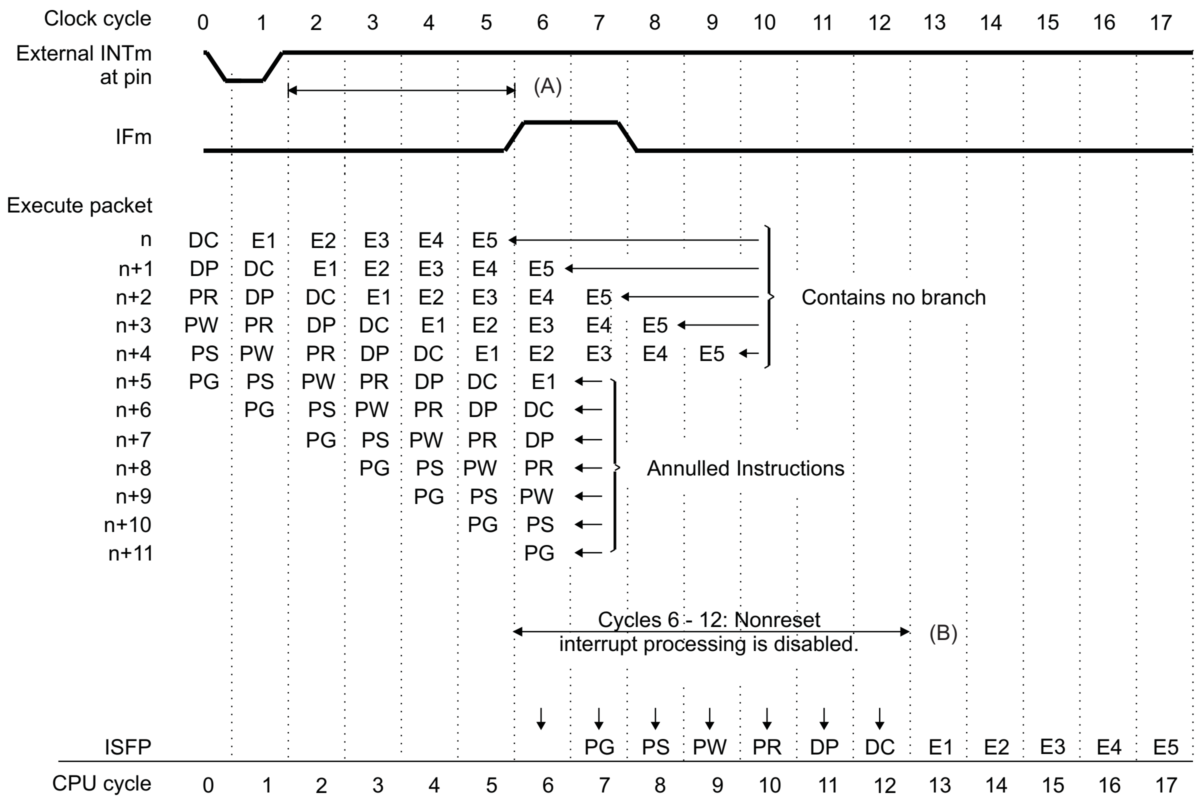
Figure 5-4 assumes INTm is the highest-priority pending interrupt and is enabled by the GIE and NMIE bits, as necessary. If it is not the highest-priority pending interrupt, IFm remains set until either you clear it by writing a 1 to bit m of ICR or the processing of INTm occurs.

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 IFR.
- The corresponding bit in 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).

Figure 5-4. Nonreset Interrupt Detection and Processing: Pipeline Operation



A IFm is set on the next CPU cycle boundary after a 4-clock cycle delay after the rising edge of INTm.

B After this point, interrupts are still disabled. All nonreset interrupts are disabled when NMIE = 0. All maskable interrupts are disabled when GIE = 0.

Any pending interrupt will be taken as soon as pending branches are completed.

5.4.3 Actions Taken During Nonreset Interrupt Processing

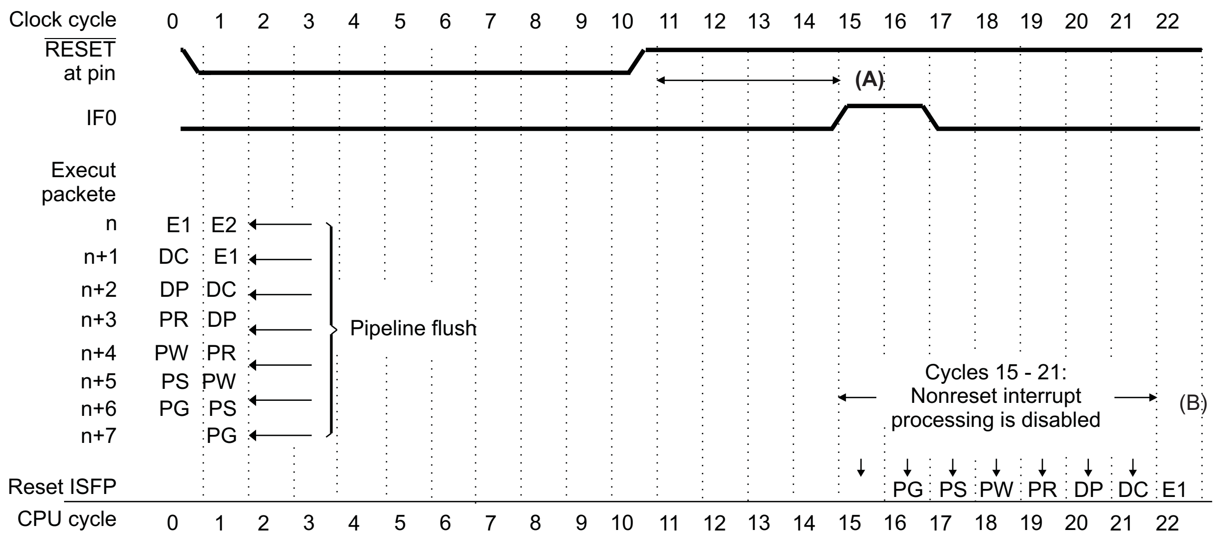
During CPU cycles 6 through 12 of Figure 5-4, the following interrupt processing actions occur:

- 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).
- A branch to the address held in ISTP (the pointer to the ISFP for INTm) is forced into the E1 phase of the pipeline during cycle 7.
- IFm is cleared during cycle 8.

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



A IF0 is set on the next CPU cycle boundary after a 4-clock cycle delay after the rising edge of .

B 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. The GIE bit, the NMIE bit, and the ISTB bits in 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.
- 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 Interrupt Detection and Processing on the C64x+ CPU

This section describes interrupts on the C64x+ CPU. For information about interrupts on the C64x CPU, see Section 5.4.

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.

5.5.1 Setting the Nonreset Interrupt Flag

Figure 5-6 shows the processing of a nonreset interrupt (INTm) for the C64x+ CPU. The flag (IFm) for INTm in 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 interrupt pin, the interrupt is detected as pending inside the CPU. When the interrupt signal has been detected (cycle 4). Two clock cycles after detection, the interrupt's corresponding flag bit in IFR is set (cycle 6).

In Figure 5-6, IFm is set during CPU cycle 6. You could attempt to clear bit IFm by using an **MVC** instruction to write a 1 to bit m of 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 5-6 assumes INTm is the highest priority pending interrupt and is enabled by the GIE and NMIE bits, 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 ICR, or the processing of INTm occurs.

5.5.1.1 Detection of Missed Interrupts

Each INTm input has a corresponding output that indicates if a low-to-high transition occurred on the input while the pending flag for that input had not yet been cleared. These outputs may be used by the interrupt controller to create an exception back to the CPU to notify the user of the missed interrupt. See your device-specific data manual to verify your device supports this feature.

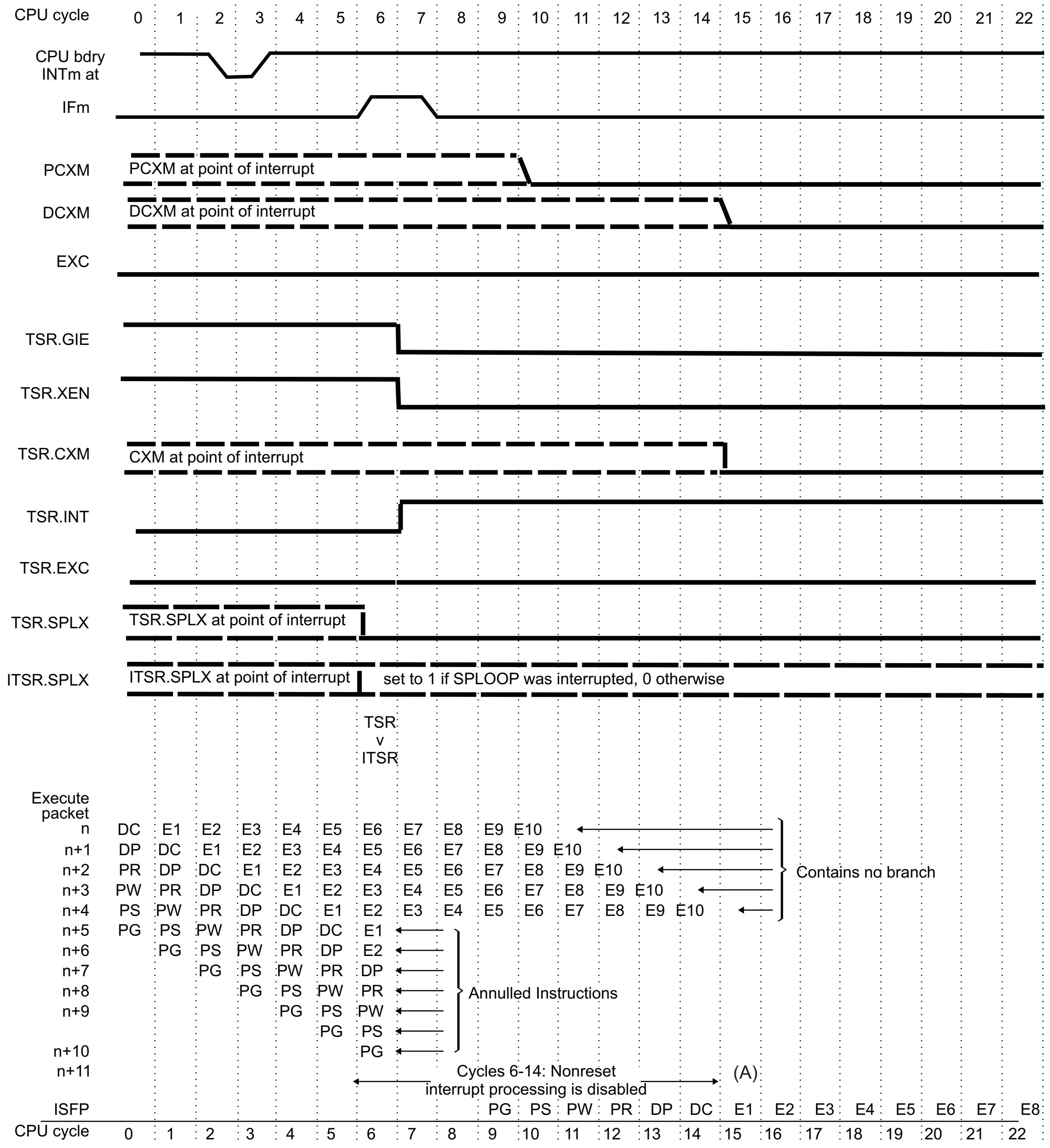
5.5.2 Conditions for Processing a Nonreset Interrupt

In clock cycle 4 of Figure 5-6, 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 IFR.
- The corresponding bit in 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.
- The two previous execute packets and the current execute packet (n + 3 through n + 5) do note contain an **SPLOOP, SPLOOPD,** or **SPLOOPW** instruction.
- If an SPLOOP is active, then the conditions set forth in Section 7.13.1 apply.

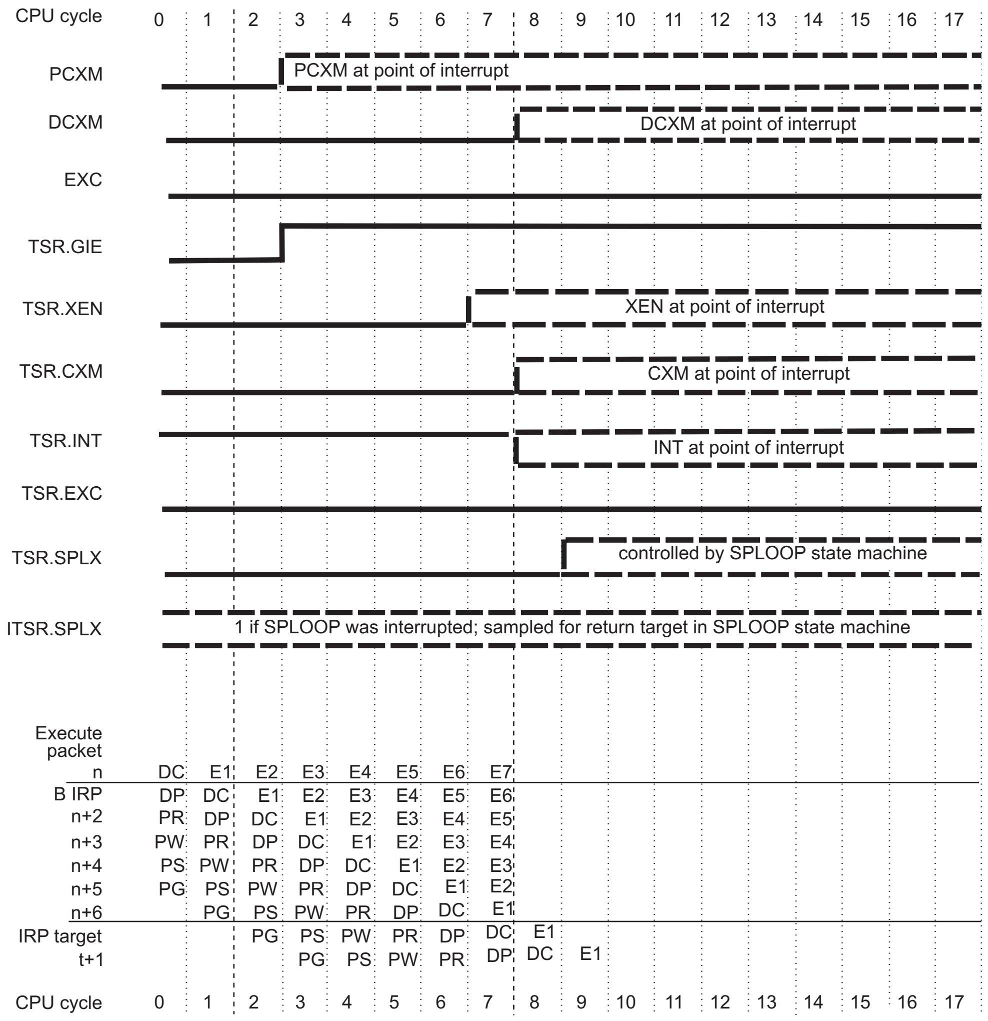
Any pending interrupt will be taken as soon as pending branches are completed.

Figure 5-6. C64x+ Nonreset Interrupt Detection and Processing: Pipeline Operation


CPU cycle
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22
CPU bdry INTm at
IFm
PCXM
PCXM at point of interrupt
DCXM
DCXM at point of interrupt
EXC
TSR.GIE
TSR.XEN
TSR.CXM
CXM at point of interrupt
TSR.INT
TSR.EXC
TSR.SPLX
TSR.SPLX at point of interrupt
ITSR.SPLX
ITSR.SPLX at point of interrupt
set to 1 if SPLOOP was interrupted, 0 otherwise
TSR v ITSR
Execute packet
n DC E1 E2 E3 E4 E5 E6 E7 E8 E9 E10
n+1 DP DC E1 E2 E3 E4 E5 E6 E7 E8 E9 E10
n+2 PR DP DC E1 E2 E3 E4 E5 E6 E7 E8 E9 E10
n+3 PW PR DP DC E1 E2 E3 E4 E5 E6 E7 E8 E9 E10
n+4 PS PW PR DP DC E1 E2 E3 E4 E5 E6 E7 E8 E9 E10
Contains no branch
n+5 PG PS PW PR DP DC E1
n+6 PG PS PW PR DP E2
n+7 PG PS PW PR DP
n+8 PG PS PW PR
n+9 PG PS PW
PG PS
n+10 PG
Annulled Instructions
n+11
Cycles 6-14: Nonreset interrupt processing is disabled
(A)
ISFP PG PS PW PR DP DC E1 E2 E3 E4 E5 E6 E7 E8
CPU cycle 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22

A After this point, interrupts are still disabled. All nonreset interrupts are disabled when NMIE = 0. All maskable interrupts are disabled when GIE = 0.

Figure 5-7. C64x+ Return from Interrupt Execution and Processing: Pipeline Operation


CPU cycle
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
PCXM
PCXM at point of interrupt
DCXM
DCXM at point of interrupt
EXC
TSR.GIE
TSR.XEN
XEN at point of interrupt
TSR.CXM
CXM at point of interrupt
TSR.INT
INT at point of interrupt
TSR.EXC
TSR.SPLX
controlled by SPLOOP state machine
ITSR.SPLX
1 if SPLOOP was interrupted; sampled for return target in SPLOOP state machine
Execute packet
n: DC E1 E2 E3 E4 E5 E6 E7
B IRP: DP DC E1 E2 E3 E4 E5 E6
n+2: PR DP DC E1 E2 E3 E4 E5
n+3: PW PR DP DC E1 E2 E3 E4
n+4: PS PW PR DP DC E1 E2 E3
n+5: PG PS PW PR DP DC E1 E2
n+6: PG PS PW PR DP DC E1
IRP target: PG PS PW PR DP DC E1
t+1: PG PS PW PR DP DC E1

5.5.3 Saving TSR Context in Nonreset Interrupt Processing

When control is transferred to the interrupt processing sequence, the context needed to return from the ISR is saved in the interrupt task state register (ITSR). The task state register (TSR) is set for the default interrupt processing context. Table 5-3 shows the behavior for each bit in TSR. Figure 5-6 shows the timing of the changes to the TSR bits as well as the CPU outputs used in interrupt processing.

The current execution mode is held in a piped series of register bits allowing a change in the mode to progress from the PS phase through the E1 phase. Fetches from program memory use the PS-valid register which is only loaded at the start of a transfer of control. This value is an output on the program memory interface and is shown in the timing diagram as PCXM. As the target execute packet progresses through the pipeline, the new mode is registered for that stage. Each stage uses its registered version of the execution mode. The field in TSR is the E1-valid version of CXM. It always indicates the execution mode for the instructions executing in E1. The mode is used in the data memory interface, and is registered for all load/store instructions when they execute in E1. This is shown in the timing diagram as DCXM. Note that neither PCXM nor DCXM is visible in any register to you.

Table 5-3. TSR Field Behavior When an Interrupt is Taken

Bit	Field	Action
0	GIE	Saved to GIE bit in ITSR (will be 1). Cleared to 0.
1	SGIE	Saved to SGIE bit in ITSR. Cleared to 0.
2	GEE	Saved to GEE bit in ITSR. Unchanged.
3	XEN	Saved to XEN bit in ITSR. Cleared to 0.
7-6	CXM	Saved to CXM bits in ITSR. Cleared to 0 (Supervisor mode).
9	INT	Saved to INT bit in ITSR. Set to 1.
10	EXC	Saved to EXC bit in ITSR. Cleared to 0.
14	SPLX	SPLX is set in the TSR by the SPLOOP buffer whenever it is in operation. Upon interrupt, if the SPLOOP buffer is operating (thus SPLX = 1), then ITSR.SPLX will be set to 1, and the TSR.SPLX bit will be cleared to 0 after the SPLOOP buffer winds down and the interrupt vector is taken. See Section 7.4.5 for more information on SPLOOP.
15	IB	Saved to IB bit in ITSR (will be 0). Set by CPU control logic.

5.5.4 Actions Taken During Nonreset Interrupt Processing

During CPU cycles 6-14 of Figure 5-6, the following interrupt processing actions occur:

- Processing of subsequent nonreset interrupts is disabled.
- The PGIE bit is set to the value of the GIE bit and then the GIE bit is cleared. TSR context is saved into ITSR, and TSR is set to default interrupt processing values as shown in Table 5-3. Explicit (MVC) writes to the TSR are completed before the TSR is saved to the ITSR.
- 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 IRP.
- A branch to the address formed from ISTP (the pointer to the ISFP for INTm) is forced into the E1 phase of the pipeline during cycle 9.
- IFm is cleared during cycle 8.

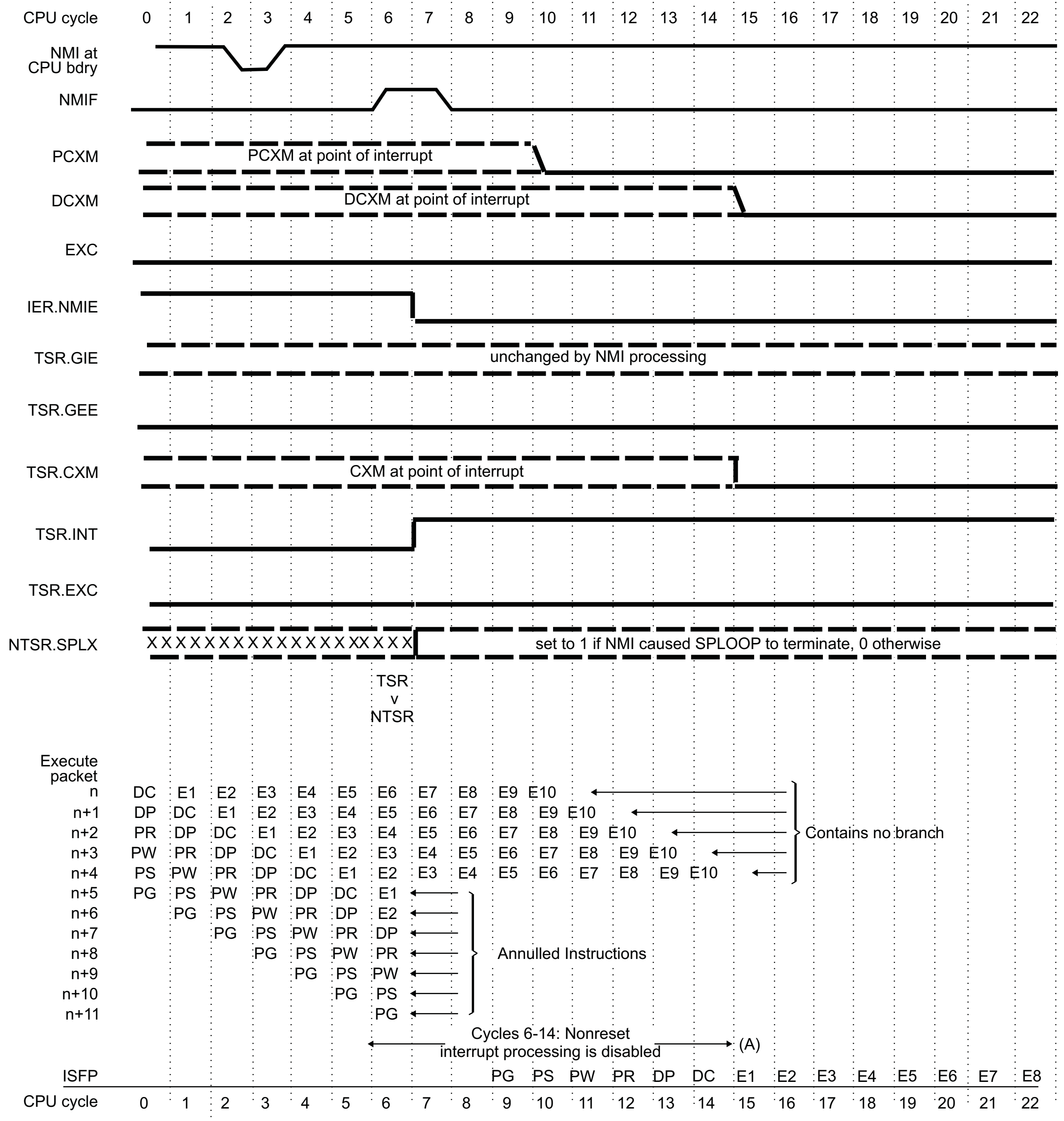
5.5.5 Conditions for Processing a Nonmaskable Interrupt

In clock cycle 4 of Figure 5-8, a nonmaskable interrupt (NMI) 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:

- The NMIF bit is set during CPU cycle 6. (This determination is made in CPU cycle 4 by the interrupt logic.)
- Reset is not active.
- 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. Note that this functionality has changed when exceptions are enabled, see Chapter 6 for more information.

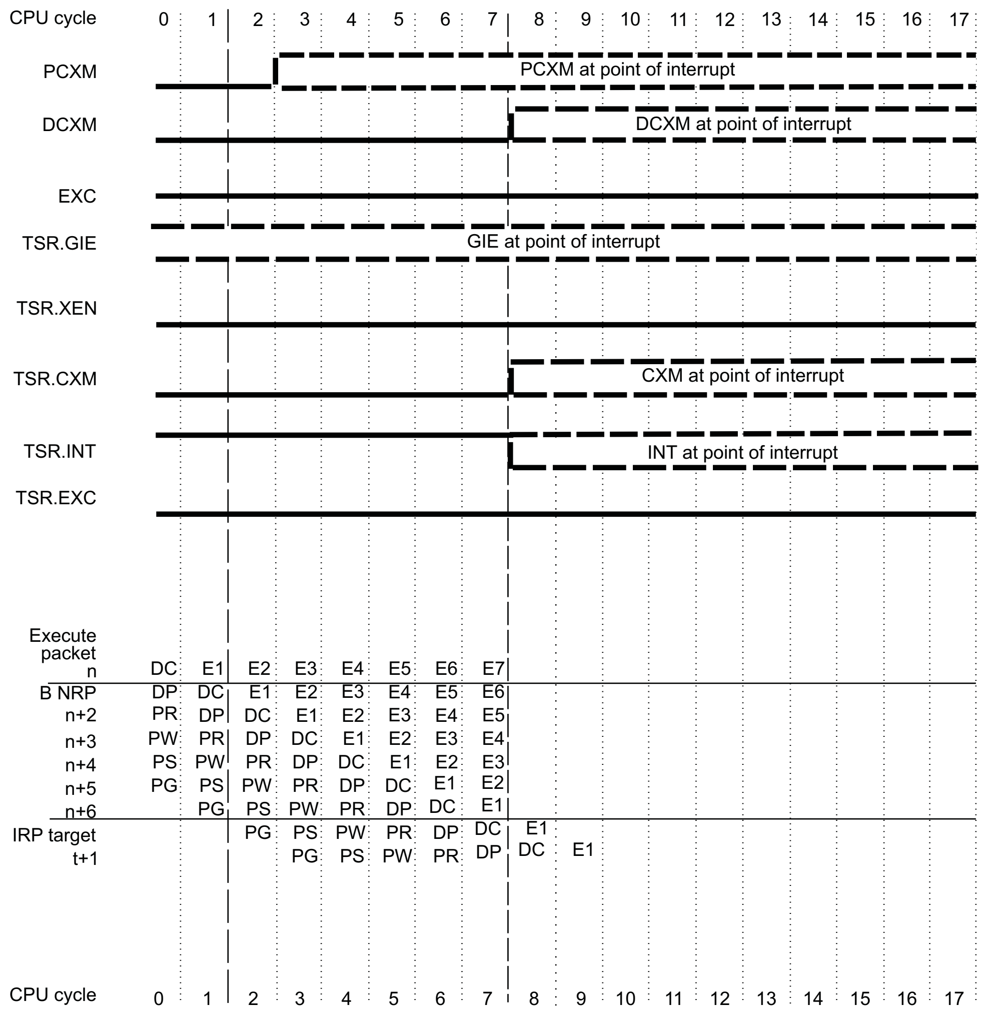
A pending NMI will be taken as soon as pending branches are completed.

Figure 5-8. C64x+ CPU Nonmaskable Interrupt Detection and Processing: Pipeline Operation


CPU cycle
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22
NMI at CPU bdry
NMIF
PCXM
PCXM at point of interrupt
DCXM
DCXM at point of interrupt
EXC
IER.NMIE
TSR.GIE
unchanged by NMI processing
TSR.GEE
TSR.CXM
CXM at point of interrupt
TSR.INT
TSR.EXC
NTSR.SPLX
XXXXXXXXXXXXXXXXXXXXX
set to 1 if NMI caused SPLOOP to terminate, 0 otherwise
TSR v NTSR
Execute packet
n DC E1 E2 E3 E4 E5 E6 E7 E8 E9 E10
n+1 DP DC E1 E2 E3 E4 E5 E6 E7 E8 E9 E10
n+2 PR DP DC E1 E2 E3 E4 E5 E6 E7 E8 E9 E10
n+3 PW PR DP DC E1 E2 E3 E4 E5 E6 E7 E8 E9 E10
n+4 PS PW PR DP DC E1 E2 E3 E4 E5 E6 E7 E8 E9 E10
Contains no branch
n+5 PG PS PW PR DP DC E1
n+6 PG PS PW PR DP E2
n+7 PG PS PW PR DP
n+8 PG PS PW PR
n+9 PG PS PW
n+10 PG PS
n+11 PG
Annulled Instructions
Cycles 6-14: Nonreset interrupt processing is disabled
(A)
ISFP PG PS PW PR DP DC E1 E2 E3 E4 E5 E6 E7 E8
CPU cycle
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22

A After this point, interrupts are still disabled. All nonreset interrupts are disabled when NMIE = 0. All maskable interrupts are disabled when GIE = 0.

Figure 5-9. C64x+ CPU Return from Nonmaskable Interrupt Execution and Processing: Pipeline Operation



	0	1	2	3	4	5	6	7	8	9
n	DC	E1	E2	E3	E4	E5	E6	E7		
B NRP	DP	DC	E1	E2	E3	E4	E5	E6		
n+2	PR	DP	DC	E1	E2	E3	E4	E5		
n+3	PW	PR	DP	DC	E1	E2	E3	E4		
n+4	PS	PW	PR	DP	DC	E1	E2	E3		
n+5	PG	PS	PW	PR	DP	DC	E1	E2		
n+6		PG	PS	PW	PR	DP	DC	E1		
IRP target			PG	PS	PW	PR	DP	DC	E1	
t+1				PG	PS	PW	PR	DP	DC	E1

5.5.6 Saving of Context in Nonmaskable Interrupt Processing

When control is transferred to the interrupt processing sequence, the context needed to return from the ISR is saved in the nonmaskable interrupt task state register (NTSR). The task state register (TSR) is set for the default NMI processing context. Table 5-4 shows the behavior for each bit in TSR. Figure 5-8 shows the timing of the changes to the TSR bits as well as the CPU outputs used in interrupt processing.

Table 5-4. TSR Field Behavior When an NMI Interrupt is Taken

Bit	Field	Action
0	GIE	Saved to GIE bit in NTSR. Unchanged.
1	SGIE	Saved to SGIE bit in NTSR. Cleared to 0.
2	GEE	Saved to GEE bit in NTSR. Unchanged.
3	XEN	Saved to XEN bit in NTSR. Cleared to 0.
7-6	CXM	Saved to CXM bits in NTSR. Cleared to 0 (Supervisor mode).
9	INT	Saved to INT bit in NTSR. Set to 1.
10	EXC	Saved to EXC bit in NTSR. Cleared to 0.
14	SPLX	SPLX is set in the TSR by the SPLOOP buffer whenever it is in operation. Upon interrupt, if the SPLOOP buffer is operating (thus SPLX = 1), then ITSR.SPLX will be set to 1, and the TSR.SPLX bit will be cleared to 0 after the SPLOOP buffer winds down and the interrupt vector is taken. See Section 7.4.5 for more information on SPLOOP.
15	IB	Saved to IB bit in NTSR. Set by CPU control logic.

5.5.7 Actions Taken During Nonmaskable Interrupt Processing

During CPU cycles 6-14 of Figure 5-8, the following interrupt processing actions occur:

- Processing of subsequent nonreset interrupts is disabled.
- The GIE and PGIE bits are unchanged. TSR context is saved into NTSR, and TSR is set to default NMI processing values as shown in Table 5-4.
- 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.
- A branch to the NMI ISFP (derived from ISTP) is forced into the E1 phase of the pipeline during cycle 9.
- NMIF is cleared during cycle 8.

5.5.8 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 5-10, 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.

5.5.9 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. The GIE bit, the NMIE bit, and the ISTB bits in ISTP are cleared. For the CPU state after reset, see Section 5.3.4.1.

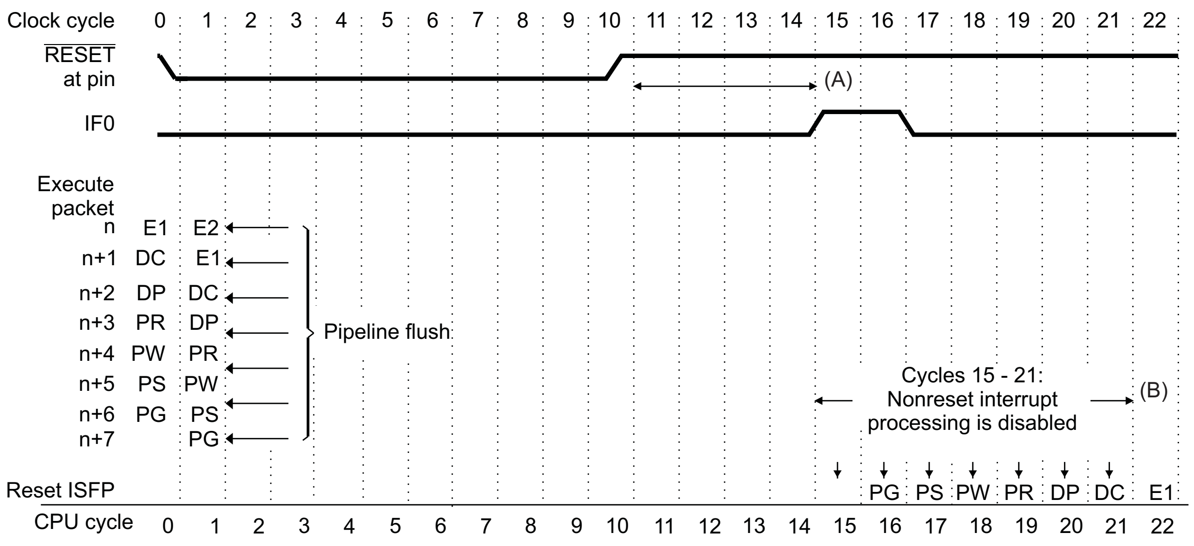
Note that a nested exception can force an internally-generated reset that does not reset all the registers to their hardware reset state. See Section 6.3.4 for more information.

During CPU cycles 15-21 of Figure 5-10, 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.
- 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.

Figure 5-10. $\overline{\text{RESET}}$ Interrupt Detection and Processing: Pipeline Operation



A IF0 is set on the next CPU cycle boundary after a 4-clock cycle delay after the rising edge of .

B After this point, interrupts are still disabled. All nonreset interrupts are disabled when NMIE = 0. All maskable interrupts are disabled when GIE = 0.

5.6 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.

5.6.1 General Performance

- **Overhead**. Overhead for all CPU interrupts on the C64x CPU is 7 cycles. You can see this in Figure 5-4, where no new instructions are entering the E1 pipeline phase during CPU cycles 6 through 12.

  Overhead for all CPU interrupts on the C64x+ CPU is 9 cycles. You can see this in Figure 5-6 and Figure 5-7, where no new instructions are entering the E1 pipeline phase during CPU cycles 6 through 14.
- **Latency**. Interrupt latency on the C64x CPU is 11 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 13.

  Interrupt latency on the C64x+ CPU is 13 cycles (21 cycles for $\overline{\text{RESET}}$) . In Figure 5-8, 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.

5.6.2 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.

5.7 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.

5.7.1 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 5-14 shows a code fragment which stores two variables into A1 using multiple assignment. Example 5-15 shows equivalent code using the single assignment programming method which stores the two variables into two different registers.

For example, before reaching the code in Example 5-14, suppose that register A1 contains 0 and register A0 points to a memory location containing a value of 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 5-15, 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 5-14. 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 5-15. Code Using Single Assignment

```
LDW   .D1   *A0,A6
ADD   .L1   A1,A2,A3
NOP   3
MPY   .M1   A6,A4,A5     ; uses A6
```

Another method for preventing problems with nonsingle-assignment programming would be to disable interrupts before using multiple assignment, then reenable them afterwards. Of course, you must be careful with the tradeoff between high-speed code that uses multiple-assignment and increasing interrupt latency. When using multiple assignment within software pipelined code, the SPLOOP buffer on the C64x+ CPU can help you deal with the tradeoff between performance and interruptibility. See Chapter 7 for more information.

5.7.2 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.

Also, 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 contents of ITSR must be saved (C64x+ CPU only)
4. 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 contents of ITSR must be restored (C64x+ CPU only)
4. The IRP (or NRP) saved value must be restored

Although steps 2, 3, and 4 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 5-16 shows a simple assembly interrupt handler that allows nested interrupts on the C64x 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. To adapt this code to run on the C64x+ CPU, the ITSR would also need to be saved.

Example 5-17 shows a C-based interrupt handler that allows nested interrupts on the C64x CPU. The steps are similar, although the compiler takes care of allocating the stack and saving CPU registers. To adapt this code to run on the C64x+ CPU, the ITSR would also need to be restored. For more information on using C to access control registers and write interrupt handlers, see the *TMS320C6000 Optimizing Compiler User's Guide*, SPRU187.

NOTE: When coding nested interrupts for the C64x+ CPU, the ITSR should be saved and restored to prevent corruption by the nested interrupt.

Example 5-16. 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             ;/    (Clear 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 5-17. 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                            */
```

Example 5-17 uses the interrupt keyword along with explicit context save and restore code. An alternative is to use the DSP/BIOS interrupt dispatcher that also provides an easy way to nest interrupt service routines.

5.7.3 Manual Interrupt Processing (polling)

You can poll IFR and IER to detect interrupts manually and then branch to the value held in the ISTP as shown below in Example 5-18.

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 5-18. Manual Interrupt Processing

```
          MVC    ISTP,B2                 ; get related ISFP address
          EXTU   B2,23,27,B1             ; extract HPEINT
     [B1] B      B2                      ; branch to interrupt
||   [B1] MVKL   1,A0                    ; setup ICR word
     [B1] MVKL   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)
```

5.7.4 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: A0, 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 5-19 and Example 5-20 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.

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 5-20 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 5-19. Code Sequence to Invoke a Trap

```
[A1]  MVKL   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]  MVKL   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 5-20. Code Sequence for Trap Return

```
B      B1          ; return
MVC    B0,CSR      ; restore CSR
NOP    4           ; delay slots
```

Often traps are used to handle unexpected conditions in the execution of the code. The C64x+ CPU provides explicit exception handling support which may be used for this purpose.

Another alternative to using traps as software triggered interrupts is the software interrupt capability (SWI) provided by the DSP/BIOS real-time kernel.

5.8 Differences Between C64x and C64x+ CPU Interrupts

Table 5-5 summarizes the differences between the interrupt function on the C64x CPU and the C64x+ CPU.

Table 5-5. Differences Between C64x and C64x+ CPU Interrupts

Function	C64x CPU	C64x+ CPU
Interrupt latency	7 cycles	9 cycles
Interrupt overhead	11 cycles	13 cycles
Exceptions	No	Yes
TSR, ITSR, and NTSR registers	No	Yes
DINT and **RINT** instructions	No	Yes
On interrupt	GIE bit in CSR copied to PGIE bit in CSR	TSR copied to ITSR
On return from interrupt	PGIE bit in CSR copied to GIE bit in CSR	ITSR copied to TSR
Default location of IST	0000 0000h	Varies. Check device-specific data manual for correct value.
NMI used as	Nonmaskable interrupt	Nonmaskable Interrupt, or Exception processing
SPLOOP buffer	None	Provides interruptible software pipelined code
Interrupt branch-in-progress	No	Yes (via exceptions)
Missed interrupt detection	No	Yes
Number of instruction in ISFP	8	Up to 14
Flag to show when processing hardware interrupts	No	Yes (TSR.INT)

C64x+ CPU Exceptions

This chapter describes CPU exceptions on the C64x+ CPU. It details the related CPU control registers and their functions in controlling exceptions. It also describes exception processing, the method the CPU uses to detect automatically the presence of exceptions and divert program execution flow to your exception service code. Finally, the chapter describes the programming implications of exceptions.

The C64x CPU does not support exceptions. This chapter applies only to the C64x+ CPU.

Topic **Page**

6.1 Overview

The exception mechanism on the C64x+ CPU is intended to support error detection and program redirection to error handling service routines. Error signals generated outside of the CPU are consolidated to one exception input to the CPU. Exceptions generated within the CPU are consolidated to one internal exception flag with information as to the cause in a register. Fatal errors detected outside of the CPU are consolidated and incorporated into the NMI input to the CPU.

6.1.1 Types of Exceptions and Signals Used

There are three types of exceptions on the C64x+ CPU.

- one externally generated maskable exception
- one externally generated nonmaskable exception,
- a set of internally generated nonmaskable exceptions

Check the device-specific data manual for your external exception specifications.

6.1.1.1 Reset ($\overline{\text{RESET}}$)

While reset can be classified as an exception, its behavior is fully described in the chapter on interrupts and its operation is independent of the exception mechanism.

6.1.1.2 Nonmaskable Interrupt (NMI)

NMI is also described in the interrupt chapter, and as stated there it is generally used to alert the CPU of a serious hardware problem. The intent of NMI on C6000 devices was clearly for use as an exception. However, the inability of NMI to interrupt program execution independent of branch delay slots lessens its usefulness as an exception input. By default the NMI input retains its behavior for backward compatibility. When used in conjunction with the exception mechanism it will be treated as a nonmaskable exception with the primary behavioral difference being that branch delay slots will not prevent the recognition of NMI. The new behavior is enabled when exceptions are globally enabled. This is accomplished by setting the global exception enable (GEE) bit in the task state register (TSR) to 1. Once the exception mechanism has been enabled, it remains enabled until a reset occurs (GEE can only be cleared by reset). When the GEE bit is set to 1, NMI behaves as an exception. All further discussion of NMI in this chapter is in reference to its behavior as an exception.

For NMI processing to occur, the nonmaskable interrupt enable (NMIE) bit in the interrupt enable register (IER) must be set to 1. If the NMIE bit is set to 1, the only condition that can prevent NMI processing is the CPU being stalled.

The NMIE bit is cleared to 0 at reset to prevent interruption of the reset processing. 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 NMIE is cleared, all external exceptions are disabled. Internal exceptions are not affected by NMIE.

When NMI is recognized as pending, the NMI exception flag (NXF) bit in the exception flag register (EFR) is set. Unlike the NMIF bit in the interrupt flag register (IFR), the NXF bit is not cleared automatically upon servicing of the NMI. The NXF bit remains set until manually cleared in the exception service routine.

Transitions on the NMI input while the NXF bit is set are ignored. In the event an attempt to clear the flag using the MVC instruction coincides with the automated write by the exception detection logic, the automatic write takes precedence and the NXF bit remains set.

6.1.1.3 Exception (EXCEP)

EXCEP is the maskable external exception input to the CPU. It is enabled by the XEN bit in TSR. For this exception to be recognized, the XEN bit must be set to 1, the GEE bit must be set to 1, and the NMIE bit must be set to 1.

When EXCEP is recognized as pending, the external exception flag (EXF) bit in EFR is set. The EXF bit remains set until manually cleared in the exception service routine.

6.1.1.4 Internal Exceptions

Internal exceptions are those generated within the CPU. There are multiple causes for internal exceptions. Examples are illegal opcodes, illegal behavior within an instruction, and resource conflicts. Once enabled by the GEE bit, internal exceptions cannot be disabled. They are recognized independently of the state of the NMIE and XEN exception enable bits.

Instructions that have already entered E1 before a context switch begins are allowed to complete. Any internal exceptions generated by these completing instructions are ignored. This is true for both interrupt and exception context switches.

When an internal exception is recognized as pending, the internal exception flag (IXF) bit in EFR is set. The IXF bit remains set until manually cleared in the exception service routine.

6.1.1.5 Exception Acknowledgment

The exception processing (EXC) bit in TSR is provided as an output at the CPU boundary. This signal in conjunction with the IACK signal alerts hardware external to the C64x+ CPU that an exception has occurred and is being processed.

6.1.2 Exception Service Vector

When the CPU begins processing an exception, it references the interrupt service table (IST). The NMI interrupt service fetch packet (ISFP) is the fetch packet used to service all exceptions (external, internal, and NMI).

In general, the exception service routine for an exception is too large to fit in a single fetch packet, so a branch to the location of additional exception service routine code is required. This is shown in Figure 6-1.

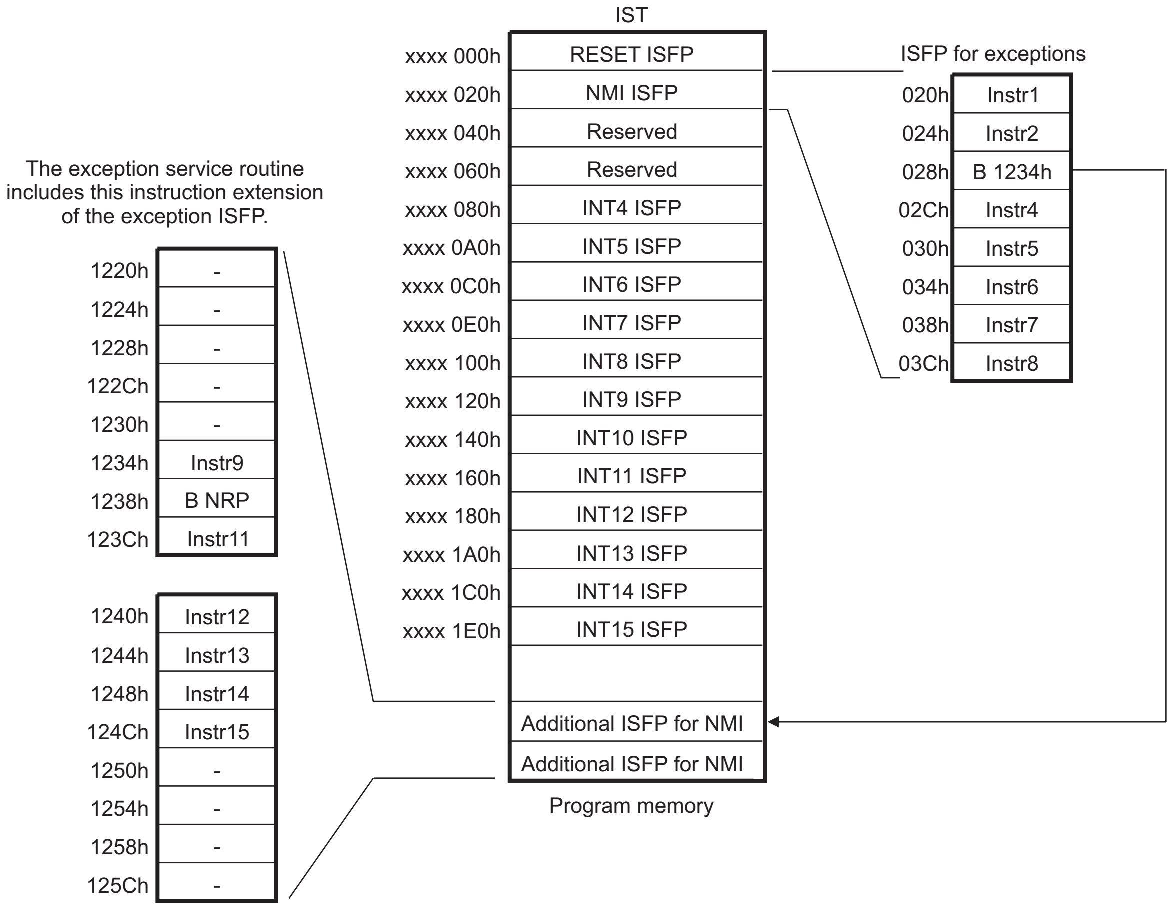
6.1.3 Summary of Exception Control Registers

Table 6-1 lists the control registers related to exceptions on the C64x+ CPU.

Table 6-1. Exception-Related Control Registers

Acronym	Register Name	Description	Section
ECR	Exception clear register	Used to clear pending exception flags	Section 2.9.3
EFR	Exception flag register	Contains pending exception flags	Section 2.9.4
IER	Interrupt enable register	Contains NMI exception enable (NMIE) bit	Section 2.8.7
IERR	Internal exception report register	Indicates cause of an internal exception	Section 2.9.7
ISTP	Interrupt service table pointer register	Pointer to the beginning of the interrupt service table that contains the exception interrupt service fetch packet	Section 2.8.11
NRP	Nonmaskable interrupt return pointer register	Contains the return address used on return from an exception. This return is accomplished via the **B NRP** instruction	Section 2.8.12
NTSR	Nonmaskable interrupt/exception task state register	Stores contents of TSR upon taking an exception	Section 2.9.10
REP	Restricted entry point address register	Contains the address to where the **SWENR** instruction transfers control	Section 2.9.11
TSR	Task state register	Contains global exception enable (GEE) and exception enable (XEN) bits	Section 2.9.15

Figure 6-1. Interrupt Service Table With Branch to Additional Exception Service Code Located Outside the IST



6.2 Exception Control

Enabling and disabling individual exceptions is done with the task state register (TSR) and the interrupt enable register (IER). The status of pending exceptions is stored in the exception flag register (EFR). The nonmaskable interrupt return pointer register (NRP) and the nonmaskable interrupt/exception task state register (NTSR) are used to restore context after servicing exceptions. In many cases it is not possible to return to the interrupted code since exceptions can be taken at noninterruptible points.

6.2.1 Enabling and Disabling External Exceptions

Exceptions are globally enabled by the GEE bit in TSR. This bit must be set to 1 to enable any exception processing. Once it is set to 1, the GEE bit can only be cleared by a reset. The GEE bit is the only enable for internal exceptions. Therefore, once internal exceptions have been enabled they cannot be disabled. Global enabling of exceptions also causes the NMI input to be treated as an exception rather than an interrupt.

External exceptions are also qualified by the NMIE bit in IER. An external exception (EXCEP or NMI) can trigger exception processing only if this bit is set. Internal exceptions are not affected by NMIE. The IER is shown in Figure 2-8 and described in Table 2-12. The EXCEP exception input can also be disabled by clearing the XEN bit in TSR.

When NMIE = 0, all interrupts and external exceptions are disabled, preventing interruption of an exception service routine. The NMIE bit is cleared at reset to prevent any interruption of processor initialization until you enable exceptions. After reset, you must set the NMIE bit to enable external exceptions and to allow INT15-INT4 to be enabled by the GIE bit and the appropriate 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 NMIE.

6.2.2 Pending Exceptions

EFR contains four bits that indicate which exceptions have been detected. It is possible for all four bits to be set when entering the exception service routine. The prioritization and handling of multiple exceptions is left to software. Clearing of the exception flags is done by writing a 1 to the bit position to be cleared in the exception clear register (ECR). Bits that are written as 0 to ECR have no effect. The EFR is shown in Figure 2-17 and described in Table 2-18.

6.2.3 Exception Event Context Saving

TSR contains the CPU execution context that is saved into NTSR at the start of exception processing. TSR is then set to indicate the transition to supervisor mode and that exception processing is active. The TSR is shown in Figure 2-28 and described in Table 2-23.

Execution of a **B NRP** instruction causes the saved context in NTSR to be loaded into TSR to resume execution. Similarly, a **B IRP** instruction restores context from ITSR into TSR.

Information about the CPU context at the point of an exception is retained in NTSR. Table 6-2 shows the behavior for each bit in NTSR. The information in NTSR is used upon execution of a **B NRP** instruction to restore the CPU context before resuming the interrupted instruction execution. The HWE bit in NTSR is set when an internal or external exception is taken. The HWE bit is cleared by the **SWE** and **SWENR** instructions. The NTSR is shown in Figure 2-23 and described in Table 2-21.

Table 6-2. NTSR Field Behavior When an Exception is Taken

Bit	Field	Action
0	GIE	GIE bit in TSR at point exception is taken.
1	SGIE	SGIE bit in TSR at point exception is taken.
2	GEE	GEE bit in TSR at point exception is taken (must be 1).
3	XEN	XEN bit in TSR at point exception is taken.
7-6	CXM	CXM bits in TSR at point exception is taken.
9	INT	INT bit in TSR at point exception is taken.
10	EXC	EXC bit in TSR at point exception is taken (must be 0).
14	SPLX	Terminated an SPLOOP
15	IB	Exception occurred while interrupts were blocked.
16	HWE	Hardware exception taken (NMI, EXCEP, or internal).

6.2.4 Returning From Exception Servicing

The NMI return pointer register (NRP) stores the return address used by the CPU to resume correct program execution after NMI processing. A branch using the address in the NRP (**B NRP**) in your exception service routine causes the program to exit the exception service routine and return to normal program execution.

It is not always possible to safely exit the exception handling routine. Conditions that can prevent a safe return from exceptions include:

- SPLOOPs that are terminated by an exception cannot be resumed correctly. The SPLX bit in NTSR should be verified to be 0 before returning.
- Exceptions that occur when interrupts are blocked cannot be resumed correctly. The IB bit in NTSR should be verified to be 0 before returning.
- Exceptions that occur at any point in the code that cannot be interrupted safely (for example, a tight loop containing multiple assignments) cannot be safely returned to. The compiler will normally disable interrupts at these points in the program; check the GIE bit in NTSR to be 1 to verify that this condition is met.

Example 6-1 shows code that checks these conditions.

If the exception cannot be safely returned from, the appropriate response will be different based on the specific cause of the exception. In some cases, a warm reset will be required. In other cases, restarting a user task may be sufficient.

The NRP contains the 32-bit address of the first execute packet in the program flow that was not executed because of an exception. Although you can write a value to this register, any subsequent exception processing may overwrite that value. The NRP is shown in Figure 2-13.

Example 6-1. Code to Return From Exception

```
      STNDW  B1:B0,*SP--           ; save B0 and B1
||    MVC    NTSR,B0               ; read NTSR
      EXTU   B0,16,30,B1           ; B1 = NTSR.IB and NTSR.SPLX
||    AND    B0,1,B0               ; B0 = NTSR.GIE
[B1]  MVK    0,B0                  ; B0 = 1 if resumable
[B0]  B      NRP                   ; if B0 != 0, return
      LDNDW  *SP++,B1:B0           ; restore B0 and B1
      NOP    4                     ; delay slots
cant_restart:
      ;code to handle non-resumable case starts here
```

6.3 Exception Detection and Processing

When an exception occurs, it sets a flag in the exception flag register (EFR). Depending on certain conditions, the exception may or may not be processed. This section discusses the mechanics of setting the flag bit, the conditions for processing an exception, and the order of operation for detecting and processing an exception.

6.3.1 *Setting the Exception Pending Flag*

Figure 6-2 shows the processing of an external exception (EXCEP) for the C64x+ CPU. The internal pending flag for EXCEP is set following the low-to-high transition of the EXCEP 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. Two clock cycles after detection, the EXCEP exception's corresponding flag bit (EXF) in EFR is set (cycle 6).

Figure 6-2 assumes EXCEP is enabled by the XEN, NMIE, and GEE bits, as necessary.

6.3.2 *Conditions for Processing an External Exception*

In clock cycle 4 of Figure 6-2, a nonreset exception in need of processing is detected. For this exception to be processed, the following conditions must be valid on the same clock cycle and are evaluated every clock cycle:

- EXF or NXF bit is set during CPU cycle 6. (This determination is made in CPU cycle 4 by the exception/interrupt logic.)
- NMIE = 1
- GEE = 1
- For EXCEP, XEN = 1

Any pending exception will be taken as soon as any stalls are completed.

When control is transferred to the interrupt processing sequence the context needed to return from the ISR is saved in ITSR. TSR is set for the default interrupt processing context. Table 6-3 shows the behavior for each bit in TSR. Figure 6-2 shows the timing of the changes to the TSR bits as well as the CPU outputs used in exception processing.

Fetches from program memory use the PS-valid register that is only loaded at the start of a context switch. This value is an output on the program memory interface and is shown in the timing diagram as PCXM. As the target execute packet progresses through the pipeline, the new mode is registered for that stage. Each stage uses its registered version of the execution mode. The field in TSR is the E1-valid version of CXM. It always indicates the execution mode for the instructions executing in E1. The mode is used in the data memory interface, and is registered for all load/store instructions when they execute in E1. This is shown in the timing diagram as DCXM.

Figure 6-3 shows the transitions in the case of a return from exception initiated by executing a **B NRP** instruction.

Figure 6-2. External Exception (EXCEP) Detection and Processing: Pipeline Operation

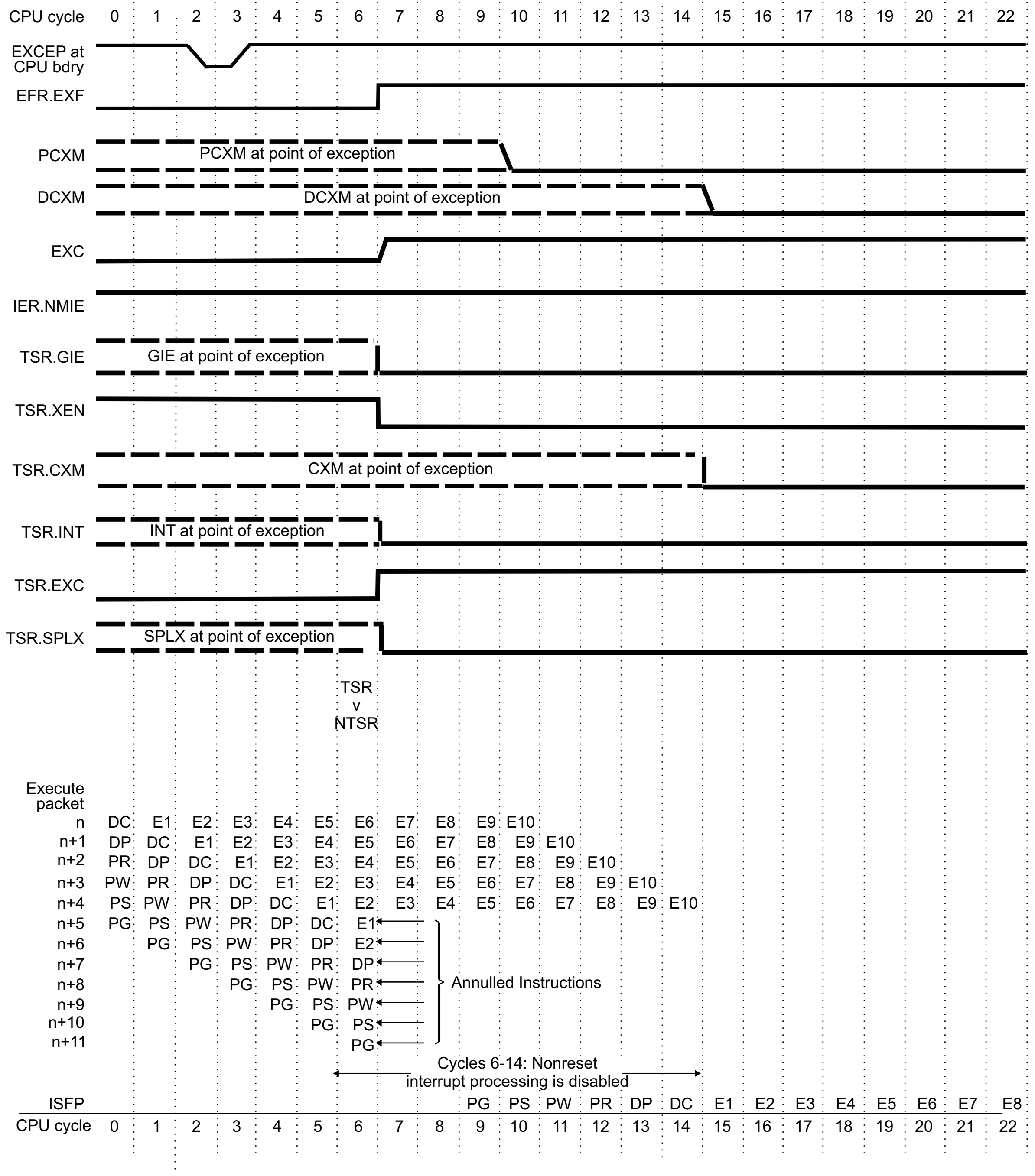
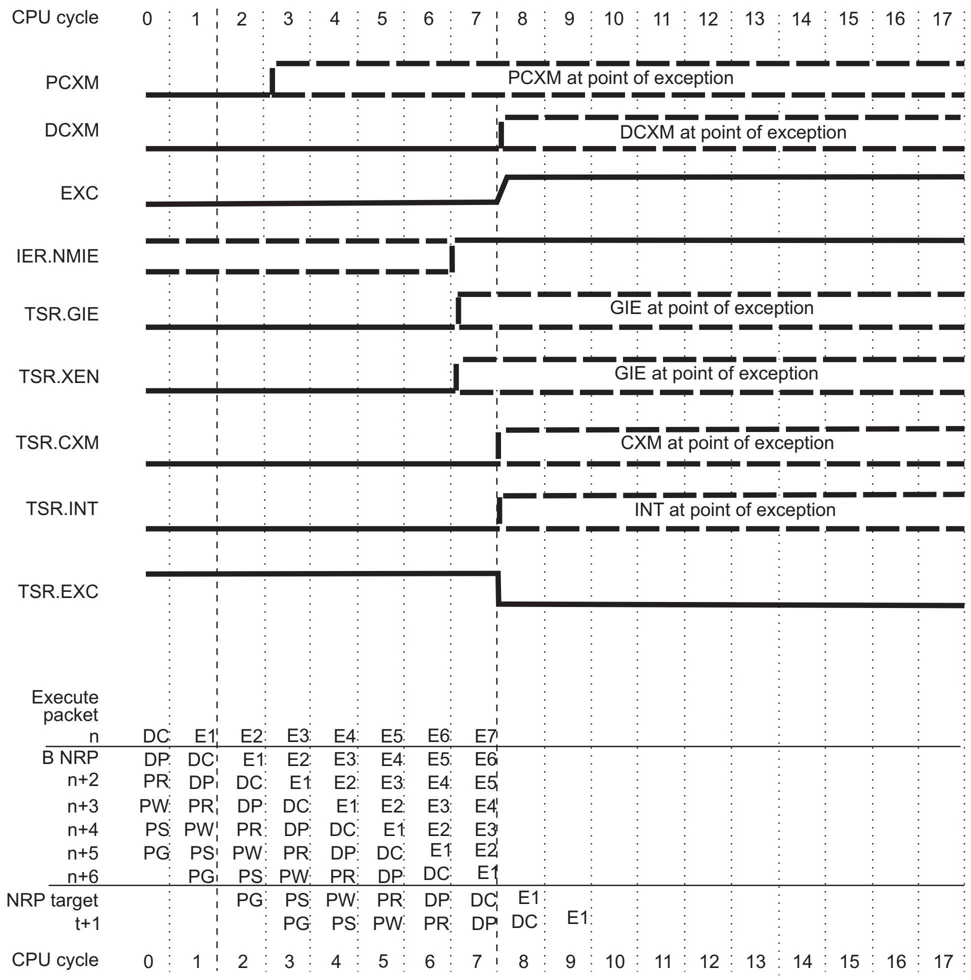

CPU cycle
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22
EXCEP at CPU bdry
EFR.EXF
PCXM
PCXM at point of exception
DCXM
DCXM at point of exception
EXC
IER.NMIE
TSR.GIE
GIE at point of exception
TSR.XEN
TSR.CXM
CXM at point of exception
TSR.INT
INT at point of exception
TSR.EXC
TSR.SPLX
SPLX at point of exception
TSR v NTSR
Execute packet
n DC E1 E2 E3 E4 E5 E6 E7 E8 E9 E10
n+1 DP DC E1 E2 E3 E4 E5 E6 E7 E8 E9 E10
n+2 PR DP DC E1 E2 E3 E4 E5 E6 E7 E8 E9 E10
n+3 PW PR DP DC E1 E2 E3 E4 E5 E6 E7 E8 E9 E10
n+4 PS PW PR DP DC E1 E2 E3 E4 E5 E6 E7 E8 E9 E10
n+5 PG PS PW PR DP DC E1
n+6 PG PS PW PR DP E2
n+7 PG PS PW PR DP
n+8 PG PS PW PR
n+9 PG PS PW
n+10 PG PS
n+11 PG
Annulled Instructions
Cycles 6-14: Nonreset interrupt processing is disabled
ISFP PG PS PW PR DP DC E1 E2 E3 E4 E5 E6 E7 E8
CPU cycle
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22

Table 6-3. TSR Field Behavior When an Exception is Taken (EXC = 0)

Bit	Field	Action
0	GIE	Saved to GIE bit in NTSR. Cleared to 0.
1	SGIE	Saved to SGIE bit in NTSR. Cleared to 0.
2	GEE	Saved to GEE bit in NTSR (will be 1). Unchanged.
3	XEN	Saved to XEN bit in NTSR. Cleared to 0.
7-6	CXM	Saved to CXM bits in NTSR. Set to Supervisor mode.
9	INT	Saved to INT bit in NTSR. Cleared to 0.
10	EXC	Saved to EXC bit in NTSR. Set to 1.
14	SPLX	Saved to SPLX bit in NTSR. Cleared to 0.
15	IB	Saved to IB bit in NTSR. Set by CPU control logic.

Figure 6-3. Return from Exception Processing: Pipeline Operation


CPU cycle
PCXM
PCXM at point of exception
DCXM
DCXM at point of exception
EXC
IER.NMIE
TSR.GIE
GIE at point of exception
TSR.XEN
GIE at point of exception
TSR.CXM
CXM at point of exception
TSR.INT
INT at point of exception
TSR.EXC
Execute packet

Execute packet	0	1	2	3	4	5	6	7	8	9
n	DC	E1	E2	E3	E4	E5	E6	E7		
B NRP	DP	DC	E1	E2	E3	E4	E5	E6		
n+2	PR	DP	DC	E1	E2	E3	E4	E5		
n+3	PW	PR	DP	DC	E1	E2	E3	E4		
n+4	PS	PW	PR	DP	DC	E1	E2	E3		
n+5	PG	PS	PW	PR	DP	DC	E1	E2		
n+6		PG	PS	PW	PR	DP	DC	E1		
NRP target			PG	PS	PW	PR	DP	DC	E1	
t+1				PG	PS	PW	PR	DP	DC	E1

6.3.3 Actions Taken During External Exception (EXCEP) Processing

During CPU cycles 6-14 of Figure 6-2, the following exception processing actions occur:

- Processing of subsequent EXCEP exceptions is disabled by clearing the XEN bit in TSR.
- Processing of interrupts is disabled by clearing the GIE bit in TSR.
- The next execute packets (from n + 5 on) are annulled. If an execute packet is annulled during a particular pipeline stage, it does not modify any CPU state. Annulling also forces an instruction to be annulled in future pipeline stages.
- The address of the first annulled execute packet (n + 5) is loaded in to NRP.
- A branch to the NMI ISFP is forced into the E1 phase of the pipeline during cycle 9.
- During cycle 7, IACK and EXC are asserted to indicate the exception is being processed. INUM is also valid in this cycle with a value of 1.

6.3.4 Nested Exceptions

When the CPU enters an exception service routine, the EXC bit in TSR is set to indicate an exception is being processed. If a new exception is recognized while this bit is set, then the reset vector is used when redirecting program execution to service the second exception. In this case, NTSR and NRP are left unchanged. TSR is copied to ITSR and the current PC is copied to IRP. TSR is set to the default exception processing value and the NMIE bit in IER is cleared in this case preventing any further external exceptions.

The NTSR, ITSR, IRP, and the NRP can be tested in the users boot code to determine if reset pin initiated reset or a reset caused by a nested exception.

6.4 Performance Considerations

6.4.1 General Performance

- **Overhead**. Overhead for all CPU exceptions on the C64x+ CPU is 9 cycles. You can see this in Figure 6-2, where no new instructions are entering the E1 pipeline phase during CPU cycles 6 through 14.
- **Latency**. Exception latency is 13 cycles. If the exception is active in cycle 2, execution of exception service code does not begin until cycle 15.
- **Frequency**. The pending exceptions are not automatically cleared upon servicing as is the case with interrupts.

6.4.2 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 exceptions exist. There are two operations or conditions that can affect, or are affected by, exceptions:

- **Memory stalls.** Memory stalls delay exception processing, because they inherently extend CPU cycles.
- **Multicycle NOPs.** Multicycle NOPs (including the **IDLE** instruction) operate like other instructions when interrupted by an exception, except when an exception 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. This prevents returning to an **IDLE** instruction or a multicycle **NOP** that was interrupted.

Figure 6-4. NMI Exception Detection and Processing: Pipeline Operation

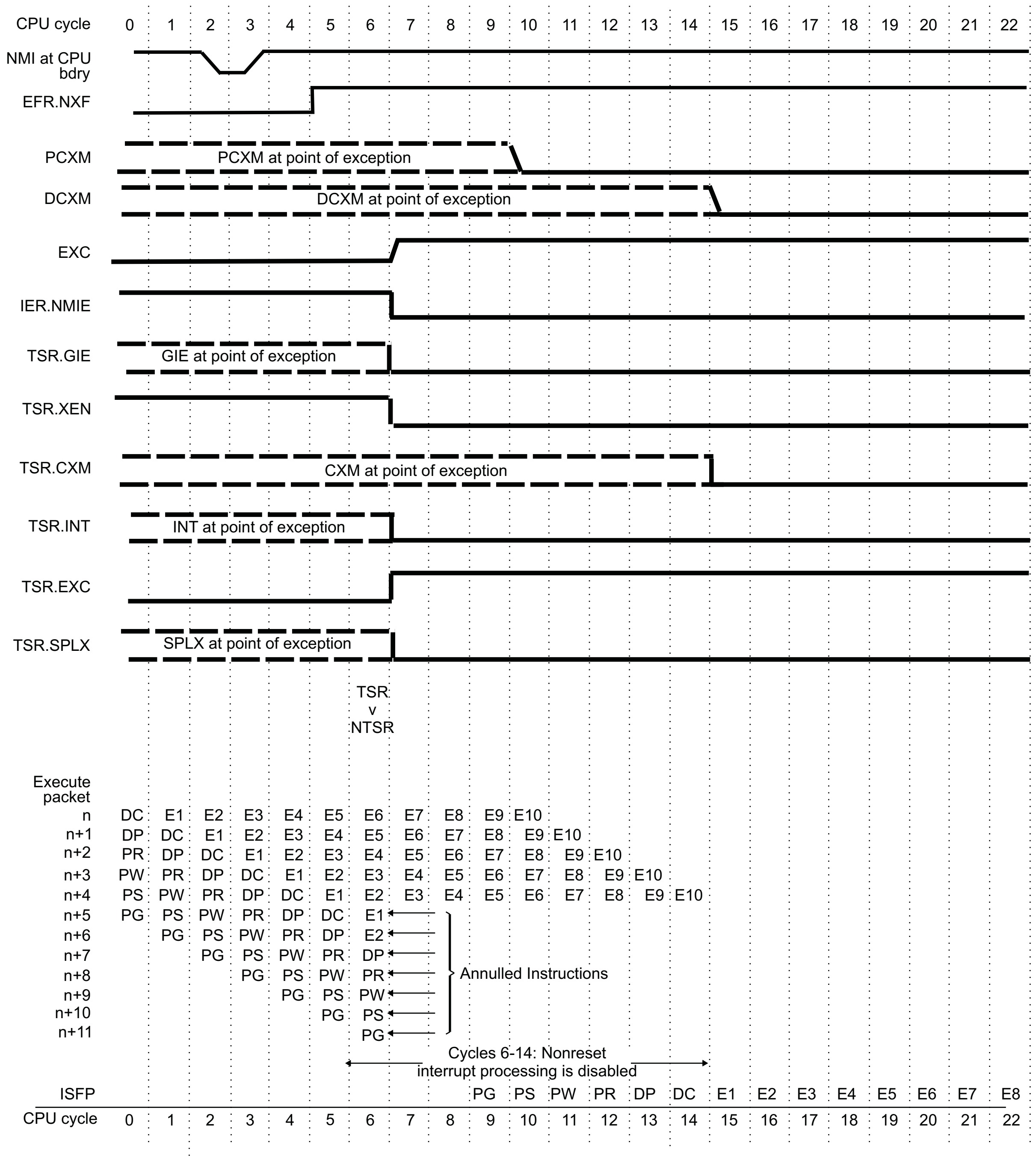
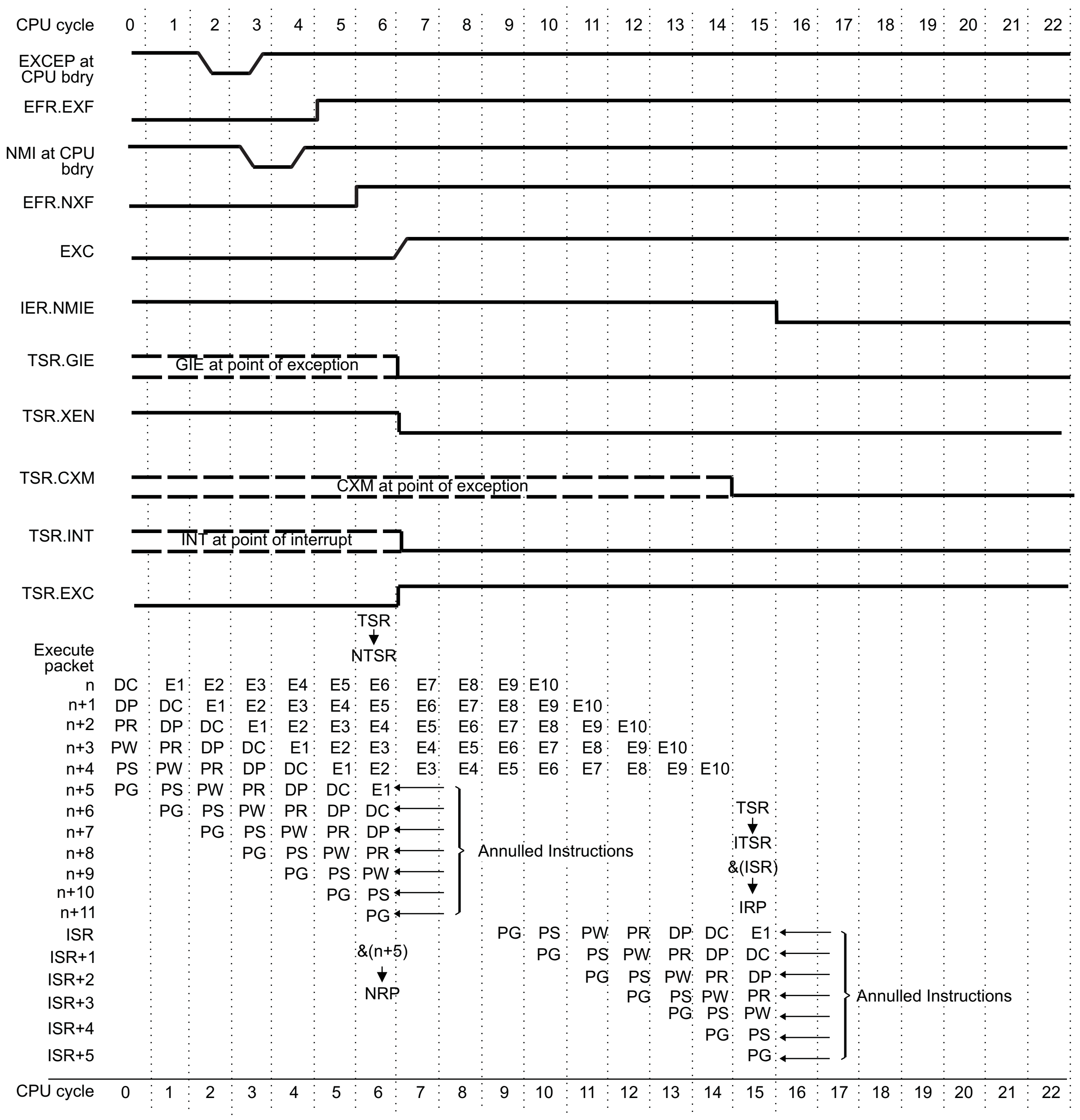

CPU cycle
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22
NMI at CPU bdry
EFR.NXF
PCXM
PCXM at point of exception
DCXM
DCXM at point of exception
EXC
IER.NMIE
TSR.GIE
GIE at point of exception
TSR.XEN
TSR.CXM
CXM at point of exception
TSR.INT
INT at point of exception
TSR.EXC
TSR.SPLX
SPLX at point of exception
TSR v NTSR
Execute packet
n DC E1 E2 E3 E4 E5 E6 E7 E8 E9 E10
n+1 DP DC E1 E2 E3 E4 E5 E6 E7 E8 E9 E10
n+2 PR DP DC E1 E2 E3 E4 E5 E6 E7 E8 E9 E10
n+3 PW PR DP DC E1 E2 E3 E4 E5 E6 E7 E8 E9 E10
n+4 PS PW PR DP DC E1 E2 E3 E4 E5 E6 E7 E8 E9 E10
n+5 PG PS PW PR DP DC E1
n+6 PG PS PW PR DP E2
n+7 PG PS PW PR DP
n+8 PG PS PW PR
n+9 PG PS PW
n+10 PG PS
n+11 PG
Annulled Instructions
Cycles 6-14: Nonreset interrupt processing is disabled
ISFP PG PS PW PR DP DC E1 E2 E3 E4 E5 E6 E7 E8
CPU cycle
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22

Figure 6-5. Double Exception Detection and Processing: Pipeline Operation



6.5 Programming Considerations

There are two types of exceptions that can result directly from instruction execution. The first is an intentional use via the **SWE** or **SWENR** instructions. The second is execution error detection exceptions that are internally generated within the CPU. These internal exceptions are primarily intended to facilitate program debug.

6.5.1 Internal Exceptions

Causes of internal exceptions:

- Fetch error
  - Program memory fetch error (privilege, parity, etc.)
    - Single input from L1P returned with data indicates error
  - Two branches taken in same execute packet
  - Branch to middle of 32-bit instruction in header-based fetch packet
  - Branch to header
- Illegal fetch packets
  - Reserved fetch packet header
- Illegal opcode
  - Specified set of reserved opcodes
  - Header not in word 7
- Privilege violation
  - Access to restricted control register
  - Attempt to execute restricted instruction
- Register write conflicts
- Loop buffer exceptions (**SPLOOP**, **SPKERNEL**)
  - Unit conflicts
  - Missed (but required) stall
  - Attempt to enter early-exit in reload while draining
  - Unexpected **SPKERNEL**
  - Write to ILC or RILC in prohibited timing window
  - Multicycle NOP prior to **SPKERNEL** or **SPKERNELR** instruction

6.5.2 Internal Exception Report Register (IERR)

The internal exception report register (IERR) contains flags that indicate the cause of the internal exception. In the case of simultaneous internal exceptions, the same flag may be set by different exception sources. In this case, it may not be possible to determine the exact causes of the individual exceptions. The IERR is shown in Figure 2-20 and described in Table 2-19.

6.5.3 Software Exception

6.5.3.1 SWE Instruction

When an **SWE** instruction is executed, the SXF bit in EFR is set. On the following cycle, the exception detection logic sees the SXF bit in EFR as a 1, and takes an exception. This instruction can be used to effect a system call while running in User mode. Execution of the instruction results in transfer of control to the exception service routine operating in Supervisor mode. If the SXF bit is the only bit set in EFR, then the exception can be interpreted as a system service request. An appropriate calling convention using the general-purpose registers may be adopted to provide parameters for the call. This is left as a programming choice. An example of code to quickly detect the system call case at the beginning of the exception service routine is shown in Example 6-2. Since other exceptions are in general error conditions and interrupt program execution at nonreturnable points, the need to process these is not particularly time critical.

Example 6-2. Code to Quickly Detect OS Service Request

```
      STW    B0,*SP--                 ; save B0
||    MVC    EFR,B0                   ; read EFR
      CMPEQ  B0,1,B0                  ; is SEF the only exception?
[B0]  B      OS_Service               ; if so,
[B0]  ...                             ; conditionally execute service
[B0]  ...                             ; code until branch takes effect
```

6.5.3.2 SWENR Instruction

The **SWENR** instruction causes a software exception to be taken similarly to that caused by the **SWE** instruction. It is intended for use in systems supporting a secure operating mode. The **SWENR** instruction can be used as a mechanism for nonsecure programs to return to secure Supervisor mode services such as an interrupt dispatcher. It differs from the **SWE** instruction in four ways:

1. TSR is not copied into NTSR
2. No return address is placed in NRP (it stays unmodified)
3. A branch to restricted entry point control register (REP) is forced in the context switch rather than the ISTP-based exception (NMI) register.
4. The IB bit in TSR is set to 1. This is observable only in the case where another exception is recognized simultaneously.

If another exception (internal or external) is recognized simultaneously with the **SWENR**-raised exception, then the other exceptions(s) take priority and normal exception behavior occurs; that is, NTSR and NRP are used, execution is directed to the NMI vector. In this case, the setting of the IB bit in TSR by the **SWENR** instruction is registered in NTSR. Assuming the **SWE** or **SWENR** instruction was not placed in an execute slot where interrupts are architecturally blocked (as should always be the case), then the IB bit in NTSR will differentiate whether the simultaneous exception occurred with **SWE** or **SWENR**.

The **SWENR** instruction causes a change in control to the address contained in REP. It should have been previously initialized to a correct value by a privileged supervisor mode process.

TEXAS INSTRUMENTS

Chapter 7
SPRU732J–July 2010

Software Pipelined Loop (SPLOOP) Buffer

This chapter describes the software pipelined loop (SPLOOP) buffer hardware and software mechanisms.

The C64x CPU does not support software pipelined loops (SPLOOP). This chapter applies only to the C64x+ CPU.

Under normal circumstances, the compiler/assembly optimizer will do a good job coding SPLOOPs and it will not be necessary for the programmer to hand code usage of the SPLOOP buffer. This chapter is intended to describe the functioning of the buffer hardware and the instructions that control it.

Topic **Page**

7.1 Software Pipelining

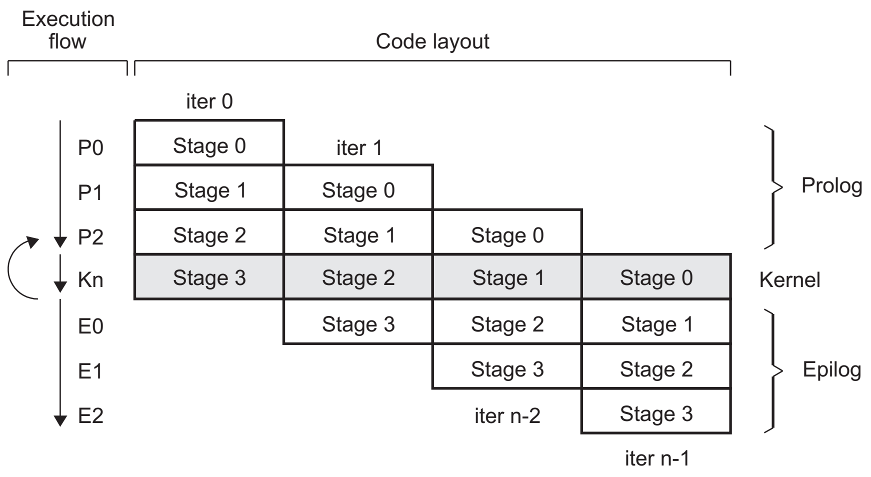
Software pipelining is a type of instruction scheduling that exploits instruction level parallelism (ILP) across loop iterations. Modulo scheduling is a form of software pipelining that initiates loop iterations at a constant rate, called the initiation interval (ii). To construct a modulo scheduled loop, a single loop iteration is divided into a sequence of stages, each with length ii. In the steady state of the execution of the software pipelined loop, each of the stages is executing in parallel.

The instruction schedule for a modulo scheduled loop has three components: a kernel, a prolog, and an epilog (Figure 7-1). The kernel is the instruction schedule that executes the pipeline steady state. The prolog and epilog are the instruction schedules that setup and drain the execution of the loop kernel. In Figure 7-1, the steady state has four stages, each from a different iteration, executing in parallel. A single iteration produces a result in the time it takes four stages to complete, but in the steady state of the software pipeline, a result is available every stage (that is, every ii cycles).

The first prolog stage, P0, is equal to the first loop stage, S0. Each prolog stage, Pn (where n > 0), is made up of the loop stage, Sn, plus all the loop stages in the previous prolog stage, Pn - 1. The kernel includes all the loop stages. The first epilog stage, E0, is made up of the kernel stage minus the first loop stage, S0. Each epilog stage, En (where n > 0), is made up of the previous epilog stage, En - 1, minus the loop stage, Sn.

The dynamic length (dynlen) of the loop is the number of instruction cycles required for one iteration of the loop to complete. The length of the prolog is (dynlen − ii). The length of the epilog is the same as the length of the prolog.

Figure 7-1. Software Pipelined Execution Flow



7.2 Software Pipelining

The SPLOOP facility on the C64x+ DSP stores a single iteration of loop in a specialized buffer and contains hardware that will selectively overlay copies of the single iteration in a software pipeline manner to construct an optimized execution of the loop.

This provides the following benefits.

- Since the prolog and epilog do not need to be explicitly code, code size is significantly reduced.
- The SPLOOP version of the loop can be easily interrupted unlike the non- SPLOOP version of the same loop.
- Since the instructions in the loop do not need to be fetched on each cycle, the memory bandwidth and power requirements are reduced.
- Since the loop executes out of a buffer, the branch to the start of loop is implicit (hence not required). In some cases this may permit a tighter loop since a .S unit is freed.

7.3 Terminology

The following terminology is used in the discussion in this chapter.

- Iteration interval (ii) is the interval (in instruction cycles) between successive iterations of the loop.
- A stage is the code executed in one iteration interval.
- Dynamic length (dynlen) is the length (in instruction cycles) of a single iteration of the loop. It is therefore equal to the number of stages times the iteration interval.5
- The kernel is the period when the loop is executing in a steady state with the maximum number of loop iterations executing simultaneously. For example: in Figure 7-1 the kernel is the set of instructions contained in stage 0, stage 1, stage, 2, and stage 3.
- The prolog is the period before the loop reaches the kernel in which the loop is winding up. The length of the prolog will by the dynamic length minus the iteration interval (dynlen - ii).
- The epilog is the period after the loop leaves the kernel in which the loop is winding down. The length of the prolog will by the dynamic length minus the iteration interval (dynlen - ii).

7.4 SPLOOP Hardware Support

The basic hardware support for the SPLOOP operation is:

- Loop buffer
- Loop buffer count register (LBC)
- Inner loop count register (ILC)
- Reload inner loop count register (RILC)
- Task state register (TSR)
- Interrupt task state register (ITSR)
- NMI/Exception task state register (NTSR)

7.4.1 Loop Buffer

The loop buffer is used to store the instructions that comprise the loop and information describing the sequence that the instructions were added to the buffer and the state (active or inactive) of each instruction.

The loop buffer has enough storage for up to 14 execute packets.

7.4.2 Loop Buffer Count Register (LBC)

A loop buffer count register (LBC) is maintained as an index into the loop buffer. It is cleared to 0 when an **SPLOOP, SPLOOPD** or **SPLOOPW** instruction is encountered and is incremented by 1 at the end of each cycle. When LBC becomes equal to the iteration interval (ii) specified by the **SPLOOP, SPLOOPD** or **SPLOOPW** instruction, then a stage boundary has been reached and LBC is reset to 0 and the inner loop count register (ILC) is decremented.

There are two LBCs to support overlapped nested loops. LBC is not a user-visible register.

7.4.3 Inner Loop Count Register (ILC)

The inner loop count register (ILC) is used as a down counter to determine when the SPLOOP is complete when the SPLOOP is initiated by either a **SPLOOP** or **SPLOOPD** instruction. When the loop is initiated using a **SPLOOPW** instruction, the ILC is not used to determine when the SPLOOP is complete. It is decremented once each time a stage boundary is encountered; that is, whenever the loop buffer count register (LBC) becomes equal to the iteration interval (ii).

There is a 4 cycle latency between when ILC is loaded and when its contents are available for use. When used with the **SPLOOP** instruction, it should be loaded 4 cycles before the **SPLOOP** instruction is encountered. ILC must be loaded explicitly using the **MVC** instruction.

7.4.4 Reload Inner Loop Count Register (RILC)

The reload inner loop count register (RILC) is used for resetting the inner loop count register (ILC) for the next invocation of a nested inner loop. There is a 4 cycle latency between when RILC is loaded with the **MVC** instructions and when the value loaded to RILC is available for use. RILC must be loaded explicitly using the **MVC** instruction.

7.4.5 Task State Register (TSR), Interrupt Task State Register (ITSR), and NMI/Exception Task State Register (NTSR)

The SPLX bit in the task state register (TSR) indicates whether an SPLOOP is currently executing or not executing.

When an interrupt occurs, the contents of TSR (including the SPLX bit) is copied to the interrupt task state register (ITSR).

When an exception or non-maskable interrupt occurs, the contents of TSR (including the SPLX bit) is copied to the NMI/Exception task state register (NTSR).

See Section 2.9.15 for more information on TSR. See Section 2.9.9 for more information on ITSR. See Section 2.9.10 for more information on NTSR.

7.5 SPLOOP-Related Instructions

The following instructions are used to control the operation of an SPLOOP:

- SPLOOP, SPLOOPD, and SPLOOPW
- SPKERNEL and SPKERNELR
- SPMASK and SPMASKR

7.5.1 SPLOOP, SPLOOPD, and SPLOOPW Instructions

One of the **SPLOOP**, **SPLOOPD**, or **SPLOOPW** (collectively called **SPLOOP(D/W)**) instructions are used to invoke the loop buffer mechanism. They each fulfil the same basic purpose, but differ in details. In each case, they must be the first instruction of the execute packet containing it. They cannot be placed in the same execute packet as any instruction that initiates a multicycle NOP (for example: **BNOP** or **NOP** *n*).

When you know in advance the number of iterations that the loop will execute, you can use the **SPLOOP** or **SPLOOPD** instructions. If you do not know the exact number of iterations that the loop should execute, you can use the **SPLOOPW** in a fashion similar to a do−while loop.

The **SPLOOP(D/W)** instructions each clear the loop buffer count register (LBC), load the iteration interval (ii), and start the LBC counting.

7.5.1.1 SPLOOP Instruction

The **SPLOOP** instruction is coded as:

```
[cond] SPLOOP ii
```

The ii parameter is the iteration interval which specifies the interval (in instruction cycles) between successive iterations of the loop.

The **SPLOOP** instruction is used when the number of loop iterations is known in advance. The number of loop iterations is determined by the value loaded to the inner loop count register (ILC). ILC should be loaded with an initial value 4 cycles before the **SPLOOP** instruction is encountered.

The (optional) conditional predication is used to indicate when and if a nested loop should be reloaded. The contents of the reload inner loop counter (RILC) is copied to ILC when either a **SPKERNELR** or a **SPMASKR** instruction is executed with the predication condition on the **SPLOOP** instruction true. If the loop is not nested, then the conditional predication should not be used.

7.5.1.2 SPLOOPD Instruction

The **SPLOOPD** instruction is coded as:

```
[cond] SPLOOPD ii
```

The ii parameter is the iteration interval which specifies the interval (in instruction cycles) between successive iterations of the loop.

The **SPLOOPD** instruction is used to initiate a loop buffer operation when the known minimum iteration count of the loop is great enough that the inner loop count register (ILC) can be loaded in parallel with the **SPLOOPD** instruction and the 4 cycle latency will have passed before the last iteration of the loop.

Unlike the **SPLOOP** instruction, the load of ILC is performed in parallel with the **SPLOOPD** instruction. Due to the inherent latency of the load to ILC, the value to ILC should be predecremented to account for the 4 cycle latency. The amount of the predecrement is given in Table 7-4.

The number of loop iterations is determined by the value loaded to ILC.

The (optional) conditional predication is used to indicate when and if a nested loop should be reloaded. The contents of the reload inner loop counter (RILC) is copied to ILC when either a **SPKERNELR** or a **SPMASKR** instruction is executed with the predication condition on the **SPLOOP** instruction true. If the loop is not nested, then the conditional predication should not be used.

The use of the **SPLOOPD** instruction can result in reducing the time spent in setting up the loop by eliminating up to 4 cycles that would otherwise be spent in setting up ILC. The tradeoff is that the **SPLOOPD** instruction cannot be used if the loop is not long enough to accommodate the 4 cycle delay.

7.5.1.3 SPLOOPW Instruction

The **SPLOOPW** instruction is coded as:

```
[cond] SPLOOPW ii
```

The ii parameter is the iteration interval which specifies the interval (in instruction cycles) between successive iterations of the loop.

The **SPLOOPW** instruction is used to initiate a loop buffer operation when the total number of loops required is not known in advance. The **SPLOOPW** instruction must be predicated. The loop terminates if the predication condition is not true. The value in the inner loop count register (ILC) is not used to determine the number of loops.

Unlike the **SPLOOP** and **SPLOOPD** instructions, predication on the **SPLOOPW** instruction does not imply a nested SPLOOP operation. The **SPLOOPW** instruction cannot be used in a nested SPLOOP operation.

When using the **SPLOOPW** instruction, the predication condition is used to determine the exit condition for the loop. The ILC is not used for this purpose when using the **SPLOOPW** instruction.

When the **SPLOOPW** instruction is used to initiate a loop buffer operation, the epilog is skipped when the loop terminates.

7.5.2 SPKERNEL and SPKERNELR Instructions

The **SPKERNEL** or the **SPKERNELR** (collectively called **SPKERNEL(R)**) instruction is used to mark the end of the software pipelined loop. The **SPKERNEL(R)** instruction is placed in parallel with the last execute packet of the SPLOOP code body indicating that there are no more instructions to load to the loop buffer.

The **SPKERNEL(R)** instruction also controls the point in the epilog that the execution of post-SPLOOP instructions begin.

In each case, the **SPKERNEL(R)** instruction must be the first instruction in an execute packet and cannot be placed in the same execute packet as any instruction that initiates multicycle NOPs.

7.5.2.1 SPKERNEL Instruction

The **SPKERNEL** instruction is coded as:

```
SPKERNEL (fstg, fcyc)
```

The (optional) *fstg* and *fcyc* parameters specify the delay interval between the **SPKERNEL** instruction and the start of the post epilog code. The *fstg* specifies the number of complete stages and the *fcyc* specifies the number of cycles in the last stage in the delay.

The **SPKERNEL** instruction has arguments that instruct the SPLOOP hardware to begin execution of post-SPLOOP instructions by an amount of delay (stages/cycles) after the start of the epilog.

Note that the post-epilog instructions are fetched from program memory and overlaid with the epilog instructions fetched from the SPLOOP buffer. Functional unit conflicts can be avoided by either coding for a sufficient delay using the **SPKERNEL** instruction arguments or by using the **SPMASK** instruction to inhibit the operation of instructions from the buffer that might conflict with the instructions from the epilog.

7.5.2.2 SPKERNELR Instruction

The **SPKERNELR** instruction is coded as:

```
SPKERNELR
```

The **SPKERNELR** instruction is used to support nested SPLOOP execution where a loop needs to be restarted with perfect overlap of the prolog of the second loop with the epilog of the first loop.

If a reload is required with a delay between the **SPKERNEL** and the point of reload (that is, nonperfect overlap) use the **SPMASKR** instruction with the **SPKERNEL** (not **SPKERNELR**) to indicate the point of reload.

The **SPKERNELR** instruction has no arguments. The execution of post-SPLOOP instructions commences simultaneous with the first cycle of epilog. If the predication of the **SPLOOP** instruction indicates that the loop is being reloaded, the instructions are fetched from both the SPLOOP buffer and of program memory.

The **SPKERNELR** instruction cannot be used in the same SPLOOP operation as the **SPMASKR** instruction.

7.5.3 SPMASK and SPMASKR Instructions

The **SPMASK** and **SPMASKR** (collectively called **SPMASK(R)**) instructions are used to inhibit the operation of instructions on specified functional units within the current execute packet.

- If there is an instruction from the buffer that would utilize the specified functional unit in the current cycle, the execution of that instruction is inhibited.
- If the buffer is in the loading stage and there is an instruction (regardless of functional unit) that is scheduled for execution during that cycle, the execution of that instruction proceeds, but the instruction is not loaded into the buffer.
- If the case where an **SPMASK(R)** instruction is encountered while the loop is resuming after returning from an interrupt, the **SPMASK(R)** instruction causes the instructions coming from the buffer to execute, but instructions coming from program memory to be inhibited and are not loaded to the buffer.

The **SPMASKR** instruction is identical to the function of the **SPMASK** instruction with one additional operation. In the case of nested loops where it is not desired that the reload of the buffer happen immediately after the **SPKERNEL** instruction, the **SPMASKR** instruction can be used to mark the point in the epilog that the reload should begin.

The **SPMASKR** instruction cannot be used in the same SPLOOP operation as the **SPKERNELR** instruction.

The **SPMASK** and **SPMASKR** instructions are coded as:

```
SPMASK (unitmask)
SPMASKR (unitmask)
```

The unitmask parameter specifies which functional units are masked by the **SPMASK** or **SPMASKR** instruction. The units may alternatively be specified by marking the instructions with a caret (^) symbol. The following two forms are equivalent and will each mask the .D1 unit. Example 7-1 and Example 7-2 show the two ways of specifying the masked instructions.

Example 7-1. SPMASK Using Unit Mask to Indicate Masked Unit

```
    SPMASK   D1                ;Mask .D1 unit
||  LDW     .D1   *A0,A1       ;This instruction is masked
||  MV      .L1   A2,A3        ;This instruction is NOT masked
```

Example 7-2. SPMASK Using Caret to Indicate Masked Unit

```
    SPMASK                     ;Mask .D1 unit
||^ LDW     .D1   *A0,A1       ;This instruction is masked
||  MV      .L1   A2,A3        ;This instruction is NOT masked
```

7.6 Basic SPLOOP Example

This section discusses a simple SPLOOP example. Example 7-3 shows an example of a loop coded in C and Example 7-4 shows an implementation of the same loop using the **SPLOOP** instruction. The loop copies a number of words from one location in memory to another.

Example 7-3. Copy Loop Coded as C Fragment

```
     for (I=0; i<val; I++)  {
                  dest[i]=source[i];
                  }
```

Example 7-4. SPLOOP Implementation of Copy Loop

```
    MVC        .S2   8,ILC          ;Do 8 loops
    NOP        3                    ;4 cycle for ILC to load
    SPLOOP     1                    ;Iteration interval is 1
    LDW        *A1++,A2             ;Load source
    NOP        4                    ;Wait for source to load
    MV         .L2X  A2,B2          ;Position data for write
    SPKERNEL   6,0                  ;End loop and store value
||  STW        B2,*B0++
```

Example 7-5 is an alternate implementation of the same loop using the **SPLOOPD** instruction. The load of the inner loop count register (ILC) can be made in the same cycle as the **SPLOOPD** instruction, but due to the inherent delay between loading the ILC and its use, the value needs to be predecremented to account for the 4 cycle delay.

Example 7-5. SPLOOPD Implementation of Copy Loop

```
    SPLOOPD    1                    ;Iteration interval is 1
||  MVC        .S2   8-4,ILC        ;Do 8 iterations
    LDW        *A1++,A2             ;Load source
    NOP        4                    ;Wait for source to load
    MV         .L1X   A2,B2         ;Position data for write
    SPKERNEL   6,0                  ;End loop and store value
||  STW        B2,*B0++
```

Table 7-1. SPLOOP Instruction Flow for Example 7-4 and Example 7-5

	Loop							
Cycle	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
1	LDW							
2	NOP	LDW						
3	NOP	NOP	LDW					
4	NOP	NOP	NOP	LDW				
5	NOP	NOP	NOP	NOP	LDW			
6	MV	NOP	NOP	NOP	NOP	LDW		
7	STW	MV	NOP	NOP	NOP	NOP	LDW	
8		STW	MV	NOP	NOP	NOP	NOP	LDW
9			STW	MV	NOP	NOP	NOP	NOP
10				STW	MV	NOP	NOP	NOP
11					STW	MV	NOP	NOP
12						STW	MV	NOP
13							STW	MV
14								STW

7.6.1 Some Points About the Basic SPLOOP Example

Note the following points about Example 7-4, Example 7-5, and Table 7-1.

- In Example 7-4, due to the 4 cycle latency of loading ILC, the load to ILC happens at least 4 cycles before the **SPLOOP** instruction is encountered. In this case, the **MVC** instruction that loads ILC is followed by 3 cycles of NOP.
- In Example 7-5, the use of the **SPLOOPD** instruction allows you to load ILC in the same cycle of the **SPLOOPD** instruction; but the value loaded to ILC is adjusted to account for the inherent 4 cycle delay between loading the ILC and its use.
- The iteration interval (ii) is specified in the argument of the SPLOOP instruction.
- The termination condition (ILC equal to 0) is tested at every stage boundary. In this example, with ii equal to 1, it is tested at each instruction cycle. Once the termination condition is true, the loop starts draining.
- Cycles 1 through 6 constitute the prolog. Until cycle 7, the pipeline is filling.
- Cycles 7 and 8 each constitute a single iteration of the kernel. During each of these cycles, the pipeline is filled as full as it is going to be.
- Cycles 9 through 14 constitute the epilog. During this interval, the pipeline is draining.
- In this example, the iteration interval is 1. A new iteration of the loop is started each cycle.
- The dynamic length (dynlen) is 7. One cycle for the **LDW** instruction. One cycle for the **MV** instruction. One instruction for the **SPKERNEL** and **STW** instructions executed in parallel. Four cycles of NOP.
- The length of the prolog is (dynlen - ii) = 6 cycles. The length of the epilog is equal to the length of the prolog.
- There is no branch back to the start of the loop. The instructions are executed from the SPLOOP buffer and the **SPKERNEL** instruction marks the point that the execution is reset to the start of the buffer.
- The body of the SPLOOP is a single scheduled iteration without pipeline optimization. The execution of the SPLOOP overlays the execution of the instructions to optimize the execution of the loop.
- The argument in the **SPKERNEL** instruction indicates that the post-epilog code is delayed until after the epilog (6 cycles) completes.
- The **MV** instruction needs to be there to move the data between the A side and the B side. If it were not there, there would eventually be a unit conflict between the **LDW** and **STW** instructions as they try to execute in the same cycle.

7.6.2 Same Example Using the SPLOOPW Instruction

For completeness, Example 7-6 shows an example of a loop coded in C and Example 7-7 is the same example using the **SPLOOPW** instruction. The **SPLOOPW** instruction is intended to support while−loop constructions in which the number of loops is not known in advance and (in general) is determined as the loop progresses.

Example 7-6. Example C Coded Loop

```
    do         {
                  I--;
                  dest[i]=source[i];
                  } while (I);
```

Example 7-7. SPLOOPW Implementation of C Coded Loop

```
        MVK        8,A0                 ;Do 8 loops
    [!A0]   SPLOOPW    1                    ;Check loop
        LDW        .D1   *A1++,A2       ;Load source value
        NOP        1
        SUB        .S1   A0,1,A0        ;Adjust loop counter
        NOP        2                    ;Wait for source to load
        MV         .L2X  A2,B2          ;Position data for write
        SPKERNEL   0,0                  ;End loop
||      STW        .D2   B2,*B0++       ;Store value
```

Table 7-2. SPLOOPW Instruction Flow for Example 7-7

	Loop											
Cycle	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
1	LDW											
2	NOP	LDW										
3	SUB	NOP	LDW									
4	NOP	SUB	NOP	LDW								
5	NOP	NOP	SUB	NOP	LDW							
6	MV	NOP	NOP	SUB	NOP	LDW						
7	STW	MV	NOP	NOP	SUB	NOP	LDW					
8		STW	MV	NOP	NOP	SUB	NOP	LDW				
9			STW	MV	NOP	NOP	SUB	NOP	LDW			
10				STW	MV	NOP	NOP	SUB	NOP	LDW		
11					STW	MV	NOP	NOP	SUB	NOP	LDW	
12						STW	MV	NOP	NOP	SUB	NOP	LDW

7.6.3 Some Points About the SPLOOPW Example

Note the following points about Example 7-7 and Table 7-2.

- Unlike the **SPLOOP** and **SPLOOPD** instructions, the number of loops does not depend on ILC. It depends, instead, on the value in the predication register used with the **SPLOOPW** instruction (in this case, the value in A0).
- The termination condition (A0 equal to 0) is tested at every stage boundary. In this example, with ii equal to 1, it is tested at each instruction cycle. Once the termination condition is true, the loop terminates abruptly without executing the epilog.
- The termination condition (A0 equal to 0) needs to be true 3 cycles before you actually test it, so the value of A0 needs to be adjusted 4 cycles before the **SPKERNEL** instruction. In this case, the SUB instruction was positioned using the **NOP** instructions so that its result would be available 3 cycles before the **SPKERNEL**.
- Unlike the **SPLOOP** and **SPLOOPD** instructions, the **SPLOOPW** instruction causes the loop to exit without an epilog. In cycle 13, the loop terminates abruptly.
- Loop 9 through loop 14 begin, but they do not finish. The loop exits abruptly on the stage boundary 3 cycles after the termination condition becomes true. It is important that the loop is coded so that the extra iterations of the early instructions do not cause problems by overwriting significant locations. For example: if the loop contains an early write to a buffer we might find that in incorrectly coded loop might write beyond the end of the buffer overwriting data unintentionally.

7.7 Loop Buffer

The basic facility to support software pipelined loops is the loop buffer. The loop buffer has storage for up to 14 execute packets. The buffer is filled from the SPLOOP body that follows an **SPLOOP(D/W)** instruction and ends with the first execute packet containing an **SPKERNEL** instruction.

The SPLOOP body is a single, scheduled iteration of the loop. It consists of one or more stages of ii cycles each. The execution of the prolog, kernel, and epilog are generated from copies of this single iteration time shifted by multiples of ii cycles and overlapped for simultaneous execution. The final stage may contain fewer than ii cycles, omitting the final cycles if they have only **NOP** instructions.

The dynamic length (dynlen) is the length of the SPLOOP body in cycles starting with the cycle after the **SPLOOP(D)** instruction. The dynamic length counts both execute packets and NOP cycles, but does not count stall cycles. The loop buffer can accommodate a SPLOOP body of up to 48 cycles.

Example 7-8 demonstrates counting of dynamic length. There are 4 cycles of NOP that could be combined into a single **NOP 4**. It is split up here to be clearer about the cycle and stage boundaries.

Example 7-8. SPLOOP, SPLOOP Body, and SPKERNEL

```
        SPLOOP     2                      ;ii=2, dynlen=7
||      MV         A0,A1                  ;save previous cond reg
;-----Start of stage 0
        LDW        *B7[A0],A5             ;cycle 1
        NOP                               ;cycle 2
;-----stage boundary. End of stage 0, start of stage 1
        NOP        2                      ;cycles 3 and 4
;-----stage boundary. End of stage 1, start of stage 2
        NOP                               ;cycle 5
        EXTU       A5,12,7,A6             ;cycle 6
;-----stage boundary. End of stage 2, start of stage 3
        SPKERNEL   0,0                    ;last exe pkt of SPLOOOP body
||      ADD        .D1  A6,A7,A7          ;accumulate (cycle 7)
;       NOP                               ;can omit final NOP of last stage
;-----stage boundary. End of stage 3
```

7.7.1 Software Pipeline Execution From the Loop Buffer

The loop buffer is the mechanism that both generates the execution of the loop prolog, kernel, and epilog, and saves the repeated fetching and decoding of instructions in the loop. As the SPLOOP body is fetched and executed the first time, it is loaded into the loop buffer. By the time the entire SPLOOP body has been loaded into the loop buffer, the loop kernel is present in the loop buffer and the execution of the loop kernel can be entirely from the loop buffer. The last portion of the software pipeline is the epilog; which is generated by removing instructions from the buffer in the order that they were loaded into it.

In Table 7-3, the instructions in the CPU pipeline are executed from program memory. The instructions in the SPL buffer are executed from the SPLOOP buffer. At K0 for example, stage3 is being executed from program memory and stage0, stage1, and stage2 are being executed from the SPLOOP buffer. At Kn and later, by contrast, all stages are being executed from the SPLOOP buffer.

Table 7-3. Software Pipeline Instruction Flow Using the Loop Buffer

Execution Flow	CPU Pipeline	SPL Buffer			
P0	stage0	-			
P1	stage1	stage0			
P2	stage2	stage0	stage1		
K0	stage3	stage0	stage1	stage2	
Kn	-	stage0	stage1	stage2	stage3
E0	-	-	stage1	stage2	stage3
E1	-	-	-	stage2	stage3
E2	-	-	-	-	stage3

7.7.2 Stage Boundary Terminology

A stage boundary is reached every ii cycles. The following terminology is used to describe specific stage boundaries.

- **First loading stage boundary:** The first stage boundary after the **SPLOOP(D/W)** instruction. The stage boundary at the end of P0 in Table 7-3.
- **Last loading stage boundary:** The first stage boundary that occurs in parallel with or after the **SPKERNEL** instruction. The stage boundary at the end of K0 in Table 7-3. This is the same as the first kernel stage boundary.
- **First kernel stage boundary:** The same as the last loading stage boundary.
- **Last kernel stage boundary:** The last stage boundary before the loop is only executing epilog instructions. The stage boundary at the end of Kn in Table 7-3.

7.7.3 Loop Buffer Operation

On the cycle after an **SPLOOP(D/W)** instruction is encountered, instructions are loaded into the loop buffer. A loop buffer count register (LBC) is maintained as an index into the loop buffer. At the end of each cycle, LBC is incremented by 1. If LBC becomes equal to the ii, then a stage boundary has been reached and LBC is reset to 0. There are two LBCs to support overlapped nested loops.

The loop buffer has four basic operations:

- **Load:** instructions fetched from program memory are executed and written into the loop buffer at the current LBC and marked as valid on the next cycle
- **Fetch:** valid instructions are fetched from the loop buffer at the current LBC and executed in parallel with any instructions fetched from program memory.
- **Drain:** instructions at the current LBC are marked as invalid on the current cycle and not executed.
- **Reload:** instructions at the current LBC are marked as valid on the current cycle and executed.

The execution of a software pipeline prolog and the first kernel stage are implemented by fetching valid instructions from the loop buffer and executing them in parallel with instructions fetched from program memory. The instructions fetched from program memory are loaded into the loop buffer and marked as valid on the next cycle. The execution of the remaining kernel stages is implemented by exclusively fetching valid instructions from the loop buffer. The execution of a software pipeline epilog is implemented by draining the loop buffer by marking instructions as invalid, while fetching the remaining valid instructions from the loop buffer.

For example: referring to Example 7-4 and Table 7-1; as each instruction in loop 1 is reached in turn, it is fetched from program memory, executed and stored in the loop buffer. When each instruction is reached in loop 2 through loop 12, it is fetched from the loop buffer and executed. As cycles 8 through 12 execute, instructions in the loop buffer are marked as invalid so that for each cycle fewer instructions are fetched from the loop buffer.

The loop buffer supports the execution of a nested software pipelined loop by reenabling the instructions stored in the loop buffer. The reexecution of the software pipeline prolog is implemented by reenabling instructions in the loop buffer (by marking them as valid) and then fetching valid instructions from the loop buffer. The point of reload for the nested loop is signaled by the **SPKERNELR** or **SPMASKR** instruction.

The loop buffer also supports do−while type of constructs in which the number of iterations is not known in advance, but is determined in the course of the execution of the loop. In this case, the loop immediately completes after the last kernel stage without executing the epilog.

7.7.3.1 Interrupt During SPLOOP Operation

If an interrupt occurs while a software pipeline is executing out of the loop buffer, the loop will pipe down by executing an epilog and then service the interrupt. The interrupt return address stored in the interrupt return pointer register (IRP) or the nonmaskable interrupt return pointer register (NRP) is the address of the execute packet containing the **SPLOOP** instruction. The task state register (TSR) is copied into the interrupt task state register (ITSR) or the NMI/exception task state register (NTSR) with the SPLX bit set to 1. On return from the interrupt with ITSR or NTSR copied back into TSR and the SPLX bit set to 1, execution is resumed at the address of the **SPLOOP(D/W)** instruction, and the loop is piped back up by executing a prolog.

7.7.3.2 Loop Buffer Active or Idle

After reset the loop buffer is idle. The loop buffer becomes active when an **SPLOOP(D/W)** instruction is encountered. The loop buffer remains active until one of the following conditions occur:

- The loop buffer is not reloading and after the last delay slot of a taken branch.
- The SPLOOPW loop stage boundary termination condition is true.
- An interrupt occurs and the loop finishes draining in preparation for interrupt (prior to taking interrupt).
- The SPLOOP(D) loop is finished draining and the loop is not reloading.

When the loop buffer is active, the SPLX bit in TSR is set to 1; when the loop buffer is idle, the SPLX bit in TSR is cleared to 0.

There is one case where the SPLX bit is set to 1 when the loop buffer is idle. When executing a **B IRP** instruction to return to an interrupted SPLOOP, the ITSR is copied back into TSR in the E1 stage of the branch. The SPLX bit is set to 1 beginning in the E2 stage of the branch, which is before the loop buffer has restarted. If the loop buffer state machine is started in the branch delay slots of a **B IRP** or **B NRP** instruction, it uses the SPLX bit to determine if this is a restart of an interrupted SPLOOP. The SPLX bit is not checked if starting an SPLOOP outside the delay slots of one of these branches.

7.7.3.3 Loading Instructions into the Loop Buffer

A loading counter is used to keep track of the current offset from the beginning of the loop and to determine the dynlen. The loading counter is incremented each cycle until an **SPKERNEL** instruction is encountered. When an **SPLOOP(D/W)** instruction is encountered, LBC and the loading counter are cleared to 0. On each cycle thereafter, the instructions fetched from program memory are stored in the loop buffer indexed by LBC along with a record of the loading counter. On the next cycle, these instructions appear as valid in the loop buffer.

When the **SPKERNEL** instruction is encountered, the loop is finished loading, the dynlen is assigned the current value of the loading counter, and program memory fetch is disabled. If the SPKERNEL is on the last kernel stage boundary, program memory fetch may immediately be reenabled (or effectively never disabled).

SPMASKed instructions from program memory are not stored in the loop buffer. The **BNOP** <displacement> instruction does not use a functional unit and cannot be specified by the **SPMASK** instruction, so this instruction is treated in the same way as an **SPMASK**ed instruction.

When returning to an **SPLOOP(D)** instruction with the SPLX bit in TSR set to 1, **SPMASK**ed instructions from program memory execute like a **NOP**. The NOP cycles associated with **ADDKPC**, **BNOP**, or protected **LD** instructions that are masked, are always executed when resuming an interrupted SPLOOP(D).

A warning or error (detected by the assembler) occurs when loading if:

- An **MVC**, **ADDKPC**, or S-unit **B** (branch) instruction appears in the SPLOOP body and the instruction is not masked by an **SPMASK** instruction.
- Another **SPLOOP(D)** instruction is encountered.
- The loading counter reaches 48 before an **SPKERNEL** instruction is encountered.
- A resource conflict occurs when storing an instruction in the loop buffer.
- The dynlen is less than ii + 1 for **SPLOOP(D)** or dynlen < ii for **SPLOOPW**.

The assembler will ensure that there are no resource conflicts that would occur if the first kernel stage were actually reached.

7.7.3.4 Fetching (Dispatching) Instructions from the Loop Buffer

After the first loading stage boundary, instructions marked as valid in the loop buffer at the current LBC are fetched from the loop buffer and executed in parallel with any instructions fetched from program memory. Once fetching begins, it continues until the loop buffer is no longer active for the given loop.

Instructions fetched from the loop buffer that are masked by an **SPMASK** instruction are not executed. An instruction fetched from program memory may execute on the units that were used by an **SPMASK**ed instruction. (See Section 7.15).

7.7.3.5 Disabling (Draining) Instructions in the Loop Buffer

The loop buffer starts draining:

- On the cycle after the loop termination condition is true
- On the cycle after the interrupt is detected and the conditions required for taking the interrupt are met (see Section 7.13).

The draining counter is used to retrace the order in which instructions were loaded into the loop buffer. The draining counter is initialized to 0 and then incremented by 1 each cycle. Instructions in the loop buffer are marked as invalid in the order that they were loaded.

Instructions in the loop buffer indexed by LBC are marked as invalid if their loading counter value (from when they were loaded into the loop buffer) is equal to the draining counter value.

When the draining counter is equal to (dynlen - ii), draining is complete. Any remaining valid instructions for the loop (with a loading counter > (dynlen - ii)) are all marked as invalid.

If the loop is interrupt draining, then program memory fetch remains disabled until the interrupt is taken. If the loop is normal draining, program memory fetch is enabled after a delay specified by the **SPKERNEL(R)** instruction.

7.7.3.6 Enabling (Reloading) Instructions in the Loop Buffer

On the cycle after the reload condition is true (see Section 7.9.6), the loop buffer begins reloading instructions in the loop buffer. Instructions in the loop buffer are marked as valid in the order that they were originally loaded.

The reloading counter is initialized to 0 and then incremented by 1 each cycle until it equals the dynlen. The reloading counter is used to retrace the order in which instructions were loaded into the loop buffer.

Instructions in the loop buffer indexed by LBC are marked as valid, if their loading counter value (from when they were written into the loop buffer) is equal to the reloading counter value.

Reloading does not have to start on a stage boundary. Reloading and draining may access different offsets in the loop buffer. Therefore, there are two LBCs. When reload begins, the unused LBC (the one not being used for draining) is allocated for reloading.

When the reloading counter is equal to the dynlen, the reloading of the software pipeline loop is complete, all the original loop instructions have been reenabled, and the reloading counter stops incrementing.

Program memory fetch of the epilog is disabled when the reload counter equals the dynlen or after the last delay slot of a branch that executed with a true condition. In general, the branch is used in a nested loop to place the PC back at the address of the execute packet after the **SPKERNEL(R)** instruction to reuse the same epilog code between each execution of the inner loop.

A hardware exception is raised while reloading if the termination condition is true and the draining counter for the previous invocation of the loop has not reached the value of dynlen − ii. This describes a condition where both invocations of the loop are attempting to drain at the same time (this could happen, for example, if the RILC value was smaller than the ILC value).

7.8 Execution Patterns

The four loop buffer operations (load, fetch, drain, and reload) are combined in ways that implement various software pipelined loop execution patterns. The three execution patterns are:

- Full execution of a single loop (Section 7.8.1)
- Early exit from a loop (Section 7.8.2)
- Reload of a loop (Section 7.8.3)

7.8.1 Prolog, Kernel, and Epilog Execution Patterns

Figure 7-2 shows a generalization of the basic prolog, kernel, and epilog execution pattern. For simplicity these patterns assume that the **SPKERNEL** instruction appears on a stage boundary.

In Figure 7-3, the termination condition is true on the first kernel stage boundary K0, and falling through to the epilog, the software pipeline only executes a single kernel stage.

Figure 7-2. General Prolog, Kernel, and Epilog Execution Pattern

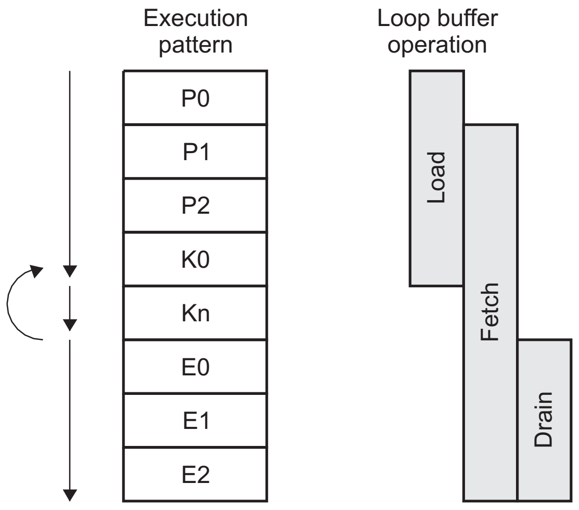
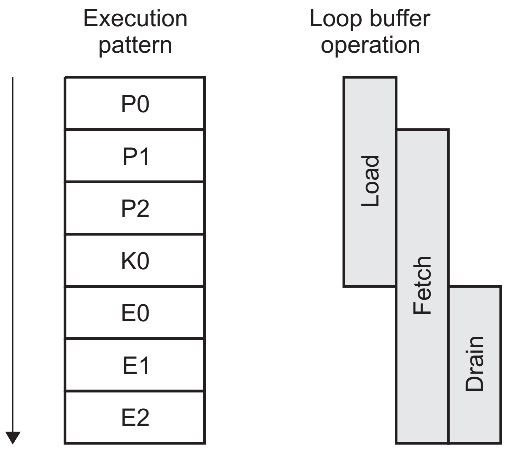


Figure 7-3. Single Kernel Stage Execution Pattern



7.8.2 *Early-Exit Execution Pattern*

If the termination condition is true before an **SPKERNEL(R)** instruction is encountered, then the epilog execution pattern begins before the prolog execution pattern is complete. Since the loop has started draining before it has finished loading, this is referred to as an early-exit.

The execution of a software pipeline early-exit is implemented by beginning to drain the loop buffer by disabling instructions in the order that they were originally loaded, fetching the remaining valid instructions from the loop buffer, and then loading the instructions fetched from program memory into the loop buffer and marking them as valid on the next cycle. An early-exit execution pattern is shown in Figure 7-4. In this case the termination condition was found to be true at the end of P1.

If the termination condition is encountered on the first stage boundary (end of P0) as in Figure 7-5, then no instructions actually execute from the loop buffer. In this special case of early-exit, the loop is only executing a single iteration.

Figure 7-4. Early-Exit Execution Pattern

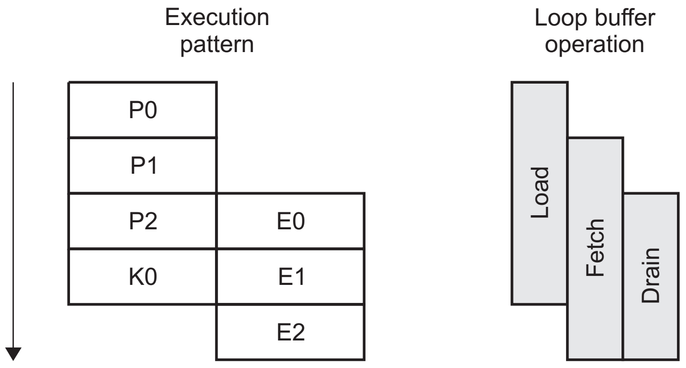
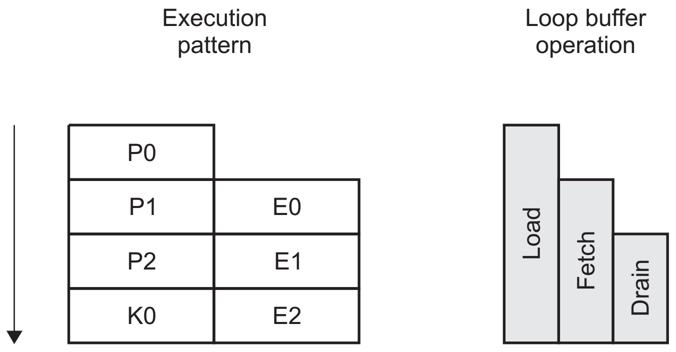


Figure 7-5. Single Loop Iteration Execution Pattern



7.8.3 Reload Execution Pattern

The loop buffer can reload a software pipeline by reactivating the instructions that are stored in the loop buffer. A reload prolog uses the information stored in the loop buffer to reenable instructions in the order that they were originally loaded during the initial prolog.

In Figure 7-6, the loop buffer begins executing a reload prolog while completing the epilog of a previous invocation of the same loop.

The execution of a reload early-exit is implemented by reloading (marking as valid) instructions in the loop buffer, disabling (marking as invalid) instructions in the loop buffer, and then fetching the remaining valid instructions from the loop buffer. A reload early-exit execution pattern is shown in Figure 7-7.

Figure 7-6. Reload Execution Pattern

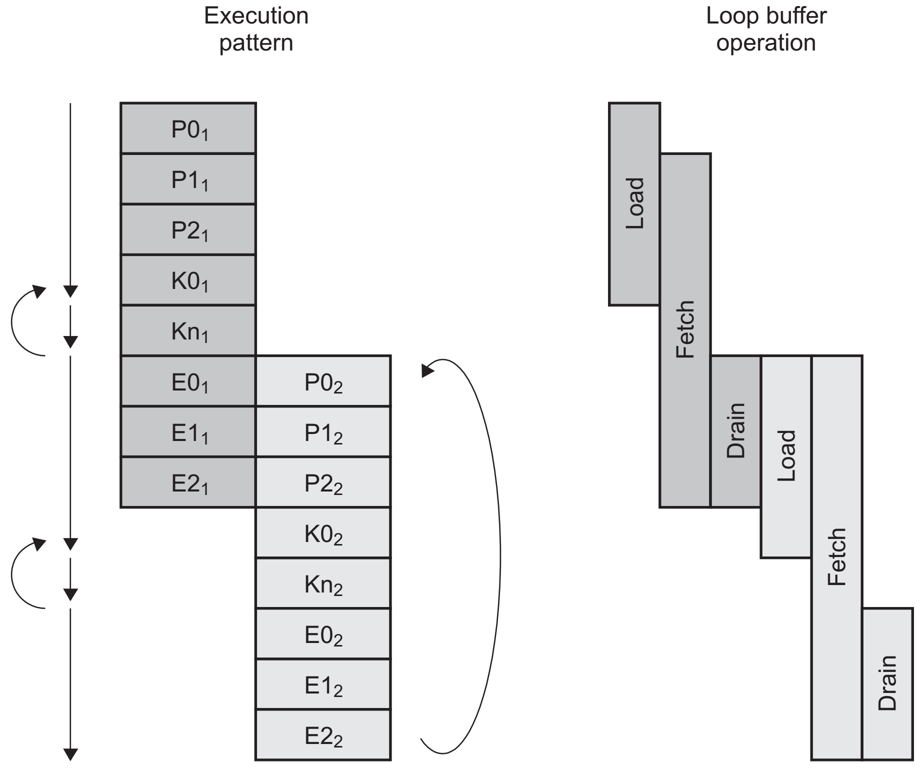
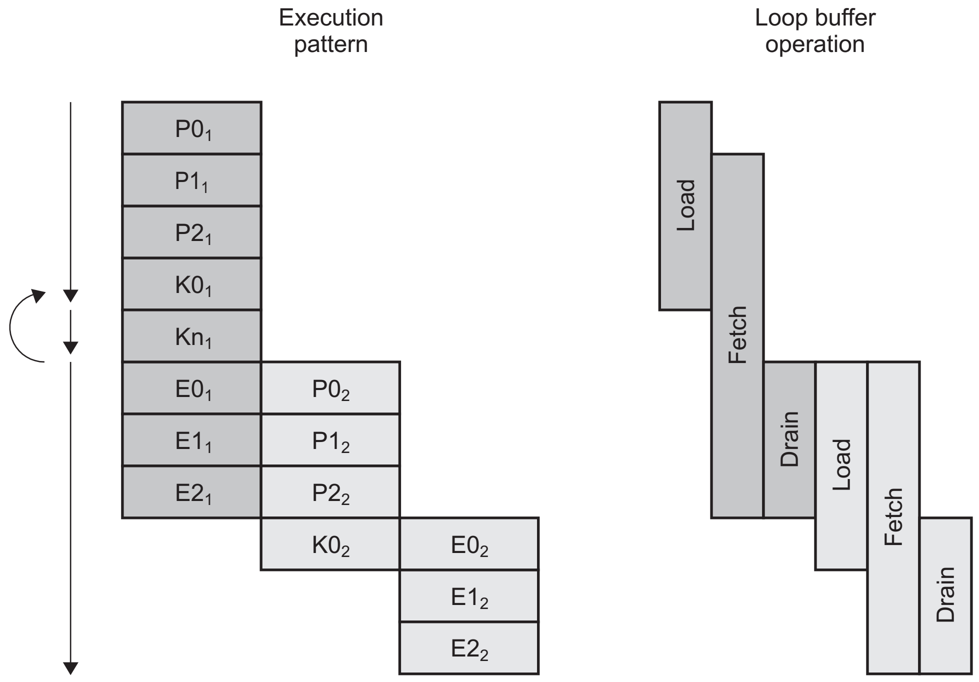


Figure 7-7. Reload Early-Exit Execution Pattern



7.9 Loop Buffer Control Using the Unconditional SPLOOP(D) Instruction

The unconditional form of the **SPLOOP(D)** instruction uses an inner loop count register (ILC) as a down counter, it can delay execution of program memory instructions overlapped with epilog instructions, and it can reload to support nested loops.

7.9.1 Initial Termination Condition Test and ILC Decrement

The termination condition is the set of conditions which determine whether or not to continue the execution of an SPLOOP. The initial termination condition is the value of the termination condition upon entry to the SPLOOP. When using the SPLOOPW or SPLOOPD, the initial termination condition is always false. When using the SPLOOP, the initial termination condition is true if ILC is equal to zero, false otherwise.

If the initial termination condition is true, then the following occur:

- Non-**SPMASK**ed instructions are stored in the loop buffer as disabled.
- Non-**SPMASK**ed instructions execute as NOPs.
- **SPMASK**ed program memory instructions execute as normal.

If the initial termination condition is true and the **SPKERNEL** instruction is unconditional, the loop buffer is idle after the last loading stage boundary. If the **SPKERNEL** instruction is not on a stage boundary, the loop buffer issues NOPs until the last loading stage boundary. If the **SPKERNEL** instruction is conditional, indicating a possible reload, then the reload condition is evaluated at the last loading stage boundary.

When all of the following conditions are true, ILC is decremented:

- An unconditional **SPLOOP** (not SPLOOPD) instruction is encountered.
- ILC is not 0.

The bottom line is, the minimum number of iterations:

- is zero for an SPLOOP;
- depends on the iteration interval, but will be at least one iteration for an SPLOOPD;
- will be at least one iteration for an SPLOOPW.

7.9.2 Stage Boundary Termination Condition Test and ILC Decrement

The stage boundary termination condition is true when a stage boundary is reached and ILC is equal to 0; otherwise, the stage boundary termination condition is false. When the stage boundary termination condition is true, the loop buffer starts draining instructions.

When all of the following conditions are true, ILC is decremented:

- A stage boundary has been reached.
- ILC is not 0.
- The loop is not interrupt draining.
- The loop will not start interrupt draining on the next cycle.

For the first 3 cycles of a loop initiated by an unconditional **SPLOOPD** instruction, the stage boundary termination condition is always false, ILC decrement is disabled, and the loop cannot be interrupted.

If the loop is interrupted and after interrupt draining is complete, ILC contains the current number of remaining loop iterations.

Example 7-9 shows a case in which the value loaded to ILC is determined at run time. The loop may begin draining at any point whenever the ILC decrements to zero (that is, the loop may execute 0 or more iterations). The comments in the example show the stage number (N), the test for termination and the conditional decrement of ILC. ILC will not decrement below zero.

Example 7-9. Using ILC With the SPLOOP Instruction

```
    MVC        A1,ILC           ;ILC = A1
    NOP                         ;delay slot 1
    NOP                         ;delay slot 2
    ZERO       A3               ;delay slot 3
    SPLOOP     1                ;Initial, term_condition=!ILC, if (ILC) ILC--;
    LDW        *A5++,A2         ;Stage 0, term_condition=!ILC, if (ILC) ILC--;
    NOP                         ;Stage 1, term_condition=!ILC, if (ILC) ILC--;
    NOP                         ;Stage 2, term_condition=!ILC, if (ILC) ILC--;
    NOP                         ;Stage 3, term_condition=!ILC, if (ILC) ILC--;
    NOP                         ;Stage 4, term_condition=!ILC, if (ILC) ILC--;
    SPKERNEL   5,0              ;Delay fetch until done with epilog;
||  ADD        A2,A3,A3         ;StageN, term_condition=!ILC, if (ILC) ILC--;
    MV         A3,A4
```

7.9.3 Using SPLOOPD for Loops with Known Minimum Iteration Counts

For loops with known iteration counts, the unconditional **SPLOOPD** instruction is used to compensate for the 4-cycle latency to the assignment of ILC. The unconditional **SPLOOPD** instruction differs from the **SPLOOP** instruction in the following ways:

- The initial termination condition test is always false and the initial ILC decrement is disabled. The loop must execute at least one iteration.
- The stage boundary termination condition is forced to false, and ILC decrement is disabled for the first 3 cycles of the loop.
- The loop cannot be interrupted for the first 3 cycles of the loop.

The **SPLOOPD** will test the SPLX bit in the TSR to determine if it is already set to one (indicating a return from interrupt). In this case the **SPLOOPD** instruction executes like an unconditional **SPLOOP** instruction.

The **SPLOOPD** instruction is used when the loop is known to execute for a minimum number of loop iterations. The required minimum of number of iterations is a function of ii, as shown in Table 7-4.

Table 7-4. SPLOOPD Minimum Loop Iterations

ii	Minimum Number of Loop Iterations
1	4
2	2
3	2
≥ 4	1

When using the **SPLOOPD** instruction, ILC must be loaded with a value that is biased to compensate for the required minimum number of loop iterations. As shown in Example 7-10, for a loop with an ii equal to 1 that will execute 100 iterations, ILC is loaded with 96.

Example 7-10. Using ILC With a SPLOOPD Instruction

```
    MVK       96,B1            ;Execute 96+4 iterations
    SPLOOPD   1                ;Initial, term condition=false
||  MVC       B1,ILC           ;ILC=A1 (E1 Stage)
||  ZERO      A3
    LDW       *A1++,A2         ;Stage0, term condition=false
    NOP                        ;Stage1, term condition=false
    NOP                        ;Stage2, term condition=false
    NOP                        ;Stage3, term_cond=!ILC; if (ILC) ILC--;
    NOP                        ;Stage3, term_cond=!ILC; if (ILC) ILC--;
    SPKERNEL                   ;StageN, term_cond=!ILC; if (ILC) ILC--;
||  ADD       A2,A3,A3
```

7.9.4 Program Memory Fetch Enable Delay During Epilog

After the last kernel stage boundary, program memory fetch is enabled such that instructions are fetched from program memory and executed in parallel with instructions fetched from the loop buffer. This provides for overlapping post-loop instructions with loop epilog instructions.

Program memory fetch enable is delayed until a specific stage and cycle in the execution of the epilog. The **SPKERNEL** instruction fstg and fcyc operands are combined (by the assembler) to calculate the delay in instruction cycles:

delay = (*fstg* * ii) + *fcyc*

Program memory fetch is delayed until the following conditions are all true:

- The loop has reached the last kernel stage boundary
- The loop is not interrupt draining
- The draining counter has reached the delay value specified by *fstg* and *fcyc*.

Referring back to Example 7-4, the program memory fetch delay is set to start fetching after the last epilog instruction.

If the loop buffer goes to idle (for example, if the epilog is smaller than the specified delay or if the loop early−exit execution pattern), program memory fetch is enabled and the fetch enable delay is ignored.

7.9.5 Stage Boundary and SPKERNEL(R) Position

An **SPKERNEL(R)** instruction does not have to occur on a stage boundary. If an **SPKERNEL(R)** instruction is not on a stage boundary and the loop executes 0 or 1 iteration, then the loop buffer executes until the last loading stage boundary. If there are instructions in between the **SPKERNEL(R)** instruction and the last loading stage boundary, the loop buffer issues **NOP** instructions and program memory fetch remains disabled.

If the loop is reloading and the loop executes 0 or 1 iteration, then the loop buffer executes until the last reloading stage boundary. Between when the reloading counter becomes equal to the dynlen and the last reloading stage boundary, the loop buffer issues **NOP** instructions.

7.9.6 Loop Buffer Reload

Using the conditional form of the **SPLOOP(D)** instruction, the loop buffer supports the execution of a nested loop by reloading a new loop invocation while draining the previous invocation of the loop. A loop that reloads must have either an **SPKERNELR** instruction to end the loop body or an **SPMASKR** instruction in the post-**SPLOOP** code.

Under all of the following conditions, the reload condition is true.

- The loop is on the last kernel stage boundary (that is, ILC = 0).
- The **SPLOOP(D)** instruction condition is true 4 cycles before the last kernel stage boundary.

7.9.6.1 Reload Start

When the reload condition is true, reenabling of instructions in the loop buffer begins on the cycle after:

- the last kernel stage boundary for loops using **SPKERNELR**
- an **SPMASKR** is encountered in the post-**SPLOOP** instruction stream.

The reload does not have to start on a stage boundary of the draining loop as indicated by the second and third conditions above.

7.9.6.2 Resetting ILC With RILC

The reload inner loop count register (RILC) is used for resetting the inner loop count register (ILC) for the next invocation of the nested inner loop. There is a 4-cycle latency (3 delay slots) between an instruction that writes a value to RILC and the value appearing to the loop buffer.

If the initial termination condition is false, then the value stored in RILC is extracted, decremented and copied into ILC and normal reloading begins. The value of RILC is unchanged.

If RILC is equal to 0 on the cycle before the reload begins, the initial termination condition is true for the reloaded loop. If the initial termination condition is true, then the reloaded loop invocation is skipped: the instructions in the loop buffer execute as NOPs until the last reloading stage boundary and the reload condition is evaluated again.

7.9.6.3 Program Memory Fetch Disable During Reload

After the reload condition becomes true, program memory fetch is disabled after the last delay slot of a branch that executed with a true condition or after the reload counter equals the dynlen.

The PC remains at its current location when program memory fetch is disabled. If a branch disabled program memory fetch, then the PC remains at the branch target address.

Note that the first condition above is the only time that the loop buffer will not go to idle after the last delay slot of a taken branch.

7.9.6.4 Restrictions on Interruptible Loops that Reload

When the loop buffer has finished loading after returning from an interrupt, the PC points at the address after the **SPKERNEL** instruction. A reloaded loop is not interruptible if a branch does not execute during reloading that places the PC back at the execute packet after the **SPKERNEL** instruction. You should disable interrupts around these types of loops.

7.9.6.5 Restrictions on Reload Enforced by the Assembler

By enforcing the following restrictions by issuing either errors or warnings, the assembler enforces the most common and useful cases for using reload. An assembler switch disables these checks for advanced users.

There must be at least one valid outer loop branch that will always execute with a true condition when the loop is reloading. An outer loop branch is valid under all of the following conditions:

- The branch always executes if the reload condition was true.
- The branch target is the execute packet after the SPKERNEL execute packet.
- The last delay slot of the branch occurs before the reloading counter equals the dynlen. Note that this restriction implies a minimum for dynlen of 6 cycles.

There may be one or more valid post loop branch instructions that will always execute with a false condition when the loop is reloading, and that may execute with a true condition when the loop is not reloading.

For loops initiated with a conditional **SPLOOP** or **SPLOOPD** instruction, an exception (detected by the assembler) occurs if:

- There is not a valid outer loop branch instruction after the **SPKERNEL(R)** instruction.
- A reload has not been initiated by an **SPMASKR** instruction before the delay slots of the outer branch have completed.
- There is a branch instruction after the **SPKERNEL** instruction that may execute when the loop is reloading that is neither a valid outer loop branch nor a valid post loop branch.
- An **SPMASKR** is encountered for a loop that uses **SPKERNELR**.
- An **SPMASKR** is encountered for an unconditional (nonreload) loop.

Example 7-11 is a nested loop using the reload condition. Figure 7-8 shows the instruction execution flow for an invocation of the inner loop, the outer loop code, and then another inner loop. Notice that the reload starts after the first epilog stage of the inner loop as specified by the **SPMASKR** instruction in the last cycle of that stage.

Example 7-11. Using ILC With a SPLOOPD Instruction

```
;*------------------------------
;* for (j=0; j<32; j++)
;* for (I=0; i<32; I++)
;* y[j] += x[i+j] * h[i]
;*------------------------------
;* x=a4, h=b4, y=a6
      MVK        .S2    32,B0
      MVC        .S2    B0,ILC                  ;Inner loop count
      NOP        3
[B0]  SPLOOP     2
||    MVC        .S2    B0,RILC                 ;Reload inner loop count
||    SUB        .D2    B0,1,B0                 ;Outer loop count
||    MVK        .S1    62,A5                   ;X delta
||    MV         .L2    B4,B5                   ;Copy h
||    ZERO       .D1    A7                      ;Sum = 0

;*------------Start of loop------------------------
      LDH        .D1T1  *A4++,A2                ;t1 = *x
||    LDH        .D2T2  *B4++,B2                ;t2 = *h
      NOP        4
      MPY        .M1X   A2,B2,A2                ;p = t1*t2
      NOP        1
      SPKERNEL   0
||    ADD        .L1    A2,A7,A7                ;sum += p

outer:
;*--------start epilog
      SUB        .D1    A4,A5,A4                ;x -= 62
||    MV         .D2    B4,B5                   ;h -= 64
      SPMASKR

;*--------start reload, I=0
[B0]  BNOP       .S1    outer,4
[B0]  SUB        .S2    B0,1,b0                 ;j -= 1
      STH        .D1    A7,*A6++                ;*Y++ = sum
||    MVK        .S1    0,A7                    ;Sum = 0
;*--------branch, stop fetching
```

Figure 7-8. Instruction Flow Using Reload

```
CPU Pipeline         SPL Buffer
---------------      ---------------------------------------------
LD  || LD                 --
nop                       --
nop                       LD  || LD
nop                       --
nop                       LD  || LD
MPY                       --
nop                       LD  || LD
ADD                       MPY
--                        LD  || LD
--                        MPY || ADD
.                         .
.                         .
--                        LD  || LD
--                        MPY || ADD
SUB || MV                 --
SPMASKR                   MPY || ADD            <- ILC = RILC
[B0] BNOP || [B0] SUB                   LD  || LD  <- first reload cycle
nop                       MPY || ADD
nop                                     LD  || LD
nop                       ADD                   <- last epilog cycle
nop                                     LD  || LD
STH || MVK                              MPY
--                                      LD  || LD
--                                      MPY || ADD
.                         .
.                         .
--                        LD  || LD
--                        MPY || ADD
SUB || MV                 --
SPMASKR                   MPY || ADD            <- ILC = RILC
[B0] BNOP || [B0] SUB                   LD  || LD  <- first reload cycle
nop                       MPY || ADD
nop                                     LD  || LD
nop                       ADD                   <- last epilog cycle
nop                                     LD  || LD
STH || MVK                              MPY
--                                      LD  || LD
--                                      MPY || ADD
.                         .
.                         .
```

7.9.7 Restrictions on Accessing ILC and RILC

There is a 4-cycle latency (3 delay slots) between an instruction that writes a value to the inner loop count register (ILC) or the reload inner loop count register (RILC) and a read of the register by the loop buffer.

If an **SPLOOP** (not SPLOOPD or SPLOOPW) instruction is used, then ILC is used for the 3 cycles before the **SPLOOP** instruction and until the loop buffer is draining and not reloading.

If an **SPLOOPD** instruction is used, then ILC is used on the first cycle after the **SPLOOPD** instruction and until the loop buffer is draining and not reloading.

In general, it is an error to read or write ILC or RILC while the loop buffer is using them. This error is enforced by the following hardware and assembler exceptions. The value obtained by reading ILC during loading is not assured to be consistent across different implementations, due to potential differences in timing of the decrement of the register by the loop hardware.

An exception (detected by hardware) occurs if:

- RILC is written in the 3 cycles before the loop buffer reads it on the cycle before reloading begins.
- ILC is written in the 3 cycles before an unconditional **SPLOOP** (not SPLOOPD) instruction.

An error or warning (detected by the assembler) occurs if:

- An **MVC** instruction that writes ILC appears in parallel or in the 3 executes packets preceding an **SPLOOP** (not SPLOOPD) instruction.
- An **MVC** instruction that reads or writes ILC appears in the SPLOOP body.
- An **MVC** instruction that writes the RILC appears in the 3 execute packets preceding the execute packet before a reload prolog is initiated.
- An **MVC** instruction that reads or writes the ILC appears in an execute packet after an **SPKERNEL** instruction in a nested loop, and the **MVC** instruction may execute during or in three cycles preceding the reload prolog of the loop.

7.10 Loop Buffer Control Using the SPLOOPW Instruction

For the **SPLOOPW** instruction, the termination condition is determined by evaluating the **SPLOOPW** instruction condition operand. When the **SPLOOPW** instruction is encountered, the condition operand is recorded by the loop buffer. The initial termination testing is the same as for the **SPLOOPD** instruction, that is, no checking is done for the first four cycles of the loop.

The **SPLOOPW** instruction is intended to be used for do-while loops. These are loops whose termination condition is more complex than a simple down counter by 1. In addition, these types of loops compute the loop termination condition and exit without executing an epilog. This technique may require over executing (or speculating) some instructions.

When using the **SPLOOPW** instruction condition operand as the termination condition, the following behavior occurs:

- Termination is determined by the **SPLOOPW** instruction condition that will be on a stage boundary.
- ILC and RILC are not accessed or modified.
- The loop cannot be reloaded.
- After the last kernel stage boundary, the loop buffer goes idle.
- The stage boundary termination condition is evaluated while interrupt draining.
- The SPKERNEL fetch delay must be 0.
- When returning to a conditional SPLOOPW from an interrupt with the SPLX bit set to 1 in TSR, the SPLOOPW retains its delayed initial termination testing behavior. This is different from the **SPLOOPD** instruction.

7.10.1 Initial Termination Condition Using the SPLOOPW Condition

The initial termination condition is always false when an **SPLOOPW** instruction is encountered. The loop must execute at least one iteration.

7.10.2 Stage Boundary Termination Condition Using the SPLOOPW Condition

The stage boundary termination condition is true when a stage boundary is reached and the **SPLOOPW** instruction condition operand evaluates as false 3 cycles before the stage boundary; otherwise, the termination condition is false. The termination condition is always false for the first 3 cycles of the loop.

7.10.3 Interrupting the Loop Buffer When Using SPLOOPW

If the loop is interrupted when using the conditional form of the **SPLOOPW** instruction, the stage boundary termination condition is evaluated on each stage boundary while interrupt draining. If the stage boundary termination condition is true while interrupt draining, the loop buffer goes to idle, execution resumes at the instruction after the loop body, and that instruction is interrupted.

The instruction that defines the termination condition register must occur at least 4 cycles before a stage boundary and at least 4 cycles before the last instruction in the loop. If the termination condition is determined in the last loading stage, the dynlen must be a multiple of ii. These restrictions ensure that on return from an interrupt to a **SPLOOPW** instruction, the loop executes 1 or more iterations.

Note that when returning to a **SPLOOPW** instruction from an interrupt service routine with the SPLX bit set to 1 in TSR, the **SPLOOPW** instruction termination condition behavior is unchanged, that is, the initial termination condition is always false, and the stage boundary termination condition is always false for the first 3 cycles of the loop.

An exception occurs if the termination condition register is not defined properly to ensure correct interrupt behavior.

Example 7-12 shows a loop with a loop counter that down counts by an unknown value. For this loop, it must be safe to over-execute the **LDH** instructions 8 times.

Example 7-13 shows a string copy implementation. Figure 7-9 shows the execution flow if the source points to a null string. In this version, it must be safe to over-execute the **LDB** instruction 4 times.

Example 7-12. Using the SPLOOPW Instruction

```
;*------------------------------
;* do  {
;      sum += *x++ * *y++;
;      n -= m;
;      } while (n >= 0)
;*------------------------------
[!A1]  SPLOOPW 1
||     MVK     .S1    0x0,A1                  ;C = false
       LDH     .D1T1  *A5++,A3                ;t1 = *x++
||     LDH     .D2T2  *B5++,B6                ;t2 = *y++
       NOP     2
       SUB     .L2    B4,B7,B4                ;n -=m
       CMPLT   .L2    B4,0,A1                 ;c = n < 0      // term_cond = !A1
       MPY     .M1X   B6,A3,A4                ;p = t1 * t2    // delay slot 1
       NOP     1                                              // delay slot 2
       ADD     .L1    A4,A6,A6                ;sum += p;      // delay slot 3
       SPKERNEL                               ;if (c) break;  // cycle term_cond used
```

Example 7-13. strcpy() Using the SPLOOPW Instruction

```
;*------------------------------
;* do  {
;      t = *src++;
;      *dst++ = t;
;      } while (t != 0)
;*------------------------------
    [A0]  SPLOOPW  1
||        MVK     .S2   1,B0
||        MVK     .S1   1,A0
    [A0]  LDB     .D1   *A4++,A0            ;t = *src++
          NOP     4
    [B0]  MV      .L2X  A0,B0               ;if (!t) break;
          NOP     2                         ;Ensure A0 set 4 cycles early
SPKERNEL
||  [B0]  STB    .D2   B0,*B4++             ;*dest++ = t
          STB    B0,*B4                     ;*t = '/0'
```

Figure 7-9. Instruction Flow for strcpy() of Null String

```
CPU Pipeline           SPL buffer
-----------------      ------------------------------------
[a0] LDB *a4++,a0           --
     nop               [a0] LDB *a4++,a0
     nop               [a0] LDB *a4++,a0
     nop               [a0] LDB *a4++,a0
     nop               [a0] LDB *a4++,a0 0 written to a0 in this cycle
[b0] mv a0,b0          [a0] LDB *a4++,a0 <- a0 = 0, term_cond = true
     nop               [a0] LDB *a4++,a0 || [b0] mv a0,b0 <- b0 = 0
     nop               [a0] LDB *a4++,a0 || [b0] mv a0,b0
[b0] STB b0,*b4++      [a0] LDB *a4++,a0 || [b0] mv a0,b0
     STB b0,*a4             –              terminate string in post epilog with /0
```

7.10.4 Under-Execution of Early Stages of SPLOOPW When Termination Condition Becomes True While Interrupt Draining

Usually an SPLOOPW block terminates abruptly when the termination condition is true without executing an epilog; however, when an SPLOOPW block is interrupted, it executes an epilog to drain the loop prior to servicing the interrupt.

If the termination condition becomes true while interrupt draining, the action of interrupt draining results in the under-execution of the early stages of the loop body in comparison to the same loop when not interrupted. The loop body must be coded such that the under-execution of the early stages of the loop body are safe.

7.11 Using the SPMASK Instruction

A logical progression for a loop might be:

- Do initial setup
- Execute the loop
- Do post loop operations

If the loop were to be reloaded, the progression might loop like:

- Do initial setup
- Execute the loop
- Adjust setup for reloaded loop
- Reload the loop
- Do post loop operations

The initial setup, the post loop operations, and adjusting the setup for the reloaded loop are all overhead that may be minimized by moving their execution to within the same instruction cycles as the operation of the SPLOOP.

If some setup code is required to do some initialization that is not used until late in the loop; you can save instruction cycles by using the **SPMASK** instruction to overlay the setup code with the first few cycles of the SPLOOP. The **SPMASK** will cause the masked instructions to be executed once without being loaded to the SPLOOP buffer. Example 7-14 shows how this might be done.

If the **SPMASK** is used in the outer loop code (that is, post epilog code), it will force the substitution of the ***SPMASK***ed instructions in the outer loop code for the instruction using the same functional unit in the SPLOOP buffer for the first iteration of the reloaded inner loop. For example, if pointers need to be reset at the point that a loop is reloaded, the instructions that do the reset can be inhibited using the **SPMASK** instruction so that the instructions that originally adjusted the pointers are replaced in the execution flow with instruction in the outer loop that are marked with the **SPMASK** instruction. Example 7-15 shows how this might be done.

7.11.1 Using SPMASK to Merge Setup Code Example

Example 7-14 copies a number of words (the number is passed in the A6 register from one buffer to an offset into another buffer). The size of the offset is passed in the B6 register. Due to the use of the **SPMASK** instruction, the ADD instruction is executed only once and is not loaded to the SPLOOP buffer. The caret (^) symbol is used to identify the instructions masked by the **SPMASK** instruction. Table 7-5 shows the instruction flow for the first three iterations of the loop.

Example 7-14. Using the SPMASK Instruction to Merge Setup Code with SPLOOPW

```
;*------------------------------
;  dst=&(dst[n])
;* do  {
;      t = *src++;
;      *dst++ = t;
;      } while (count--)
;
;A4 = Source address
;B4 = Destination address
;A6 = Number of words to copy
;B6 = Offset into destination to do copy
;*------------------------------
[A1]  SPLOOPW   1
||    ADD       .L1   A6,1,A1         ;Position loop cnt to valid reg
||    SHL       .S2   B6,2,B6         ;Adjust offset for size of WORD
      SPMASK
||^   ADD       .L2   B6,B4,B4        ;Add offset into buffer to dest
||    LDW       .D1   *A4++,A0        ;Load word and inc ptr
      NOP       1                     ;Wait for portion of delay
[A1]  SUB       .S1   A1,1,A1         ;Decrement loop count
      NOP       2                     ;Complete necessary wait
      MV        .L2X  A0,B0           ;Position Word for write
      SPKERNEL  0,0
||    STW       .D2   B0,*B4++        ;Store word
```

Table 7-5. SPLOOP Instruction Flow for First Three Cycles of Example 7-14

	Loop			
Cycle	**1**	**2**	**3**	**Notes**
0	ADD SHL			Instructions are in parallel with the SPLOOP, so they execute only once.
1	ADD LDW			The ADD is SPMASKed so it executes only once. The LDW is loaded to the SPLOOP buffer.
2	NOP	LDW		The ADD was not added to the SPLOOP buffer in cycle 2, so it is not executed here.
3	SUB	NOP	LDW	The SUB is a conditional instruction and may not execute.
4	NOP	SUB	NOP	The SUB is a conditional instruction and may not execute.
5	NOP	NOP	SUB	The SUB is a conditional instruction and may not execute.
6	MV	NOP	NOP	
7	STW	MV	NOP	
8		STW	MV	
9			STW	

7.11.2 Some Points About the SPMASK to Merge Setup Code Example

Note the following points about the execution of Example 7-14:

- The **ADD** and **SHL** instructions in the same execute packet as the **SPLOOPW** instruction are only executed once. They are not loaded to the SPLOOP buffer.
- Because of the **SPMASK** instruction in the execute packet, the **ADD** in the same execute packet as the **SPMASK** instruction is executed only once and is not loaded to the SPLOOP buffer. Without the **SPMASK**, the **ADD** would conflict with the **MV** instruction.
- The **SHL** and the 2nd **ADD** instructions could have been placed before the start of the SPLOOP, but by placing the **SHL** in parallel with the **SPLOOP** instruction and by using the **SPMASK** to restrict the **ADD** to a single execution, you have saved a couple of instruction cycles.

7.11.3 Using SPMASK to Merge Reset Code Example

Example 7-15 copies a number of words (the number is passed in the A8 register from one buffer to another buffer). The loop is reloaded and the contents of a second source buffer are copied to a second destination buffer. Table 7-6 shows the instruction flow for the first 13 cycles of the example.

Example 7-15. Using the SPMASK Instruction to Merge Reset Code with SPLOOP

```
;*------------------------------
;  dst=&(dst[n])
;* do  {
;      t = *src++;
;      *dst++ = t;
;      } while (count--)
;  adjust buffer pointers
;* do  {
;      t = *src++;
;      *dst++ = t;
;      } while (count--)
;
;A4 = 1st source address
;B4 = 1st destination address
;A6 = 2nd source address
;B6 = 2nd destination address
;A8 = number of locations to copy from each buffer
;*------------------------------
          MVC      A8,ILC                  ;Setup number of loops
          MVC      A8,RILC                 ;Reload count
          MVK      1,A1                    ;Reload flag
          NOP      3                       ;Wait for ILC load to complete
    [A1]  SPLOOP   1                       ;Start SPLOOP with ii=1
          LDW      .D1   *A4++,A0          ;Load value from buffer
          NOP      4                       ;Wait for it to arrive
          MV       .L2X  A0,B0             ;Move it to other side for xfer
          SPKERNELR                        ;End of SPLOOP, immediate reload
          STW      .D2   B0,*B4++          ;...and store value to buffer
BR_TARGET:
          SPMASK   D1                      ;Mask LDW instruction
||  [A1]  B        BR_TARGET               ;Branch to start if post-epilog
||  [A1]  SUB      .S1   A1, 1, A1         ;Adjust reload flag
||  [A1]  LDW      .D1   *A6,A0            ;Load first word of 2nd buffer
||  [A1]  ADD      .L1   A6,4,A4           ;Select new source buffer
          NOP      4                       ;Keep in sync with SPLOOP body
          OR       .S2   B6,0,B4           ;Adjust destination to 2nd buffer
          NOP
```

Table 7-6. SPLOOP Instruction Flow for Example 7-15

	Loop												
Cycle	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**
1	LDW												
2	NOP	LDW											
3	NOP	NOP	LDW										
4	NOP	NOP	NOP	LDW									
5	NOP	NOP	NOP	NOP	LDW								
6	MV	NOP	NOP	NOP	NOP	LDW							
7	STW	MV	NOP	NOP	NOP	NOP	LDW						
8		STW	MV	NOP	NOP	NOP	NOP	LDW SUB ADD					
9			STW	MV	NOP	NOP	NOP	NOP	LDW				
10				STW	MV	NOP	NOP	NOP	NOP	LDW			
11					STW	MV	NOP	NOP	NOP	NOP	LDW		
12						STW	MV	NOP	NOP	NOP	NOP	LDW	
13							STW	MV OR	NOP	NOP	NOP	NOP	LDW
14								STW	MV	NOP	NOP	NOP	NOP

7.11.4 *Some Points About the SPMASK to Merge Reset Code Example*

Note the following points about the execution of Example 7-15 (see Table 7-6 for the instruction flow)::

- The loop begins reloading from the SPLOOP buffer immediately after the **SPKERNELR** instruction with no delay. In Table 7-6, the **SPKERNELR** is in cycle 7 and the reload happens in cycle 8.
- Because of the **SPMASK** instruction, the **LDW** instruction in the post epilog code replaces the **LDW** instruction within the loop, so that the first word copied in the reloaded loop is from the new input buffer. The **ADD** instruction is used to adjust the source buffer address for subsequent iterations within the SPLOOP body. In Table 7-6, this happens in loop 8. Note that the D1 operand in the **SPMASK** instruction indicates that the **SPMASK** applies to the .D1 unit. This could have been indicated by marking the **LDW** instruction with a caret (^) instead.
- The **OR** instructions are used to adjust the destination address. It is positioned in the post-epilog code as the **MV** instruction is within the SPLOOP body so that it will not corrupt the data from the **STW** instructions within the SPLOOP epilog still executing from before the reload. In Table 7-6, this happens in cycle 13 (loop 8).
- The **B** instruction is used to reset the program counter to the start of the epilog between executions of the inner loop.

7.11.5 *Returning from an Interrupt*

When an SPLOOP is piping up after returning from an interrupt, the **SPMASK**ed instructions coming from the buffer are executed and instructions coming from program memory are not executed.

7.12 Program Memory Fetch Control

When the loop buffer is active and program memory fetch is enabled, then instructions are fetched from program memory and the loop buffer and executed in parallel.

When the loop buffer is active and under certain conditions as described below, instruction execution from program memory is suspended. When this occurs, instructions are only fetched and executed from the loop buffer and the PC is unchanged.

7.12.1 Program Memory Fetch Disable

Instruction fetch from program memory is disabled under the following conditions:

- **When loading:** on the cycle after the **SPKERNEL** instruction is encountered, and that cycle is either a kernel cycle or a draining cycle where fetch has not yet been reenabled.
- **When reloading:** on the cycle after the last delay slot of a branch that executes with a true condition or after the reload counter equals the dynlen.

If program memory fetch is disabled on the last loading or reloading stage boundary, the stage boundary termination condition is true, and the program memory fetch enable delay has completed, then program memory fetch is not disabled.

Program memory fetch remains disabled while interrupt draining or until a specific stage and cycle during noninterrupt draining as determined by the program fetch enable delay operand of the **SPKERNEL** instruction.

7.12.2 Program Memory Fetch Enable

Program memory fetch is enabled, if the loop buffer goes idle.

7.13 Interrupts

When an **SPLOOP(D/W)** instruction is encountered, the address of the execute packet containing the **SPLOOP(D/W)** instruction is recorded. If the loop buffer is interrupted, the address stored in the interrupt return pointer register (IRP) is the address of the execute packet containing the **SPLOOP(D/W)** instruction.

7.13.1 Interrupting the Loop Buffer

Interrupts are automatically disabled 2 cycles before an **SPLOOP(D/W)** instruction is encountered. When all of the following conditions are true, the loop buffer begins interrupt draining.

- An enabled interrupt is pending and not architecturally blocked (for example, in branch delay slots).
- The loop is on a stage boundary.
- The termination condition is false.
- The loop is not loading.
- The loop is not draining.
- The loop is not reloading or waiting to reload.
- The loop is not within the first 3 CPU cycles after an **SPLOOPD** or **SPLOOPW** instruction. This means that a minimum number of 4 cycles of an **SPLOOP(D/W)** loop must be executed before an interrupt can be taken.
- The loop is not within the first 3 CPU cycles after an **SPLOOPD** or **SPLOOPW** instruction.
- For **SPLOOP** or **SPLOOPD** instructions, the current ILC >= ceil(dynlen/ii). This prevents returning to a loop that would early-exit. The value of ceil(dynlen/ii) is equal to the number of loading stages.

When the loop is finished draining and all pending register writes are complete the interrupt is taken. This means that the interrupt latency has increased by the number of instruction cycles in the epilog compared to the non−SPLOOP case.

The above conditions mean SPLOOP loops starting initial execution or starting reload with ILC < = (ceil(dynlen / ii) + 3) are not interruptible because there are not enough kernel stages to allow an interrupt to be taken without violating the last requirement.

After an **SPLOOP(D/W)** instruction is encountered, the SPLX bit is set to 1 in TSR. While the loop buffer is active, the SPLX bit is 1. When the loop buffer is idle, the SPLX bit in TSR is cleared to 0.

Program memory fetch is disabled when interrupt draining. When the draining is finished, the address of the execute packet that contains the **SPLOOP** instruction is stored in IRP or NRP, and TSR is copied to ITSR or NTSR. The SPLX bit in TSR is cleared to 0. The SPLX bit in ITSR or NTSR is set to 1.

Interrupt service routines must save and restore the ITSR or NTSR, ILC, and RILC registers. A **B IRP** instruction copies ITSR to TSR, and a **B NRP** restores TSR from NTSR. The value of the SPLX bit in ITSR or NTSR when the return branch is executed is used to alter the behavior of **SPLOOP(D/W)** when it is restarted upon returning from the interrupt.

7.13.2 Returning to an SPLOOP(D/W) After an Interrupt

When returning from an interrupt to an SPLOOP(D/W) instruction with the SPLX bit set to 1 in ITSR, the loop buffer executes normally with the following exceptions:

- Instructions executing in parallel with the **SPLOOP(D/W)** instruction are not executed.
- **SPMASK**ed instructions from program memory execute as a NOP.
- **SPMASK**ed instructions in the loop buffer execute as normal - the SPMASK is ignored.
- **BNOP** *label,n* instructions are executed as **NOP** *n* + 1
- An **SPLOOPD** instruction executes as an **SPLOOP** instruction.

Note that if returning to an unconditional **SPLOOP(D)** instruction, the interrupt return code must restore the value of ILC 4 cycles before the **SPLOOP(D)** instruction is executed (if the ISR modified ILC).

7.13.3 Exceptions

If an internal or external exception occurs while the loop buffer is active, then the following occur:

- The exception is recognized immediately and the loop buffer becomes idle.
- The loop buffer does not execute an epilog to drain the currently executing loop.
- TSR is copied into NTSR with the SPLX bit set to 1 in NTSR and cleared to 0 in TSR.

7.13.4 Branch to Interrupt, Pipe-Down Sequence

1. Hardware detects an interrupt.
2. Execute until the end of a stage boundary (if termination condition is false).
3. Pipe-down the SPLOOP by draining.
4. Fetch enable condition is false.
5. Store return address of SPLOOP in IRP or NRP.
6. Copy TSR to ITSR or NTSR with SPLX bit set to 1.
7. Complete all pending register writes (drain pipeline).
8. Begin execution at interrupt service routine target address.

7.13.5 Return from Interrupt, Pipe-Up Sequence

1. Copy ITSR or NTSR to TSR.
2. Pipe-up the SPLOOP.
3. The **SPLOOPD** instruction executes like the **SPLOOP** instruction.
4. The **SPMASK**ed instructions from program memory are executed like NOPs.
5. The **SPMASK**ed instructions in the loop buffer execute as normal.
6. Instructions in parallel with the **SPLOOP(D/W)** instruction are executed like NOPs.

7.13.6 Disabling Interrupts During Loop Buffer Operation

Instructions that disable interrupts should not be executed within 4 cycles of reaching dynlen while loading or reloading. If this condition is violated, there is a possibility that an interrupt is recognized as enabled causing the loop to drain for an interrupt with the interrupt no longer enabled when draining is completed. In this case, the loop terminates and execution continues with the post-**SPKERNEL** instruction stream with no interrupt being serviced at that point.

7.14 Branch Instructions

If a branch executes with a true condition or is unconditional, then the branch is taken on the cycle after the 5 delay slots have expired.

If a branch is taken and the loop buffer is not reloading, the loop buffer becomes idle, and execution continues from the branch target address.

If a branch executes with a false condition (the branch is not taken), the execution of the **SPLOOP(D/W)** instruction is unaffected by the presence of the untaken branch except that interrupts are blocked during the delay slots of the branch.

This behavior allows the code in Example 7-16 to run as you expect, branching around the loop if the condition is false before beginning.

If a branch is taken anytime while the loop buffer is active, except when in reloading, the loop buffer goes to idle, and execution continues from the branch target address. If a branch is taken while reloading, the PC is assigned the branch target and program memory fetch is disabled.

Example 7-16. Initiating a Branch Prior to SPLOOP Body

```
   [!A0]   B       around
||         MVC     A0,ILC
           NOP     3
           SPLOOP  ii
   ; loop body
      . . .
   ; end of loop body
around:
   ; code following loop
```

7.15 Instruction Resource Conflicts and SPMASK Operation

There are three execution candidates for each unit: prolog instructions coming from the buffer (BP), epilog instructions coming from the buffer (BE), and nonbuffer instructions coming from program memory (PM). There are four phases where conflict can occur:

- Loading phase, between BP and PM
- Draining only phase, between BE and PM
- Draining/reload phase, between BE, BP, and PM
- Reload only phase, between BP and PM

In the case of any conflict, an **SPMASK(R)** instruction must be present specifying all units having conflicts. SPMASKed units for that cycle:

- Disable execution of any loop buffer instructions: BP, BE, or both.
- Execute a PM instruction, if present, with no effect on the buffer contents.

The only special behavior is in the case of restarting SPLOOP(D/W) after return from interrupt. In this case, during loading SPMASKed units:

- Do not disable execution of any loop buffer instructions.
- Do not execute a present PM instruction.

If an **SPMASK** instruction is encountered when the loop buffer is idle or not loading or not draining, the **SPMASK** instruction executes as a NOP.

7.15.1 Program Memory and Loop Buffer Resource Conflicts

A hardware exception occurs if:

- An instruction fetched from program memory has a resource conflict with an instruction fetched from the loop buffer and the instruction coming from the loop buffer is not masked by an **SPMASK(R)** instruction.
- An instruction fetched from the loop buffer as part of draining has a resource conflict with an instruction fetched from the loop buffer for reload and the unit with the conflict is not masked by an **SPMASK(R)** instruction.

7.15.2 Restrictions on Stall Detection Within SPLOOP Operation

There are two CPU stalls that occur because of certain back-to-back execute packets. In both of these cases, the CPU generates a 1-cycle stall between the instruction doing the write and instruction using the written value.

- The cross path register file read stall where the source for an instruction executing on one side of the datapath is a register from the opposite side and that register was written in the previous cycle. No stall is required or inserted when the register being read has data placed by a load instruction. See Section 3.7.4 for more information.
- The AMR use stall where an instruction uses an address register in the cycle immediately following a write to the addressing mode register (AMR).

Stall detection is one critical speed path in the CPU design. Adding to that path for the case where instructions are coming from the loop buffer is undesirable and unnecessary. There are no compelling cases where you would want to schedule a stall within the loop body. In fact, the compiler works to ensure this does not happen. For these reasons, the C64x+ CPU will not stall for instructions coming from the loop buffer that read/use values written on the previous cycle that require a stall for correct behavior.

In the event that a case occurs where a stall is required for correct operation but did not occur, an internal exception is generated. This internal exception sets the LBX and MSX bits in the internal exception report register (IERR), indicating a missed stall with loop buffer operation. The exception is only generated in the event that the stall is actually required.

There is one special case that causes an unnecessary stall in normal operation and can be generated by the compiler. It is the case where the two instructions involved in the stall detection are predicated on opposite conditions. This means only one of the instructions actually executes and a stall was not required for correct behavior. Since the stall detection is earlier in the pipeline, the decision to stall must be made before it is known whether the instructions execute. Thus a stall is caused, even though it later turns out not to be needed. In this case, the lack of detection for the instruction coming from the loop buffer does not cause incorrect behavior. This allows the compiler to continue to generate code using this case that can result in improved scheduling and performance. The internal exception is not generated in this case.

7.16 Restrictions on Cross Path Stalls

The following restriction is enforced by the assembler (that is, an assembly error will be signaled): an instruction fetched from the loop buffer that reads a source operand from the register file cross path must be scheduled such that the read does not require a cross path stall (the register being read cannot be written in the previous cycle).

It is possible for the assembly language programmer to place an instruction in the delay slots of a branch to an SPLOOP that causes a pipelined write to happen while the loop buffer is active. It is also possible for the assembly language programmer to predicate the write and reads with different predicate values that are not mutually exclusive. The assembler cannot prevent these cases from occurring; if they do the internal exception will occur.

7.17 Restrictions on AMR-Related Stalls

The following restriction is enforced by the assembler: an instruction fetched from the loop buffer that uses an address register (A4−A7 or B4−B7) must be scheduled such that a write to the addressing mode register (AMR) does not occur in the preceding cycle.

7.18 Restrictions on Instructions Placed in the Loop Buffer

The following instructions cannot be placed in the loop buffer and must be masked by **SPMASK(R)** when occurring in the loop body: **ADDKPC**, **B** *reg*, **BNOP** *reg*, **CALLP**, and **MVC**.

The **NOP**, **NOP** ***n***, and **BNOP** instructions are the only unitless instructions allowed to be used in an SPLOOP(D/W) body. The assembler disallows the use of any other unitless instruction in the loop body.

C64x+ CPU Privilege

This chapter describes the C64x+ CPU privilege system.

The C64x CPU does not support privilege. This chapter applies only to the C64x+ CPU.

Topic **Page**

8.1 Overview

The C64x+ CPU includes support for a form of protected-mode operation with a two-level system of privileged program execution. The C64x CPU does not include support for privileged execution. This chapter does not apply to the C64x CPU.

The privilege system is designed to support several objectives:

- Support the emergence of higher capability operating systems on the C6000 family architecture.
- Support more robust end-equipment, especially in conjunction with exceptions.
- Provide protection to support system features such as memory protection.

The support for powerful operating systems is especially important. By dividing operation into privileged and unprivileged modes, the operating mode for the operating system is differentiated from applications, allowing the operating system to have special privilege to manage the processor and system. In particular, privilege allows the operating system to:

- control the operation of unprivileged software
- protect access to critical system resources (that is, interrupts)
- control entry to itself

The privilege system allows two distinct types of operation.

- Supervisor-only execution. This is used for programs that require full access to all control registers, and have no need to run unprivileged (User mode) programs. This case includes legacy (preprivilege) programs that do not comprehend a privilege system. Legacy programs run fully compatibly with the understanding that undefined or illegal operations may behave differently on the C64x+ CPU than on previous C64x devices. For example, an illegal opcode may result in an exception on the C64x+ CPU, whereas, it previously had undefined results.
- Two-tiered system. This is where the OS and trusted applications execute in Supervisor mode, and less trusted applications execute in User mode.

8.2 Execution Modes

There are two execution modes:

- Supervisor Mode
- User Mode

8.2.1 Privilege Mode After Reset

Reset forces the C64x+ CPU to the Supervisor mode. Execution of the reset interrupt service fetch packet (ISFP) begins in Supervisor mode.

8.2.2 Execution Mode Transitions

Mode transitions occur only on the following events:

- Interrupt: goes to Supervisor mode and saves mode (return with **B IRP** instruction)
- **B IRP** instruction: returns to saved mode from interrupt
- Nonmaskable interrupt (NMI): goes to Supervisor mode and saves mode (return with **B NRP** instruction)
- Exception: goes to Supervisor mode and saves mode (return with **B NRP** instruction, if restartable)
- Operating system service request: goes to Supervisor mode and saves mode (return with **B NRP** instruction)
- **B NRP** instruction: returns to saved mode from NMI or exception

8.2.3 Supervisor Mode

The Supervisor mode serves two purposes:

1. It is the compatible execution mode.
2. It is the privileged execution mode where all functions of the processor are available. In User mode, the privileged operations and resources that are restricted are listed in Section 8.2.4.

8.2.4 User Mode

The User mode provides restricted capabilities such that more privileged supervisory software may manage the machine with complete authority. User mode restricts access to certain instructions and control registers to prevent an unprivileged program from bypassing the management of the hardware by the supervisory software.

8.2.4.1 Restricted Control Register Access in User Mode

Certain control registers are not available for use in User mode. An attempt to access one of these registers in User mode results in an exception. The resource access exception (RAX) and privilege exception (PRX) bits are set in the internal exception report register (IERR) when this exception occurs. The following control registers are restricted from access in User mode:

- Exception clear register (ECR)
- Exception flags register (EFR)
- Interrupt clear register (ICR)
- Interrupt enable register (IER)
- Internal exception report register (IERR)
- Interrupt flags register (IFR)
- Interrupt service table pointer register (ISTP)
- Interrupt task state register (ITSR)
- NMI/exception task state register (NTSR)
- Restricted entry point register (REP)

8.2.4.2 Partially Restricted Control Register Access in User Mode

The following control registers are partially restricted from access in User mode:

- Control status register (CSR)
- Task state register (TSR)

All bits in these registers can be read in User mode; however, only certain bits in these registers can be written while in User mode. Writes to these restricted bits have no effect. Since access to some bits is allowed, there is no exception caused by access to these registers.

8.2.4.2.1 Restrictions on Using CSR in User Mode

The following functions of CSR are restricted when operating in User mode:

- PGIE, PWRD, PCC, and DCC bits cannot be written in User mode. Writes to these bits have no effect.
- GIE and SAT bits are not restricted in User mode, and their behavior is the same as in Supervisor mode.

8.2.4.2.2 Restrictions on Using TSR in User Mode

The GIE and SGIE bits are not restricted in User mode. All other bits are restricted from being written; writes to these bits have no effect.

8.2.4.3 Restricted Instruction Execution in User Mode

Certain instructions are not available for use in User mode. An attempt to execute one of these instructions results in an exception. The opcode exception (OPX) and privilege exception (PRX) bits are set in the internal exception report register (IERR) when this exception occurs. The following instructions are restricted in User mode:

- **B IRP**
- **B NRP**
- **IDLE**

8.3 Interrupts and Exception Handling

As described in Section 8.2.2, mode switching mostly occurs for interrupt or exception handling. This section describes the execution mode behavior of interrupt and exception processing.

8.3.1 Inhibiting Interrupts in User Mode

The GIE bit in the control status register (CSR) can be used to inhibit interrupts in User mode. This allows a usage model where User mode programs may be written to conform to a required level of interruptibility while still protecting segments of code that cannot be interrupted safely. Nonconforming behavior may be detected at the system level, and control can be taken from the User mode program by asserting the EXCEP input to the CPU.

8.3.2 Privilege and Interrupts

When an interrupt occurs, the interrupted execution mode and other key information is saved in the interrupt task state register (ITSR). The CXM bit in the task state register (TSR) is set to indicate that the current execution mode is Supervisor mode. Explicit (**MVC**) writes to TSR are completed before being saving to ITSR.

The interrupt handler begins executing at the address formed by adding the offset for the particular interrupt event to the value of the interrupt service table pointer register (ISTP). The return from interrupt (**B IRP**) instruction restores the saved values from ITSR into TSR, causing execution to resume in the execution mode of the interrupted program.

The transition to the restored execution mode is coincident to the execution of the return branch target. Execution of instructions in the delay slot of the branch are in Supervisor mode.

8.3.3 Privilege and Exceptions

When an exception occurs, the interrupted execution mode and other key information is saved in the NMI/exception task state register (NTSR). The CXM bit the task state register (TSR) is set to indicate that the current execution mode is Supervisor mode. Explicit (**MVC**) writes to TSR are completed before saved to ITSR.

The exception handler begins executing at the address formed by adding the offset for the exception/NMI event to the value of the interrupt service table pointer register (ISTP). The return from exception (**B NRP**) instruction restores the saved values from NTSR into TSR.

8.3.4 Privilege and Memory Protection

The data and program memory interfaces at the boundary of the CPU include signals indicating the execution mode in which an access was initiated. This information can be used at the system level to raise an exception in the event of an access rights violation.

8.4 Operating System Entry

A protected interface is needed so that User mode code can safely enter the operating system to request service.

There is one potential problem with allowing direct calling into the operating system: the caller can chose where to enter the OS and, if allowed to choose any OS location to enter, can:

- bypass operand checking by OS routines
- access undocumented interfaces
- defeat protection
- corrupt OS data structures by bypassing consistency checks or locking

In short, allowing unrestricted entry into an OS is a very bad idea. Instead, you need to give a very controlled way of entering the operating system and switching from User mode to Supervisor mode. The mechanism chosen is essentially an exception, where the handler decodes the requested operation and dispatches to a Supervisor mode routine that validates the arguments and services the request.

8.4.1 Entering User Mode from Supervisor Mode

There are two reasons that the CPU might need to enter User mode while operating in Supervisor mode:

- To spawn a User mode task
- To return to User mode after an interrupt or exception3

Both cases are handled by one of two related procedures:

- Place the address in NRP, ensure that the NTSR.CXM bit is set to 1, and execute a **B NRP** instruction to force a context switch to the User mode task.
- Place the desired address in IRP, ensure that the ITSR.CXM bit is set to 1, and execute a **B IRP** instruction to force a context switch to the User mode task.

When returning from an interrupt or exception, the IRP or NRP should already have the correct return address and the ITSR.CXM or NTSR.CXM bit should already be set to 1.

When spawning a user mode task, the appropriate CXM bit and the IRP or NRP will need to be initialized explicitely with the entry point address of the User mode task. In addition, the restricted entry point address register (REP) should be loaded with the desired return address that the User mode task will use when it terminates.

8.4.2 Entering Supervisor Mode from User Mode

The operating mode will change from User mode to Supervisor mode in the following cases:

- While processing any interrupt
- While processing an exception

The User mode task can force a change to Supervisor mode by forcing an exception by executing either an **SWE** or **SWENR** instruction.

The **SWE** and **SWENR** instructions both force a software exception. The **SWE** instruction is used when a return from the exception back to the point of the exception is desired. The **SWENR** instruction is used when a return to the User mode routine is not desired.

OS entry and switching from User to Supervisor mode is accomplished by forcing a software exception using either the **SWE** or **SWENR** instructions. See Section 6.5.3 for information about software exceptions.

Execution of an **SWE** instruction results in an exception being taken before the next execute packet is processed. The return pointer stored in the nonmaskable interrupt return pointer register (NRP) points to this unprocessed packet. The value of the task state register (TSR) is copied to the NMI/exception task state register (NTSR) at the end of the cycle containing the **SWE** instruction, and the interrupt/exception default value is written to TSR. The **SWE** instruction should not be placed in the delay slots of a branch since all instructions behind the **SWE** instruction in the pipe are annulled. All writes to registers in the pipe from instructions executed before and in parallel with the **SWE** instruction will complete before execution of the exception service routine, therefore, the instructions prior to the **SWE** will complete (along with all their delay slots) before the instructions after the **SWE**.

If the **SWE** instruction is executed while in User mode, the mode is changed to Supervisor mode as part of the exception servicing process. The TSR is copied to NTSR, the return address is placed in the NRP register, and a transfer of control is forced to the NMI/Exception vector pointed to by current value of the ISTP. Any code necessary to interpret a User mode request should reside in the exception service routine. After processing the request the exception handler will return control to the user task by executing a **B NRP** command.

The **SWENR** instruction can also be used to terminate a user mode task. The **SWENR** instruction is similar to the **SWE** instruction except that no provision is made for returning to the user mode task and the transfer of control is to the address pointed to by REP instead of the NMI/exception vector. The supervisor mode should have earlier placed the correct address in REP.

Chapter 9
SPRU732J–July 2010

C64x+ CPU Atomic Operations

This chapter describes the instruction and signal additions to the C64x+ CPU enabling atomic operations used in synchronization methods.

The C64x CPU does not support atomic operations. This chapter applies only to the C64x+ CPU.

NOTE: The atomic operations are not supported on all C64x+ devices, see your device-specific data manual for more information.

Topic **Page**

9.1 Synchronization Primitives

This section describes synchronization primitives. These are needed for reliable, high-performance synchronization between multiple processors and can also be important for multitasking on a uniprocessor.

9.1.1 Introduction to Atomic Operations

A synchronization primitive needs to have a few important properties. Often these properties are implemented at least partly in the memory system rather than in the processor. The critical properties are:

- Not blocking. Each access must complete in bounded time, no matter what the other processors are doing. This must be true even if one of the processors is performing the synchronization operations in strange orders or fails to ever complete a series of primitives.
- Indivisible access. A synchronization variable must be updated by only one process at a time and immediately visible to all.
- Powerful. It is advantageous to have synchronization operations that are more powerful than lock. Other atomic operations can be constructed by locking a variable then implementing the more complex operation under that lock. This approach leads to low performance as the critical section is longer and so more contention is to be expected. In the case where there is no contention, this approach is also slower as there are more operations whether contended or not. Other operations desired include atomic counters, atomic list insertion, and atomic buffer insertion. These operations are used frequently by OS and multiprocessor application code.
- Low Latency. If there are no other processors contending for a shared resource, a processor should be able to access it with low latency.
- Fair. All processors contending for a resource should get fair access to it. This may be first-come-first-served, or it may be equal probability of being next. Bad behavior of this sort results in starvation or gross under- or over-representation of some processor in the pattern of resource grant.
- Low Storage Overhead. Support for shared access should be small and have constant size or at worst linear size with the number of processors (or processes) supported.
- Low Traffic. There should be little system bus (interconnect) traffic when accessing shared resources and especially in contending for resources. It is often the case that otherwise sensible shared access synchronization has very high system bus traffic when there is contention over a shared variable.
- Implementation Choices. There should be several implementations possible at varying cost and performance. Often, the higher cost and higher performance implementations make more sense for systems with higher numbers of processors.
- Scalability. The throughput should increase as processors are added. And the latency and bus (interconnect) traffic should not rise too quickly as a practical number of processors are added.
- No/Low Impact to Interrupt or Process Switch. Does not delay or complicate interrupt handling or process switches. In particular, does not add registers to process state or require hardware or software manipulation of GIE and related bits.

9.1.2 Other Memory Operations

The aligned load and store instructions in the C6000 architecture all perform atomic load or store operations on memory. That is, the load from memory or store to memory cannot be subdivided into smaller operations in any way that is detectable by any program. However, those operations are not sufficient to construct reliable synchronization operations using memory variables when a read-modify-write update process is used in the case of multiprocessor or multiple processes might be modifying the same memory location.

By comparison, the **LL**, **SL**, and **CMTL** instructions can be used to atomically update a word of memory. To update, the value must be read, altered and written, without any intervening write by any other task or processor. This is accomplished by monitoring the location and only updating it if it has not been altered.

9.2 Atomic Operations Instructions

The atomic memory access instructions are:

- **LL** (Load Linked). Read a location and begin monitoring it
- **SL** (Store Linked). Buffer data for a linked location.
- **CMTL** (Commit Linked Stores). Stores previously-buffered data into a linked location, if it has not been altered since linked.

Only the .D2 unit can perform a synchronization primitive. The .D1 unit may be used for load, store, arithmetic or logical operations when the .D2 unit is performing a synchronization primitive.

Only one of the **LL**, **SL**, or **CMTL** instructions may be in any one execution packet.

9.2.1 LL Instruction

The **LL** instruction is coded as:

```
LL (.unit) *baseR, dst
```

The **LL** instruction reads a word of memory and prepares to execute an **SL** instruction. The **LL** instruction reads a word from memory just as an **LDW** instruction does; however, as a side effect, a link valid flag is set true and the address is monitored. If any other process stores to that address, the link valid flag is cleared. The link valid flag is also cleared if the **SL** instruction is executed with a different address.

9.2.2 SL Instruction

The **SL** instruction is coded as:

```
SL (.unit) src, *baseR
```

The **SL** instruction buffers a word to be stored to memory by the **CMTL** instruction. It does not commit the change.

If the address of the **SL** instruction does not match the effective address of the most recent **LL** instruction, the link valid flag is cleared.

9.2.3 CMTL Instruction

The **CMTL** instruction is coded as:

```
CMTL (.unit) *baseR, dst
```

The **CMTL** instruction reads the value of the link valid flag. If the link valid flag is true, the data buffered by the **SL** instruction is written to memory.

9.2.4 Valid Sequences of LL, SL, and CMTL Instructions

Because the behavior of the **LL**, **SL**, and **CMTL** instructions are fully defined, they can be executed in any order; however, unless the instruction sequence has an **LL** followed by an **SL** without an intervening **CMTL**, the **SL** data cannot be committed. And unless the **LL**, **SL** instruction pair is followed by a **CMTL** without any intervening **LL**, the **SL** data cannot be committed.

An **LL** instruction may be executed at any time and always begins a new instruction sequence, so any sequence that contains the subsequence of **LL**, **SL**, **CMTL** can succeed.

9.3 Examples of Use

This section shows examples of using the C64x+ CPU synchronization instructions. The instructions can be used to construct a number of common and useful synchronization operations.

9.3.1 Spin Lock Example

Start with a common spin lock shown in Example 9-1. The lock is free, if the lock location contains zero; otherwise, the lock location is set to nonzero value.

Example 9-1. Spin Lock

```
spin_lock:                ; here with lock address in A8 and lock value in A9
          LL *A8, A1      ; load–linked, lock location
          NOP 4
[A1]      B spin_lock     ; if not zero, locked, so spin until I see it unlocked
|| [!A1]  SL A9, *A8      ; store A9 into the lock
[!A1]     CMTL *A8, A1    ; commit the store – no need to be in BR delay
          NOP 4
[!A1]     B spin_lock     ; commit failed, so try again here when have the lock
...
unlock:                   ; here with lock address in A8
         MVK 0, A7        ; zero out A7
         STW A7, *A8      ; zero the lock
```

9.3.2 Shared Accumulator or Counter Example

Example 9-2 shows the use of a shared accumulator or counter to perform synchronization. A shared accumulator contains a sum of values for two or more concurrent processes. A process wishing to update the accumulator reads it with the **LL** instruction, computes a new value by adding its increment to the accumulator, sends the updated value with the **SL** instruction, and commits the value with the **CMTL** instruction. If the commit fails, the update must be retried.

Example 9-2. Shared Counter

```
shared_ctr:               ; here with counter address in A8
                          ; and increment value in A9
         LL *A8, A6       ; load–linked, lock location
         NOP 4
         ADD A6,A9,A6     ; compute the incremented value
         SL A6, *A8       ; new value to store back
         CMTL *A8, A1     ; commit the store
         NOP 4
[!A1]    B shared_ctr     ; commit failed so try again
```

9.3.3 Compare and Swap Example

Example 9-3 shows the use of a shared variable in memory to perform synchronization. A shared variable contains an expected value (or not) for two or more concurrent processes. A process wishing to update the variable reads it with the **LL** instruction, and returns a FALSE value if the value does not match the expected value. If the variable contains the expected value, the process sends an updated value with the **SL** instruction, and commits the value with the **CMTL** instruction. If the commit fails, the test and update must be retried.

Example 9-3. Compare and Swap

```
cas:                       ; here with address in A8
                           ; old (expected) value in A9
                           ; and new (update) value in A10
         LL *A8, A6        ; load-linked, lock location
         NOP 4
         CMPEQ A6, A9, A2  ; compare with expected value
[!A2]    B return_FALSE    ; fail because not equal to old
[A2]     SL A10, *A8       ; update value to store
         NOP 4
         CMTL *A8,A1       ; Commit the stored value
[!A1]    BNOP cas, 5       ; commit failed so try again
```

Instruction Compatibility

The C62x, C64x, and C64x+ DSPs share an instruction set. All of the instructions valid for the C62x DSP are also valid for the C64x and C64x+ DSPs. The C64x DSP adds functionality to the C62x DSP with some unique instructions. Table A-1 lists the instructions that are common to the C62x, C64x, and C64x+ DSPs.

Table A-1. Instruction Compatibility Between C62x, C64x, and C64x+ DSPs

Instruction	C62x DSP	C64x DSP	C64x+ DSP
ABS	✓	✓	✓
ABS2		✓	✓
ADD	✓	✓	✓[1]
ADDAB	✓	✓	✓
ADDAD		✓	✓
ADDAH	✓	✓	✓
ADDAW	✓	✓	✓[1]
ADDK	✓	✓	✓[1]
ADDKPC		✓	✓
ADDSUB			✓
ADDSUB2			✓
ADDU	✓	✓	✓
ADD2	✓	✓	✓
ADD4		✓	✓
AND	✓	✓	✓[1]
ANDN		✓	✓
AVG2		✓	✓
AVGU4		✓	✓
B displacement	✓	✓	✓
B register	✓	✓	✓
B IRP	✓	✓	✓
B NRP	✓	✓	✓
BDEC		✓	✓
BITC4		✓	✓
BITR		✓	✓
BNOP displacement		✓	✓[1]
BNOP register		✓	✓
BPOS		✓	✓
CALLP			✓[1]
CLR	✓	✓	✓[1]
CMPEQ	✓	✓	✓[1]
CMPEQ2		✓	✓
CMPEQ4		✓	✓
CMPGT	✓	✓	✓[1]

[1] Instruction also available in compact form, see Section 3.9.

Table A-1. Instruction Compatibility Between C62x, C64x, and C64x+ DSPs (continued)

Instruction	C62x DSP	C64x DSP	C64x+ DSP
CMPGT2		✓	✓
CMPGTU	✓	✓	✓[1]
CMPGTU4		✓	✓
CMPLT	✓	✓	✓[1]
CMPLT2		✓	✓
CMPLTU	✓	✓	✓[1]
CMPLTU4		✓	✓
CMPY			✓
CMPYR			✓
CMPYR1			✓
CMTL			✓
DDOTP4			✓
DDOTPH2			✓
DDOTPH2R			✓
DDOTPL2			✓
DDOTPL2R			✓
DEAL		✓	✓
DINT			✓
DMV			✓
DOTP2		✓	✓
DOTPN2		✓	✓
DOTPNRSU2		✓	✓
DOTPNRUS2		✓	✓
DOTPRSU2		✓	✓
DOTPRUS2		✓	✓
DOTPSU4		✓	✓
DOTPUS4		✓	✓
DOTPU4		✓	✓
DPACK2			✓
DPACKX2			✓
EXT	✓	✓	✓[2]
EXTU	✓	✓	✓[2]
GMPY			✓
GMPY4		✓	✓
IDLE	✓	✓	✓
LDB	✓	✓	✓[2]
LDB (15-bit offset)	✓	✓	✓[2]
LDBU	✓	✓	✓[2]
LDBU (15-bit offset)	✓	✓	✓
LDDW		✓	✓[2]
LDH	✓	✓	✓[2]
LDH (15-bit offset)	✓	✓	✓
LDHU	✓	✓	✓[2]
LDHU (15-bit offset)	✓	✓	✓
LDNDW		✓	✓[2]
LDNW		✓	✓[2]

[2] Instruction also available in compact form, see Section 3.9.

Table A-1. Instruction Compatibility Between C62x, C64x, and C64x+ DSPs (continued)

Instruction	C62x DSP	C64x DSP	C64x+ DSP
LDW	✓	✓	✓[2]
LDW (15-bit offset)	✓	✓	✓
LL			✓
LMBD	✓	✓	✓
MAX2		✓	✓
MAXU4		✓	✓
MIN2		✓	✓
MINU4		✓	✓
MPY	✓	✓	✓[2]
MPYH	✓	✓	✓[2]
MPYHI		✓	✓
MPYHIR		✓	✓
MPYHL	✓	✓	✓[2]
MPYHLU	✓	✓	✓
MPYHSLU	✓	✓	✓
MPYHSU	✓	✓	✓
MPYHU	✓	✓	✓
MPYHULS	✓	✓	✓
MPYHUS	✓	✓	✓
MPYIH		✓	✓
MPYIHR		✓	✓
MPYIL		✓	✓
MPYILR		✓	✓
MPYLH	✓	✓	✓[3]
MPYLHU	✓	✓	✓
MPYLI		✓	✓
MPYLIR		✓	✓
MPYLSHU	✓	✓	✓
MPYLUHS	✓	✓	✓
MPYSU	✓	✓	✓
MPYSU4		✓	✓
MPYU	✓	✓	✓
MPYU4		✓	✓
MPYUS	✓	✓	✓
MPYUS4		✓	✓
MPY2		✓	✓
MPY2IR			✓
MPY32 (32-bit result)			✓
MPY32 (64-bit result)			✓
MPY32SU			✓
MPY32U			✓
MPY32US			✓
MV	✓	✓	✓[3]
MVC	✓	✓	✓[3]
MVD		✓	✓
MVK	✓	✓	✓[3]

[3] Instruction also available in compact form, see Section 3.9.

Table A-1. Instruction Compatibility Between C62x, C64x, and C64x+ DSPs (continued)

Instruction	C62x DSP	C64x DSP	C64x+ DSP
MVKH	✓	✓	✓
MVKL	✓	✓	✓
MVKLH	✓	✓	✓
NEG	✓	✓	✓[3]
NOP	✓	✓	✓[3]
NORM	✓	✓	✓
NOT	✓	✓	✓
OR	✓	✓	✓[3]
PACK2		✓	✓
PACKH2		✓	✓
PACKH4		✓	✓
PACKHL2		✓	✓
PACKLH2		✓	✓
PACKL4		✓	✓
RINT			✓
ROTL		✓	✓
RPACK2			✓
SADD	✓	✓	✓[4]
SADD2		✓	✓
SADDSUB			✓
SADDSUB2			✓
SADDSU2		✓	✓
SADDUS2		✓	✓
SADDU4		✓	✓
SAT	✓	✓	✓
SET	✓	✓	✓[4]
SHFL		✓	✓
SHFL3			✓
SHL	✓	✓	✓[4]
SHLMB		✓	✓
SHR	✓	✓	✓[4]
SHR2		✓	✓
SHRMB		✓	✓
SHRU	✓	✓	✓[4]
SHRU2		✓	✓
SL			✓
SMPY	✓	✓	✓[4]
SMPYH	✓	✓	✓[4]
SMPYHL	✓	✓	✓[4]
SMPYLH	✓	✓	✓[4]
SMPY2		✓	✓
SMPY32			✓
SPACK2		✓	✓
SPACKU4		✓	✓
SPKERNEL			✓[4]
SPKERNELR			✓

[4] Instruction also available in compact form, see Section 3.9.

Table A-1. Instruction Compatibility Between C62x, C64x, and C64x+ DSPs (continued)

Instruction	C62x DSP	C64x DSP	C64x+ DSP
SPLOOP			✓[4]
SPLOOPD			✓[4]
SPLOOPW			✓
SPMASK			✓[4]
SPMASKR			✓[4]
SSHL	✓	✓	✓[4]
SSHVL		✓	✓
SSHVR		✓	✓
SSUB	✓	✓	✓[4]
SSUB2			✓
STB	✓	✓	✓[4]
STB (15-bit offset)	✓	✓	✓
STDW		✓	✓[4]
STH	✓	✓	✓[4]
STH (15-bit offset)	✓	✓	✓
STNDW		✓	✓[5]
STNW		✓	✓[5]
STW	✓	✓	✓[5]
STW (15-bit offset)	✓	✓	✓[5]
SUB	✓	✓	✓[5]
SUBAB	✓	✓	✓
SUBABS4		✓	✓
SUBAH	✓	✓	✓
SUBAW	✓	✓	✓[5]
SUBC	✓	✓	✓
SUBU	✓	✓	✓
SUB2	✓	✓	✓
SUB4		✓	✓
SWAP2		✓	✓
SWAP4		✓	✓
SWE			✓
SWENR			✓
UNPKHU4		✓	✓
UNPKLU4		✓	✓
XOR	✓	✓	✓[5]
XORMPY			✓
XPND2		✓	✓
XPND4		✓	✓
ZERO	✓	✓	✓

[5] Instruction also available in compact form, see Section 3.9.

Mapping Between Instruction and Functional Unit

Table B-1 lists the instructions that execute on each functional unit.

Table B-1. Instruction to Functional Unit Mapping

	Functional Unit			
Instruction	**.L Unit**	**.M Unit**	**.S Unit**	**.D Unit**
ABS	✓			
ABS2	✓			
ADD	✓		✓	✓
ADDAB				✓
ADDAD				✓
ADDAH				✓
ADDAW				✓
ADDK			✓	
ADDKPC			✓[1]	
ADDSUB [2]	✓			
ADDSUB2 [2]	✓			
ADDU	✓			
ADD2	✓		✓	✓
ADD4	✓			
AND	✓		✓	✓
ANDN	✓		✓	✓
AVG2		✓		
AVGU4		✓		
B displacement			✓	
B register			✓[1]	
B IRP			✓[1]	
B NRP			✓[1]	
BDEC			✓	
BITC4		✓		
BITR		✓		
BNOP displacement			✓	
BNOP register			✓	
BPOS			✓	
CALLP [2]			✓	
CLR			✓	
CMPEQ	✓			
CMPEQ2			✓	
CMPEQ4			✓	
CMPGT	✓			
CMPGT2			✓	

[1] S2 only
[2] C64x+ CPU-specific instruction

Table B-1. Instruction to Functional Unit Mapping (continued)

	Functional Unit			
Instruction	.L Unit	.M Unit	.S Unit	.D Unit
CMPGTU	✓			
CMPGTU4			✓	
CMPLT	✓			
CMPLT2			✓	
CMPLTU	✓			
CMPLTU4			✓	
CMPY [2]		✓		
CMPYR [2]		✓		
CMPYR1 [2]		✓		
CMTL [2]				✓[3]
DDOTP4 [2]		✓		
DDOTPH2 [2]		✓		
DDOTPH2R [2]		✓		
DDOTPL2 [2]		✓		
DDOTPL2R [4]		✓		
DEAL		✓		
DINT [4]	No unit			
DMV [4]			✓	
DOTP2		✓		
DOTPN2		✓		
DOTPNRSU2		✓		
DOTPNRUS2		✓		
DOTPRSU2		✓		
DOTPRUS2		✓		
DOTPSU4		✓		
DOTPUS4		✓		
DOTPU4		✓		
DPACK2 [4]	✓			
DPACKX2 [4]	✓			
EXT			✓	
EXTU			✓	
GMPY [4]		✓		
GMPY4		✓		
IDLE	No unit			
LDB				✓
LDB (15-bit offset)				✓[5]
LDBU				✓
LDBU (15-bit offset)				✓[5]
LDDW				✓
LDH				✓
LDH (15-bit offset)				✓[5]
LDHU				✓
LDHU (15-bit offset)				✓[5]
LDNDW				✓

[3] D2 only
[4] C64x+ CPU-specific instruction
[5] D2 only

Table B-1. Instruction to Functional Unit Mapping (continued)

	Functional Unit			
Instruction	.L Unit	.M Unit	.S Unit	.D Unit
LDNW				✓
LDW				✓
LDW (15-bit offset)				✓[5]
LL [4]				✓[5]
LMBD	✓			
MAX2	✓		✓[4]	
MAXU4	✓			
MIN2	✓		✓[4]	
MINU4	✓			
MPY		✓		
MPYH		✓		
MPYHI		✓		
MPYHIR		✓		
MPYHL		✓		
MPYHLU		✓		
MPYHSLU		✓		
MPYHSU		✓		
MPYHU		✓		
MPYHULS		✓		
MPYHUS		✓		
MPYIH		✓		
MPYIHR		✓		
MPYIL		✓		
MPYILR		✓		
MPYLH		✓		
MPYLHU		✓		
MPYLI		✓		
MPYLIR		✓		
MPYLSHU		✓		
MPYLUHS		✓		
MPYSU		✓		
MPYSU4		✓		
MPYU		✓		
MPYU4		✓		
MPYUS		✓		
MPYUS4		✓		
MPY2		✓		
MPY2IR [6]		✓		
MPY32 (32-bit result) [6]		✓		
MPY32 (64-bit result) [6]		✓		
MPY32SU [6]		✓		
MPY32U [6]		✓		
MPY32US [6]		✓		
MV	✓		✓	✓
MVC			✓[7]	

[6] C64x+ CPU-specific instruction
[7] S2 only

Table B-1. Instruction to Functional Unit Mapping (continued)

	Functional Unit			
Instruction	**.L Unit**	**.M Unit**	**.S Unit**	**.D Unit**
MVD		✓		
MVK	✓		✓	✓
MVKH			✓	
MVKL			✓	
MVKLH			✓	
NEG	✓		✓	
NOP	No unit			
NORM	✓			
NOT	✓		✓	✓
OR	✓		✓	✓
PACK2	✓		✓	
PACKH2	✓		✓	
PACKH4	✓			
PACKHL2	✓		✓	
PACKLH2	✓		✓	
PACKL4	✓			
RINT [8]	No unit			
ROTL		✓		
RPACK2 [8]			✓	
SADD	✓		✓	
SADD2			✓	
SADDSUB [8]	✓			
SADDSUB2 [8]	✓			
SADDSU2			✓	
SADDUS2			✓	
SADDU4			✓	
SAT	✓			
SET			✓	
SHFL		✓		
SHFL3 [8]	✓			
SHL			✓	
SHLMB	✓		✓	
SHR			✓	
SHR2			✓	
SHRMB	✓		✓	
SHRU			✓	
SHRU2			✓	
SL [8]				✓[9]
SMPY		✓		
SMPYH		✓		
SMPYHL		✓		
SMPYLH		✓		
SMPY2		✓		
SMPY32 [8]		✓		
SPACK2			✓	

[8] C64x+ CPU-specific instruction
[9] D2 only

Table B-1. Instruction to Functional Unit Mapping (continued)

	Functional Unit			
Instruction	**.L Unit**	**.M Unit**	**.S Unit**	**.D Unit**
SPACKU4			✓	
SPKERNEL [8]	No unit			
SPKERNELR [8]	No unit			
SPLOOP [8]	No unit			
SPLOOPD [8]	No unit			
SPLOOPW [8]	No unit			
SPMASK [8]	No unit			
SPMASKR [8]	No unit			
SSHL			✓	
SSHVL		✓		
SSHVR		✓		
SSUB	✓			
SSUB2 [8]	✓			
STB				✓
STB (15-bit offset)				✓[9]
STDW				✓
STH				✓
STH (15-bit offset)				✓[10]
STNDW				✓
STNW				✓
STW				✓
STW (15-bit offset)				✓[10]
SUB	✓		✓	✓
SUBAB				✓
SUBABS4	✓			
SUBAH				✓
SUBAW				✓
SUBC	✓			
SUBU	✓			
SUB2	✓		✓	✓
SUB4	✓			
SWAP2	✓		✓	
SWAP4	✓			
SWE [11]	No unit			
SWENR [11]	No unit			
UNPKHU4	✓		✓	
UNPKLU4	✓		✓	
XOR	✓		✓	✓
XORMPY [11]		✓		
XPND2		✓		
XPND4		✓		
ZERO	✓		✓	✓

[10] D2 only
[11] C64x+ CPU-specific instruction

.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	Format	Instruction	Format
ADD	Figure C-1, Figure C-2	LL [(1)]	Figure C-4
ADDAB	Figure C-1, Figure C-3	MV	Figure C-1, Figure C-2
ADDAD	Figure C-1	MVK	Figure C-1
ADDAH	Figure C-1, Figure C-3	NOT	Figure C-2
ADDAW	Figure C-1, Figure C-3	OR	Figure C-2
ADD2	Figure C-2	SL [(1)]	Figure C-4
AND	Figure C-2	STB	Figure C-5
ANDN	Figure C-2	STB (15-bit offset) [(1)]	Figure C-6
CMTL [(1)]	Figure C-4	STDW	Figure C-7
LDB	Figure C-5	STH	Figure C-5
LDB (15-bit offset) [(1)]	Figure C-6	STH (15-bit offset) [(1)]	Figure C-6
LDBU	Figure C-5	STNDW	Figure C-8
LDBU (15-bit offset) [(1)]	Figure C-6	STNW	Figure C-5
LDDW	Figure C-7	STW	Figure C-5
LDH	Figure C-5	STW (15-bit offset) [(1)]	Figure C-6
LDH (15-bit offset) [(1)]	Figure C-6	SUB	Figure C-1, Figure C-2
LDHU	Figure C-5	SUBAB	Figure C-1
LDHU (15-bit offset) [(1)]	Figure C-6	SUBAH	Figure C-1
LDNDW	Figure C-8	SUBAW	Figure C-1
LDNW	Figure C-5	SUB2	Figure C-2
LDW	Figure C-5	XOR	Figure C-2
LDW (15-bit offset) [(1)]	Figure C-6	ZERO	Figure C-1, Figure C-2

[(1)] 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.
dw	doubleword; 0 = word, 1 = doubleword
ld/st	load or store; 0 = store, 1 = load
mode	addressing mode, see Table C-3
na	nonaligned; 0 = aligned, 1 = nonaligned
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
ptr	offset from either A4-A7 or B4-B7 depending on the value of the *s* bit. The *ptr* field is the 2 least-significant bits of the *src2* (*baseR*) field—bit 2 of register address is forced to 1.
r	LDDW/LDNDW/LDNW instruction

Table C-2. .D Unit Opcode Map Symbol Definitions (continued)

Symbol	Meaning
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
sz	data size select; 0 = primary size, 1 = secondary size (see Section 3.9.2.2)
t	side of source/destination (*src/dst*) register; 0 = side A, 1 = side B
ucst$_n$	bit n of the unsigned constant field
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 C64x CPU and C64x+ CPU 32-bit opcodes used in the .D unit are mapped in the following figures.

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. ADDAB/ADDAH/ADDAW Long-Immediate Operations

31	30	29	28	27 23	22 8	7	6 4	3	2	1	0
0	0	0	1	*dst*	*offsetR*	*y*	*op*	1	1	*s*	*p*
				5	15	1	3			1	1

Figure C-4. Linked Word Operations

31 29	28	27 23	22 18	17	16	15	14	13	12	11	10	9 7	6	5	4	3	2	1	0
creg	*z*	*src/dst*	*baseR*	0	0	0	0	0	0	0	1	*op*	1	0	0	0	0	1	*p*
3	1	5	5									3							1

Figure C-5. 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*	0	1	*s*	*p*
3	1	5	5	5	4	1	1	3			1	1

Figure C-6. Load/Store Long-Immediate Operations

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

Figure C-7. Load/Store Doubleword Instruction Format

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

Figure C-8. Load/Store Nonaligned Doubleword Instruction Format

31 29	28	27 24	23	22 18	17 13	12 9	8	7	6 4	3	2	1	0
creg	*z*	*src/dst*	*sc*	*baseR*	*offsetR*	*mode*	1	*y*	*op*	0	1	*s*	*p*
3	1	4	1	5	5	4		1	3			1	1

C.4 16-Bit Opcode Maps

The C64x+ CPU 16-bit opcodes used in the .D unit for compact instructions are mapped in the following figures. See Section 3.9 for more information about compact instructions.

Figure C-9. Doff4 Instruction Format

15 13	12	11	10	9	8 7	6 4	3	2	1	0
ucst$_{2-0}$	*t*	*ucst*$_{3}$	0	*sz*	*ptr*	*src/dst*	*ld/st*	1	0	*s*
3	1	1		1	2	3	1			1

DSZ			*sz*	*ld/st*	Mnemonic
0	x	x	0	0	**STW** (.unit) *src*, **ptr[ucst4]*
0	x	x	0	1	**LDW** (.unit) **ptr[ucst4], dst*
0	0	0	1	0	**STB** (.unit) *src*, **ptr[ucst4]*
0	0	0	1	1	**LDBU** (.unit) **ptr[ucst4], dst*
0	0	1	1	0	**STB** (.unit) *src*, **ptr[ucst4]*
0	0	1	1	1	**LDB** (.unit) **ptr[ucst4], dst*
0	1	0	1	0	**STH** (.unit) *src*, **ptr[ucst4]*
0	1	0	1	1	**LDHU** (.unit) **ptr[ucst4], dst*
0	1	1	1	0	**STH** (.unit) *src*, **ptr[ucst4]*
0	1	1	1	1	**LDH** (.unit) **ptr[ucst4], dst*
1	0	0	1	0	**STW** (.unit) *src*, **ptr[ucst4]*
1	0	0	1	1	**LDW** (.unit) **ptr[ucst4], dst*
1	0	1	1	0	**STB** (.unit) *src*, **ptr[ucst4]*
1	0	1	1	1	**LDB** (.unit) **ptr[ucst4], dst*
1	1	0	1	0	**STNW** (.unit) *src*, **ptr[ucst4]*
1	1	0	1	1	**LDNW** (.unit) **ptr[ucst4], dst*
1	1	1	1	0	**STH** (.unit) *src*, **ptr[ucst4]*
1	1	1	1	1	**LDH** (.unit) **ptr[ucst4], dst*

Figure C-10. Doff4DW Instruction Format

15 13	12	11	10	9	8 7	6 5	4	3	2	1	0
ucst$_{2-0}$	*t*	*ucst*$_{3}$	0	*sz*	*ptr*	*src/dst*	*na*	*ld/st*	1	0	*s*
3	1	1		1	2	2	1	1			1

NOTE: *src/dst* register address formed from op6:op5:0 (even registers)

DSZ			*sz*	*ld/st*	*na*	Mnemonic
1	x	x	0	0	0	**STDW** (.unit) *src*, **ptr[ucst4]*
1	x	x	0	1	0	**LDDW** (.unit) **ptr[ucst4], dst*
1	x	x	0	0	1	**STNDW** (.unit) *src*, **ptr[ucst4]* (ucst4 unscaled only)
1	x	x	0	1	1	**LDNDW** (.unit) **ptr[ucst4], dst* (ucst4 unscaled only)

Figure C-11. Dind Instruction Format

15 13	12	11	10	9	8 7	6 4	3	2	1	0
src1	*t*	0	1	*sz*	*ptr*	*src/dst*	*ld/st*	1	0	*s*
3	1			1	2	3	1			1

Opcode map field used...	For operand type...
src1	sint

DSZ			*sz*	*ld/st*	Mnemonic
0	x	x	0	0	**STW** (.unit) *src*, **ptr[src1]*
0	x	x	0	1	**LDW** (.unit) **ptr[src1], dst*
0	0	0	1	0	**STB** (.unit) *src*, **ptr[src1]*
0	0	0	1	1	**LDBU** (.unit) **ptr[src1], dst*
0	0	1	1	0	**STB** (.unit) *src*, **ptr[src1]*
0	0	1	1	1	**LDB** (.unit) **ptr[src1], dst*
0	1	0	1	0	**STH** (.unit) *src*, **ptr[src1]*
0	1	0	1	1	**LDHU** (.unit) **ptr[src1], dst*
0	1	1	1	0	**STH** (.unit) *src*, **ptr[src1]*
0	1	1	1	1	**LDH** (.unit) **ptr[src1], dst*
1	0	0	1	0	**STW** (.unit) *src*, **ptr[src1]*
1	0	0	1	1	**LDW** (.unit) **ptr[src1], dst*
1	0	1	1	0	**STB** (.unit) *src*, **ptr[src1]*
1	0	1	1	1	**LDB** (.unit) **ptr[src1], dst*
1	1	0	1	0	**STNW** (.unit) *src*, **ptr[src1]*
1	1	0	1	1	**LDNW** (.unit) **ptr[src1], dst*
1	1	1	1	0	**STH** (.unit) *src*, **ptr[src1]*
1	1	1	1	1	**LDH** (.unit) **ptr[src1], dst*

Figure C-12. DindDW Instruction Format

15 13	12	11	10	9	8 7	6 5	4	3	2	1	0
src1	*t*	0	1	*sz*	*ptr*	*src/dst*	*na*	*ld/st*	1	0	*s*
3	1			1	2	2	1	1			1

NOTE: *src/dst* register address formed from op6:op5:0 (even registers)

Opcode map field used...	**For operand type...**
src1	sint

DSZ			*sz*	*ld/st*	*na*	**Mnemonic**
1	x	x	0	0	0	**STDW** (.unit) *src*, **ptr[src1]*
1	x	x	0	1	0	**LDDW** (.unit) **ptr[src1], dst*
1	x	x	0	0	1	**STNDW** (.unit) *src*, **ptr[src1]* (src1 unscaled only)
1	x	x	0	1	1	**LDNDW** (.unit) **ptr[src1], dst* (src1 unscaled only)

Figure C-13. Dinc Instruction Format

15	14	13	12	11	10	9	8 7	6 4	3	2	1	0
0	0	*ucst*$_0$	*t*	1	1	*sz*	*ptr*	*src/dst*	*ld/st*	1	0	*s*
		1	1			1	2	3	1			1

NOTE: *ucst2* = *ucst*$_0$ + 1

DSZ			*sz*	*ld/st*	**Mnemonic**
0	x	x	0	0	**STW** (.unit) *src*, **ptr[ucst2]++*
0	x	x	0	1	**LDW** (.unit) **ptr[ucst2]++, dst*
0	0	0	1	0	**STB** (.unit) *src*, **ptr[ucst2]++*
0	0	0	1	1	**LDBU** (.unit) **ptr[ucst2]++, dst*
0	0	1	1	0	**STB** (.unit) *src*, **ptr[ucst2]++*
0	0	1	1	1	**LDB** (.unit) **ptr[ucst2]++, dst*
0	1	0	1	0	**STH** (.unit) *src*, **ptr[ucst2]++*
0	1	0	1	1	**LDHU** (.unit) **ptr[ucst2]++, dst*
0	1	1	1	0	**STH** (.unit) *src*, **ptr[ucst2]++*
0	1	1	1	1	**LDH** (.unit) **ptr[ucst2]++, dst*
1	0	0	1	0	**STW** (.unit) *src*, **ptr[ucst2]++*
1	0	0	1	1	**LDW** (.unit) **ptr[ucst2]++, dst*
1	0	1	1	0	**STB** (.unit) *src*, **ptr[ucst2]++*
1	0	1	1	1	**LDB** (.unit) **ptr[ucst2]++, dst*
1	1	0	1	0	**STNW** (.unit) *src*, **ptr[ucst2]++*
1	1	0	1	1	**LDNW** (.unit) **ptr[ucst2]++, dst*
1	1	1	1	0	**STH** (.unit) *src*, **ptr[ucst2]++*
1	1	1	1	1	**LDH** (.unit) **ptr[ucst2]++, dst*

Figure C-14. DincDW Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	*ucst*$_0$	*t*	1	1	*sz*	*ptr*		*src/dst*		*na*	*ld/st*	1	0	*s*
		1	1			1	2		2		1	1			1

NOTES:

1. *ucst2* = *ucst*$_0$ + 1
2. *src/dst* register address formed from op6:op5:0 (even registers)

DSZ			*sz*	*ld/st*	*na*	**Mnemonic**
1	x	x	0	0	0	**STDW** (.unit) *src*, **ptr[ucst2]*++
1	x	x	0	1	0	**LDDW** (.unit) **ptr[ucst2]*++, *dst*
1	x	x	0	0	1	**STNDW** (.unit) *src*, **ptr[ucst2]*++ (ucst2 scaled only)
1	x	x	0	1	1	**LDNDW** (.unit) **ptr[ucst2]*++, *dst* (ucst2 scaled only)

Figure C-15. Ddec Instruction Format

15	14	13	12	11	10	9	8	7	6		4	3	2	1	0
0	1	*ucst*$_0$	*t*	1	1	*sz*	*ptr*		*src/dst*			*ld/st*	1	0	*s*
		1	1			1	2		3			1			1

NOTE: *ucst2* = *ucst*$_0$ + 1

DSZ			*sz*	*ld/st*	**Mnemonic**
0	x	x	0	0	**STW** (.unit) *src*, **--ptr[ucst2]*
0	x	x	0	1	**LDW** (.unit) **--ptr[ucst2], dst*
0	0	0	1	0	**STB** (.unit) *src*, **--ptr[ucst2]*
0	0	0	1	1	**LDBU** (.unit) **--ptr[ucst2], dst*
0	0	1	1	0	**STB** (.unit) *src*, **--ptr[ucst2]*
0	0	1	1	1	**LDB** (.unit) **--ptr[ucst2], dst*
0	1	0	1	0	**STH** (.unit) *src*, **--ptr[ucst2]*
0	1	0	1	1	**LDHU** (.unit) **--ptr[ucst2], dst*
0	1	1	1	0	**STH** (.unit) *src*, **--ptr[ucst2]*
0	1	1	1	1	**LDH** (.unit) **--ptr[ucst2], dst*
1	0	0	1	0	**STW** (.unit) *src*, **--ptr[ucst2]*
1	0	0	1	1	**LDW** (.unit) **--ptr[ucst2], dst*
1	0	1	1	0	**STB** (.unit) *src*, **--ptr[ucst2]*
1	0	1	1	1	**LDB** (.unit) **--ptr[ucst2], dst*
1	1	0	1	0	**STNW** (.unit) *src*, **--ptr[ucst2]*
1	1	0	1	1	**LDNW** (.unit) **--ptr[ucst2], dst*
1	1	1	1	0	**STH** (.unit) *src*, **--ptr[ucst2]*
1	1	1	1	1	**LDH** (.unit) **--ptr[ucst2], dst*

Figure C-16. DdecDW Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	*ucst*$_0$	*t*	1	1	*sz*	*ptr*		*src/dst*		*na*	*ld/st*	1	0	*s*
		1	1			1	2		2		1	1			1

NOTES:

1. *ucst2* = *ucst*$_0$ + 1
2. *src/dst* register address formed from op6:op5:0 (even registers)

DSZ			*sz*	*ld/st*	*na*	**Mnemonic**
1	x	x	0	0	0	**STDW** (.unit) *src*, **--ptr[ucst2]*
1	x	x	0	1	0	**LDDW** (.unit) **--ptr[ucst2], dst*
1	x	x	0	0	1	**STNDW** (.unit) *src*, **--ptr[ucst2]* (ucst2 scaled only)
1	x	x	0	1	1	**LDNDW** (.unit) **--ptr[ucst2], dst* (ucst2 scaled only)

Figure C-17. Dstk Instruction Format

15	14	13	12	11	10	9		7	6		4	3	2	1	0
1	*ucst*$_{1-0}$		*t*	1	1	*ucst*$_{4-2}$			*src/dst*			*ld/st*	1	0	*s*
	2		1			3			3			1			1

NOTE: ptr = B15, s = 1

ld/st	**Mnemonic**
0	**STW** (.unit) *src*,*B15*[ucst5]*
1	**LDW** (.unit)*B15*[ucst5], dst*

Figure C-18. Dx2op Instruction Format

15		13	12	11	10	9		7	6	5	4	3	2	1	0
src1/dst			x	*op*	0	*src2*			0	1	1	0	1	1	*s*
3			1	1		3									1

Opcode map field used...	**For operand type...**
src1/dst	sint
src2	xsint

op	**Mnemonic**
0	**ADD** (.unit) *src1*, *src2*, *dst* (src1 = dst)
1	**SUB** (.unit) *src1*, *src2*, *dst* (src1 = dst, dst = src1 - src2

Figure C-19. Dx5 Instruction Format

15 13	12 11	10	9 7	6	5	4	3	2	1	0
ucst$_{2\text{-}0}$	*ucst*$_{4\text{-}3}$	1	*dst*	0	1	1	0	1	1	*s*
3	2		3							1

NOTE: src2 = B15

Opcode map field used...	For operand type...
dst	sint

Mnemonic
ADDAW (.unit)B15, *ucst5, dst*

Figure C-20. Dx5p Instruction Format

15 13	12	11	10	9 8	7	6	5	4	3	2	1	0
ucst$_{2\text{-}0}$	0	1	1	*ucst*$_{4\text{-}3}$	*op*	1	1	1	0	1	1	1
3				2	1							s = 1

NOTE: src2 = dst = B15

op	Mnemonic
0	**ADDAW** (.unit)B15, *ucst5,* B15
1	**SUBAW** (.unit)B15, *ucst5,* B15

Figure C-21. Dx1 Instruction Format

15 13	12	11	10	9 7	6	5	4	3	2	1	0
op	1	1	0	*src2/dst*	1	1	1	0	1	1	*s*
3				3							1

Opcode map field used...	For operand type...
src2/dst	sint

	op		Mnemonic
0	0	0	see LSDx1, Figure G-4
0	0	1	see LSDx1, Figure G-4
0	1	0	Reserved
0	1	1	**SUB** (.unit) *src2,* 1, *dst* (src2 = dst, dst = src2 - 1)
1	0	0	Reserved
1	0	1	see LSDx1, Figure G-4
1	1	0	Reserved
1	1	1	see LSDx1, Figure G-4

Figure C-22. Dpp Instruction Format

15	14	13	12	11	10 … 7	6	5	4	3	2	1	0
dw	*dl/st*	*ucst*$_0$	*t*	0	*src/dst*	1	1	1	0	1	1	1
1	1	1	1		4							

NOTES:

1. ptr = B15
2. *ucst2* = *ucst*$_0$ + 1
3. src/dst is from A0-A15, B0-B15
4. RS header bit is ignored

dw	*ld/st*	**Mnemonic**
0	0	**STW** (.unit) *src*,*B15--*[ucst2]*
0	1	**LDW** (.unit)*++B15*[ucst2], dst*
1	0	**STDW** (.unit) *src*,*B15--*[ucst2]*
1	1	**LDDW** (.unit)*++B15*[ucst2], dst*

t	*src/dst*	**Source/Destination**	*t*	*src/dst*	**Source/Destination**
0	0000	A0	1	0000	B0
0	0001	A1	1	0001	B1
0	0010	A2	1	0010	B2
0	0011	A3	1	0011	B3
0	0100	A4	1	0100	B4
0	0101	A5	1	0101	B5
0	0110	A6	1	0110	B6
0	0111	A7	1	0111	B7
0	1000	A8	1	1000	B8
0	1001	A9	1	1001	B9
0	1010	A10	1	1010	B10
0	1011	A11	1	1011	B11
0	1100	A12	1	1100	B12
0	1101	A13	1	1101	B13
0	1110	A14	1	1110	B14
0	1111	A15	1	1111	B15

.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	Format	Instruction	Format
ABS	Figure D-1	PACK2	Figure D-1
ABS2	Figure D-2	PACKH2	Figure D-1
ADD	Figure D-1	PACKH4	Figure D-1
ADDSUB [1]	Figure D-3	PACKHL2	Figure D-1
ADDSUB2 [1]	Figure D-3	PACKLH2	Figure D-1
ADDU	Figure D-1	PACKL4	Figure D-1
ADD2	Figure D-1	SADD	Figure D-1
ADD4	Figure D-1	SADDSUB [1]	Figure D-3
AND	Figure D-1	SADDSUB2 [1]	Figure D-3
ANDN	Figure D-1	SAT	Figure D-1
CMPEQ	Figure D-1	SHFL3 [1]	Figure D-3
CMPGT	Figure D-1	SHLMB	Figure D-1
CMPGTU	Figure D-1	SHRMB	Figure D-1
CMPLT	Figure D-1	SSUB	Figure D-1
CMPLTU	Figure D-1	SSUB2[1]	Figure D-1
DPACK2[1]	Figure D-3	SUB	Figure D-1
DPACKX2[1]	Figure D-3	SUBABS4	Figure D-1
LMBD	Figure D-1	SUBC	Figure D-1
MAX2	Figure D-1	SUBU	Figure D-1
MAXU4	Figure D-1	SUB2	Figure D-1
MIN2	Figure D-1	SUB4	Figure D-1
MINU4	Figure D-1	SWAP2	Figure D-1
MV	Figure D-1	SWAP4	Figure D-2
MVK	Figure D-2	UNPKHU4	Figure D-2
NEG	Figure D-1	UNPKLU4	Figure D-2
NORM	Figure D-1	XOR	Figure D-1
NOT	Figure D-1	ZERO	Figure D-1
OR	Figure D-1		

[1] C64x+ DSP-specific instruction

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
cstn	n-bit constant field
dst	destination
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
s	side A or B for destination; 0 = side A, 1 = side B
scst$_n$	bit n of the signed constant field
sn	sign
src1	source 1
src2	source 2
ucstn	n-bit unsigned constant field
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 C64x CPU and C64x+ CPU 32-bit opcodes used in the .L unit are mapped in the following figures.

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. Unary Instruction Format

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

Figure D-3. 1 or 2 Sources, Nonconditional Instruction Format

31	30	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

D.4 16-Bit Opcode Maps

The C64x+ CPU 16-bit opcodes used in the .L unit for compact instructions are mapped in the following figures. See Section 3.9 for more information about compact instructions.

Figure D-4. L3 Instruction Format

15 13	12	11	10	9 7	6 4	3	2	1	0
src1	x	*op*	0	*src2*	*dst*	0	0	0	*s*
3	1	1		3	3				1

Opcode map field used...	**For operand type...**
src1	sint
src2	xsint
dst	sint

op	**SAT**	**Mnemonic**
0	0	**ADD** (.unit) *src1*, *src2*, *dst*
0	1	**SADD** (.unit) *src1*, *src2*, *dst*
1	0	**SUB** (.unit) *src1*, *src2*, *dst* (dst = src1 - src2)
1	1	**SSUB** (.unit) *src1*, *src2*, *dst* (dst = src1 - src2)

Figure D-5. L3i Instruction Format

15 13	12	11	10	9 7	6 4	3	2	1	0
cst3	x	*sn*	1	*src2*	*dst*	0	0	0	*s*
3	1	1		3	3				1

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

		32-Bit Opcode *cst* Equivalent				**32-Bit Opcode *cst* Equivalent**	
sn	*cst3*	*scst5*	**Decimal Value**	*sn*	*cst3*	*scst5*	**Decimal Value**
0	000	01000	8	1	000	11000	-8
0	001	00001	1	1	001	11001	-7
0	010	00010	2	1	010	11010	-6
0	011	00011	3	1	011	11011	-5
0	100	00100	4	1	100	11100	-4
0	101	00101	5	1	101	11101	-3
0	110	00110	6	1	110	11110	-2
0	111	00111	7	1	111	11111	-1

Mnemonic
ADD (.unit) *scst5*, *src2, dst*

Figure D-6. Ltbd Instruction Format

15	14	13	12	11	10	9 7	6	5	4	3	2	1	0
			x		0	*src2*				1	0	0	*s*
			1			3							1

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

Figure D-7. L2c Instruction Format

15 13	12	11	10	9 7	6 5	4	3	2	1	0
src1	x	*op*$_2$	1	*src2*	*op*$_{1-0}$	*dst*	1	0	0	*s*
3	1	1		3	2	1				1

NOTE: *dst* = A0, A1 or B0, B1 as selected by *dst* and *s*

Opcode map field used...	For operand type...
src1	sint
src2	xsint

op			Mnemonic
0	0	0	**AND** (.unit) *src1*, *src2*, *dst*
0	0	1	**OR** (.unit) *src1*, *src2*, *dst*
0	1	0	**XOR** (.unit) *src1*, *src2*, *dst*
0	1	1	**CMPEQ** (.unit) *src1*, *src2*, *dst*
1	0	0	**CMPLT** (.unit) *src1*, *src2*, *dst* (dst = src1 < src2 , signed compare)
1	0	1	**CMPGT** (.unit) *src1*, *src2*, *dst* (dst = src1 > src2 , signed compare)
1	1	0	**CMPLTU** (.unit) *src1*, *src2*, *dst* (dst = src1 < src2 , unsigned compare)
1	1	1	**CMPGTU** (.unit) *src1*, *src2*, *dst* (dst = src1 > src2 , unsigned compare)

Figure D-8. Lx5 Instruction Format

15 13	12 11	10	9 7	6	5	4	3	2	1	0
scst$_{2-0}$	*scst*$_{4-3}$	1	*dst*	0	1	0	0	1	1	*s*
3	2		3							1

Opcode map field used...	For operand type...
dst	sint

Mnemonic
MVK (.unit) *scst5*, *dst*

Figure D-9. Lx3c Instruction Format

15 13	12	11	10	9 7	6	5	4	3	2	1	0
ucst3	0	*dst*	0	*src2*	0	1	0	0	1	1	*s*
3		1		3							1

NOTE: *dst* = A0, A1 or B0, B1 as selected by *dst* and *s*

Opcode map field used...	For operand type...
src2	sint

Mnemonic
CMPEQ (.unit) *ucst3*, *src2, dst*

Figure D-10. Lx1c Instruction Format

15	14	13	12	11	10	9		7	6	5	4	3	2	1	0
op		*ucst1*	1	*dst*	0	*src2*			0	1	0	0	1	1	*s*
2		1		1		3									1

NOTE: *dst* = A0, A1 or B0, B1 as selected by *dst* and *s*

Opcode map field used...	For operand type...
src2	sint

op		Mnemonic
0	0	**CMPLT** (.unit) *ucst1, src2*, *dst* (dst = ucst1 < src2 , signed compare)
0	1	**CMPGT** (.unit) *ucst1, src2*, *dst* (dst = ucst1 > src2 , signed compare)
1	0	**CMPLTU** (.unit) *ucst1, src2*, *dst* (dst = ucst1 < src2 , unsigned compare)
1	1	**CMPGTU** (.unit) *ucst1, src2*, *dst* (dst = ucst1 > src2 , unsigned compare)

Figure D-11. Lx1 Instruction Format

15		13	12	11	10	9		7	6	5	4	3	2	1	0
op			1	1	0	*src2/dst*			1	1	0	0	1	1	*s*
3						3									1

op			Mnemonic
0	0	0	see LSDx1, Figure G-4
0	0	1	see LSDx1, Figure G-4
0	1	0	**SUB** (.unit)0*, src2, dst* (src2 = dst; dst = 0 - src2)
0	1	1	**ADD** (.unit)-1*, src2, dst* (src2 = dst)
1	0	0	Reserved
1	0	1	see LSDx1, Figure G-4
1	1	0	Reserved
1	1	1	see LSDx1, Figure G-4

.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	Format	Instruction	Format
AVG2	Figure E-1	MPYIHR	Figure E-1
AVGU4	Figure E-1	MPYIL	Figure E-1
BITC4	Figure E-2	MPYILR	Figure E-1
BITR	Figure E-2	MPYLH	Figure E-4
CMPY [(1)]	Figure E-3	MPYLHU	Figure E-4
CMPYR [(1)]	Figure E-3	MPYLI	Figure E-1
CMPYR1 [(1)]	Figure E-3	MPYLIR	Figure E-1
DDOTP4 [(1)]	Figure E-3	MPYLSHU	Figure E-4
DDOTPH2 [(1)]	Figure E-3	MPYLUHS	Figure E-4
DDOTPH2R [(1)]	Figure E-3	MPYSU	Figure E-4
DDOTPL2 [(1)]	Figure E-3	MPYSU4	Figure E-1
DDOTPL2R [(1)]	Figure E-3	MPYU	Figure E-4
DEAL	Figure E-2	MPYU4	Figure E-1
DOTP2	Figure E-1	MPYUS	Figure E-4
DOTPN2	Figure E-1	MPYUS4	Figure E-1
DOTPNRSU2	Figure E-1	MPY2	Figure E-1
DOTPNRUS2	Figure E-1	MPY2IR[(1)]	Figure E-1
DOTPRSU2	Figure E-1	MPY32 (32-bit result)[(1)]	Figure E-4
DOTPRUS2	Figure E-1	MPY32 (64-bit result)[(1)]	Figure E-4
DOTPSU4	Figure E-1	MPY32SU[(1)]	Figure E-4
DOTPUS4	Figure E-1	MPY32U[(1)]	Figure E-1
DOTPU4	Figure E-1	MPY32US[(1)]	Figure E-1
GMPY[(1)]	Figure E-3	MVD	Figure E-2
GMPY4	Figure E-1	ROTL	Figure E-1
MPY	Figure E-4	SHFL	Figure E-2
MPYH	Figure E-4	SMPY	Figure E-4
MPYHI	Figure E-1	SMPYH	Figure E-4
MPYHIR	Figure E-1	SMPYHL	Figure E-4
MPYHL	Figure E-4	SMPYLH	Figure E-4
MPYHLU	Figure E-4	SMPY2	Figure E-1
MPYHSLU	Figure E-4	SMPY32[(1)]	Figure E-3
MPYHSU	Figure E-4	SSHVL	Figure E-1
MPYHU	Figure E-4	SSHVR	Figure E-1
MPYHULS	Figure E-4	XORMPY[(1)]	Figure E-3
MPYHUS	Figure E-4	XPND2	Figure E-2
MPYIH	Figure E-1	XPND4	Figure E-2

(1) C64x+ 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 C64x CPU and C64x+ CPU 32-bit opcodes used in the .M unit are mapped in the following figures.

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 Unary Instruction Format

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

Figure E-3. Extended .M Unit 1 or 2 Sources, Nonconditional Instruction Format

31	30	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-4. MPY Instruction Format

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

E.4 16-Bit Opcode Maps

The C64x+ CPU 16-bit opcodes used in the .M unit for compact instructions are mapped in the following figure. See Section 3.9 for more information about compact instructions.

Figure E-5. M3 Instruction Format

15 13	12	11 10	9 7	6 5	4	3	2	1	0
src1	x	*dst*	*src2*	*op*	1	1	1	1	*s*
3	1	2	3	2					1

NOTE: RS = 0: dst from [A0, A2, A4, A6], [B0, B2, B4, B6]; RS = 1: dst from [A16, A18, A20, A22], [B16, B18, B20, B22]

Opcode map field used...	For operand type...
src1	sint
dst	sint
src2	xsint

SAT	*op*		Mnemonic
0	0	0	**MPY** (.unit) *src1, src2, dst*
0	0	1	**MPYH** (.unit) *src1, src2, dst*
0	1	0	**MPYLH** (.unit) *src1, src2, dst*
0	1	1	**MPYHL** (.unit) *src1, src2, dst*
1	0	0	**SMPY** (.unit) *src1, src2, dst*
1	0	1	**SMPYH** (.unit) *src1, src2, dst*
1	1	0	**SMPYLH** (.unit) *src1, src2, dst*
1	1	1	**SMPYHL** (.unit) *src1, src2, dst*

.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	Format	Instruction	Format
ADD	Figure F-1	MVKLH	Figure F-12
ADDK	Figure F-2	NEG	Figure F-1
ADDKPC [(1)]	Figure F-3	NOT	Figure F-1
ADD2	Figure F-1	OR	Figure F-1
AND	Figure F-1	PACK2	Figure F-4
ANDN	Figure F-4	PACKH2	Figure F-1
B displacement	Figure F-5	PACKHL2	Figure F-1
B register [(1)]	Figure F-6	PACKLH2	Figure F-1
B IRP [(1)]	Figure F-7	RPACK2 [(2)]	Figure F-13
B NRP [(1)]	Figure F-7	SADD	Figure F-1
BDEC	Figure F-8	SADD2	Figure F-4
BNOP displacement	Figure F-9	SADDSU2	Figure F-4
BNOP register	Figure F-10	SADDUS2	Figure F-4
BPOS	Figure F-8	SADDU4	Figure F-4
CALLP [(2)]	Figure F-11	SET	Figure F-1, Figure F-15
CLR	Figure F-1, Figure F-15	SHL	Figure F-1
CMPEQ2	Figure F-1	SHLMB	Figure F-4
CMPEQ4	Figure F-1	SHR	Figure F-1
CMPGT2	Figure F-1	SHR2	Figure F-4
CMPGTU4	Figure F-1	SHRMB	Figure F-4
CMPLT2	Figure F-1	SHRU	Figure F-1
CMPLTU4	Figure F-1	SHRU2	Figure F-4
DMV[(2)]	Figure F-4	SPACK2	Figure F-4
EXT	Figure F-1, Figure F-15	SPACKU4	Figure F-4
EXTU	Figure F-1, Figure F-15	SSHL	Figure F-1
MAX2[(2)]	Figure F-4	SUB	Figure F-1
MIN2[(2)]	Figure F-4	SUB2	Figure F-1
MV	Figure F-1	SWAP2	Figure F-1
MVC[(1)]	Figure F-1	UNPKHU4	Figure F-14
MVK	Figure F-12	UNPKLU4	Figure F-14
MVKH	Figure F-12	XOR	Figure F-1
MVKL	Figure F-12	ZERO	Figure F-1

(1) S2 only
(2) C64x+ 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, MVKH/MVKLH, or MVKL instruction
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
scstn	n-bit signed constant field
src1	source 1
src2	source 2
ucstn	n-bit unsigned constant field
ucst$_n$	bit n of the unsigned constant field
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 C64x CPU and C64x+ CPU 32-bit opcodes used in the .S unit are mapped in the following figures.

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. ADDK Instruction Format

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

Figure F-3. ADDKPC Instruction Format

31 29	28	27 23	22 16	15 13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	*dst*	*src1*	*src2*	0	0	0	0	1	0	1	1	0	0	0	*s*	*p*
3	1	5	7	3												1	1

Figure F-4. 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-5. Branch Using a Displacement 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. Branch Using a Register Instruction Format

31 29	28	27	26	25	24	23	22 18	17	16	15	14	13	12	11	10	9	8	7	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-7. Branch Using a Pointer Instruction Format

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

Figure F-8. BDEC/BPOS Instruction Format

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

Figure F-9. Branch Using a Displacement with NOP Instruction Format

31 29	28	27 16	15 13	12	11	10	9	8	7	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-10. Branch Using a Register with NOP Instruction Format

31 29	28	27	26	25	24	23	22 18	17	16	15 13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	*z*	0	0	0	0	1	*src2*	0	0	*src1*	x	0	0	1	1	0	1	1	0	0	0	1	*p*
3	1						5			3	1												1

Figure F-11. Call Nonconditional, Immediate with Implied NOP 5 Instruction Format

31	30	29	28	27 7	6	5	4	3	2	1	0
0	0	0	1	*cst21*	0	0	1	0	0	*s*	*p*
				21						1	1

Figure F-12. Move Constant 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	1

Figure F-13. Extended .S Unit 1 or 2 Sources, Nonconditional Instruction Format

31	30	29	28	27 23	22 18	17 13	12	11	10	9 6	5	4	3	2	1	0
0	0	0	1	*dst*	*src2*	*src1*	x	1	1	*op*	1	1	0	0	*s*	*p*
				5	5	5	1			4					1	1

Figure F-14. Unary Instruction Format

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

Figure F-15. 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

F.4 16-Bit Opcode Maps

The C64x+ CPU 16-bit opcodes used in the .S unit for compact instructions are mapped in the following figures. See Section 3.9 for more information about compact instructions.

Figure F-16. Sbs7 Instruction Format

15 13	12 6	5	4	3	2	1	0
N3	*scst7*	0	0	1	0	1	*s*
3	7						1

NOTE: N3 = 0, 1, 2, 3, 4, or 5

BR	Mnemonic
1	**BNOP** (.unit) *scst7*, *N3*

Figure F-17. Sbu8 Instruction Format

15	14	13 6	5	4	3	2	1	0
1	1	*ucst8*	0	0	1	0	1	*s*
		8						1

BR	Mnemonic
1	**BNOP** (.unit) *ucst8*, *5*

Figure F-18. Scs10 Instruction Format

15 6	5	4	3	2	1	0
scst10	0	1	1	0	1	*s*
10						1

NOTE: NextPC > B3, A3

BR	Mnemonic
1	**CALLP** (.unit) *scst10*, *5*

Figure F-19. Sbs7c Instruction Format

15 13	12 6	5	4	3	2	1	0
N3	*scst7*	1	*z*	1	0	1	*s*
3	7		1				1

NOTE: N3 = 0, 1, 2, 3, 4, or 5

BR	*s*	*z*	Mnemonic
1	0	0	**[A0] BNOP** .S1 *scst7*, *N3*
1	0	1	**[!A0] BNOP** .S1 *scst7*, *N3*
1	1	0	**[B0] BNOP** .S2 *scst7*, *N3*
1	1	1	**[!B0] BNOP** .S2 *scst7*, *N3*

Figure F-20. Sbu8c Instruction Format

15	14	13 6	5	4	3	2	1	0
1	1	*ucst8*	1	*z*	1	0	1	*s*
		8		1				1

BR	*s*	*z*	Mnemonic
1	0	0	**[A0] BNOP** .S1 u*cst8*, *5*
1	0	1	**[!A0] BNOP** .S1 *ucst8*, *5*
1	1	0	**[B0] BNOP** .S2 *ucst8*, *5*
1	1	1	**[!B0] BNOP** .S2 *ucst8*, *5*

Figure F-21. S3 Instruction Format

15 13	12	11	10	9 7	6 4	3	2	1	0
src1	x	*op*	0	*src2*	*dst*	1	0	1	*s*
3	1	1		3	3				1

Opcode map field used...	For operand type...
src1	sint
src2	xsint
dst	sint

BR	SAT	*op*	Mnemonic
0	0	0	**ADD** (.unit) *src1*, *src2*, *dst*
0	1	0	**SADD** (.unit) *src1*, *src2*, *dst*
0	x	1	**SUB** (.unit) *src1*, *src2*, *dst* (dst = src1 - src2)

Figure F-22. S3i Instruction Format

15 13	12	11	10	9 7	6 4	3	2	1	0
cst3	x	*op*	1	*src2*	*dst*	1	0	1	*s*
3	1	1		3	3				1

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

	32-Bit Opcode *cst* Translation	
cst3	*ucst5*	**Decimal Value**
000	10000	16
001	00001	1
010	00010	2
011	00011	3
100	00100	4
101	00101	5
110	00110	6
111	01000	8

BR	*op*	**Mnemonic**
0	0	**SHL** (.unit) *src2*, *ucst5*, *dst*
0	1	**SHR** (.unit) *src2*, *ucst5*, *dst*

Figure F-23. Smvk8 Instruction Format

15 13	12 11	10	9 7	6 5	4	3	2	1	0
$ucst_{2-0}$	*$ucst_{4-3}$*	*$ucst_7$*	*dst*	*$ucst_{6-5}$*	1	0	0	1	*s*
3	2	1	3	2					1

Opcode map field used...	**For operand type...**
dst	sint

Mnemonic
MVK (.unit) *ucst8*, *dst*

Figure F-24. Ssh5 Instruction Format

15 13	12 11	10	9 7	6 5	4	3	2	1	0
ucst$_{2-0}$	*ucst*$_{4-3}$	1	*src2/dst*	*op*	0	0	0	1	*s*
3	2		3	2					1

NOTE: x = 0, src and dst on the same side.

Opcode map field used...	**For operand type...**
src2/dst	sint

SAT	*op*		**Mnemonic**
x	0	0	**SHL** (.unit) *src2*, *ucst5*, *dst* (src2 = dst)
x	0	1	**SHR** (.unit) *src2*, *ucst5*, *dst* (src2 = dst)
0	1	0	**SHRU** (.unit) *src2*, *ucst5*, *dst* (src2 = dst)
1	1	0	**SSHL** (.unit) *src2*, *ucst5*, *dst* (src2 = dst)
x	1	1	see S2sh, Figure F-25

Figure F-25. S2sh Instruction Format

15 13	12 11	10	9 7	6	5	4	3	2	1	0
src1	*op*	1	*src2/dst*	1	1	0	0	0	1	*s*
3	2		3							1

NOTE: x = 0, src and dst on the same side.

Opcode map field used...	**For operand type...**
src2/dst	sint

op		**Mnemonic**
0	0	**SHL** (.unit) *src2*, *src1*, *dst* (src2 = dst, dst = src2 << src1)
0	1	**SHR** (.unit) *src2*, *src1*, *dst* (src2 = dst, dst = src2 >> src1)
1	0	**SHRU** (.unit) *src2*, *src1*, *dst* (src2 = dst, dst = src2 << src1)
1	1	**SSHL** (.unit) *src2*, *src1*, *dst* (src2 = dst, dst = src2 *sshl* src1)

Figure F-26. Sc5 Instruction Format

15–13	12–11	10	9–7	6–5	4	3	2	1	0
ucst$_{2-0}$	*ucst*$_{4-3}$	0	*src2/dst*	*op*	0	0	0	1	*s*
3	2		3	2					1

NOTES:

1. x = 0, src and dst on the same side
2. s = 0, dst = A0; s = 1, dst= B0

Opcode map field used...	**For operand type...**
src2/dst	sint

op		**Mnemonic**
0	0	**EXTU** (.unit) *src2*, *ucst5*,31, A0/B0
0	1	**SET** (.unit) *src2*, *ucst5*, *ucst5*, *dst* (src = dst, ucst5 = ucst5)
1	0	**CLR** (.unit) *src2*, *ucst5*, *ucst5*, *dst* (src = dst, ucst5 = ucst5)
1	1	see S2ext, Figure F-27

Figure F-27. S2ext Instruction Format

15–13	12–11	10	9–7	6	5	4	3	2	1	0
dst	*op*	0	*src2*	1	1	0	0	0	1	*s*
3	2		3							1

NOTE: x = 0, src and dst on the same side.

Opcode map field used...	**For operand type...**
dst	sint
src2	sint

op		**Mnemonic**
0	0	**EXT** (.unit) *src*,16, 16, *dst*
0	1	**EXT** (.unit) *src*,24, 24, *dst*
1	0	**EXTU** (.unit) *src*,16, 16, *dst*
1	1	**EXTU** (.unit) *src*,24, 24, *dst*

Figure F-28. Sx2op Instruction Format

15–13	12	11	10	9–7	6	5	4	3	2	1	0
src1/dst	x	*op*	0	*src2*	0	1	0	1	1	1	*s*
3	1	1		3							1

Opcode map field used...	For operand type...
src1/dst	sint
src2	xsint

op	Mnemonic
0	**ADD** (.unit) *src1*, *src2*, *dst* (src1 = dst)
1	**SUB** (.unit) *src1*, *src2*, *dst* (src1 = dst, dst = src1 - src2)

Figure F-29. Sx5 Instruction Format

15–13	12–11	10	9–7	6	5	4	3	2	1	0
ucst$_{2\text{-}0}$	*ucst*$_{4\text{-}3}$	1	*src2/dst*	0	1	0	1	1	1	*s*
3	2		3							1

Opcode map field used...	For operand type...
src2/dst	sint

Mnemonic
ADDK (.unit) *ucst5*, *dst*

Figure F-30. Sx1 Instruction Format

15–13	12	11	10	9–7	6	5	4	3	2	1	0
op	1	1	0	*src2/dst*	1	1	0	1	1	1	*s*
3				3							1

Opcode map field used...	For operand type...
src2/dst	sint

op			Mnemonic
0	0	0	see LSDx1, Figure G-4
0	0	1	see LSDx1, Figure G-4
0	1	0	**SUB** (.unit)0*, src2, dst* (src2 = dst, dst = 0 - src2)
0	1	1	**ADD** (.unit)-1, *src2, dst* (src2 = dst)
1	0	0	Reserved
1	0	1	see LSDx1, Figure G-4
1	1	0	**MVC** (.unit) *src,* ILC (s = 1)
1	1	1	see LSDx1, Figure G-4

Figure F-31. Sx1b Instruction Format

15–13	12	11	10–7	6	5	4	3	2	1	0
N3	0	0	*src2*	1	1	0	1	1	1	*s*
3			4							1

NOTE: src2 from B0-B15

Opcode map field used...	For operand type...
src2	uint

Mnemonic
BNOP (.unit) *src2, N3*

.D, .L, or .S Unit Opcode Maps

This appendix illustrates the opcode maps that execute in the .D, .L, or .S functional units.

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 .S functional unit, see Appendix F.

Topic **Page**

G.1 Opcode Map Symbols and Meanings

Table G-1 lists the symbols and meanings used in the opcode maps.

Table G-1. .D, .L, and .S Units Opcode Map Symbol Definitions

Symbol	Meaning
CC	
dst	destination
dstms	
op	opfield; field within opcode that specifies a unique instruction
s	side A or B for destination; 0 = side A, 1 = side B
src	source
src2	source 2
srcms	
ucstn	n-bit unsigned constant field
unit	unit decode
x	cross path for *src2*; 0 = do not use cross path, 1 = use cross path

G.2 32-Bit Opcode Maps

For the C64x CPU and C64x+ CPU 32-bit opcodes used in the .D functional unit, see Appendix C. For the C64x CPU and C64x+ CPU 32-bit opcodes used in the .L functional unit, see Appendix D. For the C64x CPU and C64x+ CPU 32-bit opcodes used in the .S functional unit, see Appendix F.

G.3 16-Bit Opcode Maps

The C64x+ CPU 16-bit opcodes used in the .D, .L, or .S units for compact instructions are mapped in Figure G-1 through Figure G-4. See Section 3.9 for more information about compact instructions.

Figure G-1. LSDmvto Instruction Format

15 13	12	11 10	9 7	6	5	4 3	2	1	0
dst	x	*srcms*	*src2*	0	0	*unit*	1	1	*s*
3	1	2	3			2			1

Opcode map field used...	**For operand type...**
dst	sint
src2	xsint

unit		**Mnemonic**
0	0	**MV** (.L*n*) *src, dst*
0	1	**MV** (.S*n*) *src, dst*
1	0	**MV** (.D*n*) *src, dst*

Figure G-2. LSDmvfr Instruction Format

15 13	12	11 10	9 7	6	5	4 3	2	1	0
dst	x	*dstms*	*src2*	1	0	*unit*	1	1	*s*
3	1	2	3			2			1

Opcode map field used...	**For operand type...**
dst	sint
src2	xsint

unit		**Mnemonic**
0	0	**MV** (.L*n*) *src, dst*
0	1	**MV** (.S*n*) *src, dst*
1	0	**MV** (.D*n*) *src, dst*

Figure G-3. LSDx1c Instruction Format

15	14	13	12	11	10	9	7	6	5	4	3	2	1	0
CC		*ucst1*	0	1	0	*dst*		1	1	*unit*		1	1	*s*
2		1				3				2				1

Opcode map field used...	For operand type...
dst	sint

CC		Mnemonic
0	0	**[A0] MVK** (.unit) *ucst1, dst*
0	1	**[!A0] MVK** (.unit) *ucst1, dst*
1	0	**[B0] MVK** (.unit) *ucst1, dst*
1	1	**[!B0] MVK** (.unit) *ucst1, dst*

CC		*unit*		Mnemonic
0	0	0	0	**[A0] MVK** (.L*n*) *ucst1, dst*
		0	1	**[A0] MVK** (.S*n*) *ucst1, dst*
		1	0	**[A0] MVK** (.D*n*) *ucst1, dst*

CC		*unit*		Mnemonic
0	1	0	0	**[!A0] MVK** (.L*n*) *ucst1, dst*
		0	1	**[!A0] MVK** (.S*n*) *ucst1, dst*
		1	0	**[!A0] MVK** (.D*n*) *ucst1, dst*

CC		*unit*		Mnemonic
1	0	0	0	**[B0] MVK** (.L*n*) *ucst1, dst*
		0	1	**[B0] MVK** (.S*n*) *ucst1, dst*
		1	0	**[B0] MVK** (.D*n*) *ucst1, dst*

CC		*unit*		Mnemonic
1	1	0	0	**[!B0] MVK** (.L*n*) *ucst1, dst*
		0	1	**[!B0] MVK** (.S*n*) *ucst1, dst*
		1	0	**[!B0] MVK** (.D*n*) *ucst1, dst*

Figure G-4. LSDx1 Instruction Format

15 13	12	11	10	9 7	6	5	4 3	2	1	0
op	1	1	0	*src/dst*	1	1	*unit*	1	1	*s*
3				3			2			1

Opcode map field used...	For operand type...
src/dst	sint

op			Mnemonic
0	0	0	**MVK** (.unit)0, *dst*
0	0	1	**MVK** (.unit)1, *dst*
0	1	0	See Dx1, Figure C-21; Lx1, Figure D-11; and Sx1, Figure F-30
0	1	1	See Dx1, Figure C-21; Lx1, Figure D-11; and Sx1, Figure F-30
1	0	0	See Dx1, Figure C-21; Lx1, Figure D-11; and Sx1, Figure F-30
1	0	1	**ADD** (.unit) *src,* 1, *dst* (src = dst)
1	1	0	See Dx1, Figure C-21; Lx1, Figure D-11; and Sx1, Figure F-30
1	1	1	**XOR** (.unit) *src,* 1, *dst* (src = dst)

op			*unit*		Mnemonic
0	0	0	0	0	**MVK** (.L*n*)0, *dst*
			0	1	**MVK** (.S*n*)0, *dst*
			1	0	**MVK** (.D*n*)0, *dst*

op			*unit*		Mnemonic
0	0	1	0	0	**MVK** (.L*n*)1, *dst*
			0	1	**MVK** (.S*n*)1, *dst*
			1	0	**MVK** (.D*n*)1, *dst*

op			*unit*		Mnemonic
1	0	1	0	0	**ADD** (.L*n*) *src,* 1, *dst*
			0	1	**ADD** (.S*n*) *src,* 1, *dst*
			1	0	**ADD** (.D*n*) *src,* 1, *dst*

op			*unit*		Mnemonic
1	1	1	0	0	**XOR** (.L*n*) *src,* 1, *dst*
			0	1	**XOR** (.S*n*) *src,* 1, *dst*
			1	0	**XOR** (.D*n*) *src,* 1, *dst*

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**

H.1 Instructions Executing With No Unit Specified

Table H-1 lists the instructions that execute with no unit specified.

Table H-1. Instructions Executing With No Unit Specified

Instruction	Format
DINT [1]	Figure H-1
IDLE	Figure H-2
NOP	Figure H-2
RINT [1]	Figure H-1
SPKERNEL [1]	Figure H-3
SPKERNELR [1]	Figure H-3
SPLOOP [1]	Figure H-4
SPLOOPD [1]	Figure H-4
SPLOOPW [1]	Figure H-4
SPMASK [1]	Figure H-3
SPMASKR [1]	Figure H-3
SWE [1]	Figure H-1
SWENR [1]	Figure H-1

[1] C64x+ CPU-specific instruction

H.2 Opcode Map Symbols and Meanings

Table H-2 lists the symbols and meanings used in the opcode maps.

Table H-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

H.3 32-Bit Opcode Maps

The C64x CPU and C64x+ CPU 32-bit opcodes used in the no unit instructions are mapped in the following figures.

Figure H-1. DINT and RINT, SWE and SWENR Instruction Format

31	30	29	28	27 24	23	22	21	20	19	18	17	16 13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	Reserved (0)	0	0	0	0	0	0	0	*op*	0	0	0	0	0	0	0	0	0	0	0	0	*p*
				4								4													1

Figure H-2. IDLE and NOP Instruction Format

31	30	29	28	27 18	17	16 13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	Reserved (0)	0	*op*	0	0	0	0	0	0	0	0	0	0	0	0	*p*
				10		4													1

Figure H-3. Loop Buffer, Nonconditional Instruction Format

31	30	29	28	27 23	22 18	17	16 13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	*cstb*	*csta*	1	*op*	0	0	0	0	0	0	0	0	0	0	0	*s*	*p*
				5	5		4												1	1

Figure H-4. 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

H.4 16-Bit Opcode Maps

The C64x+ CPU 16-bit opcodes used in the no unit instructions for compact instructions are mapped in the following figures. See Section 3.9 for more information about compact instructions.

Figure H-5. Uspl Instruction Format

15	14	13	12	11	10	9 7	6	5	4	3	2	1	0
0	*ii*$_3$	0	0	1	1	*ii*$_{2-0}$	1	1	0	0	1	1	*op*
	1					3							1

NOTE: Supports ii of 1-16

op	Mnemonic
0	**SPLOOP** *ii* (ii = real ii - 1)
1	**SPLOOPD** *ii*

Figure H-6. Uspldr Instruction Format

15	14	13	12	11	10	9–7	6	5	4	3	2	1	0
1	*ii*$_3$	0	0	1	1	*ii*$_{2-0}$	1	1	0	0	1	1	*op*
	1					3							1

NOTE: Supports ii of 1-16

op	**Mnemonic**
0	**[A0] SPLOOPD** *ii* (ii = real ii - 1)
1	**[B0] SPLOOPD** *ii*

Figure H-7. Uspk Instruction Format

15–14	13	12	11	10	9–7	6	5	4	3	2	1	0
ii/stg$_{4-3}$	0	1	1	1	*ii/stg*$_{2-0}$	1	1	0	0	1	1	*ii/stg*$_5$
2					3							1

Mnemonic
SPKERNEL *ii/stage*

Figure H-8. Uspm Instruction Format

a) SPMASK Instruction

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
D2	*D1*	1	0	1	1	*S2*	*S1*	*L2*	1	1	0	0	1	1	*L1*
1	1					1	1	1							1

NOTE: Supports masking of D1, D2, L1, L2, S1, and S2 instructions (not M1 or M2)

Mnemonic
SPMASK *unitmask*

b) SPMASKR Instruction

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
D2	*D1*	1	1	1	1	*S2*	*S1*	*L2*	1	1	0	0	1	1	*L1*
1	1					1	1	1							1

NOTE: Supports masking of D1, D2, L1, L2, S1, and S2 instructions (not M1 or M2)

Mnemonic
SPMASKR *unitmask*

Figure H-9. Unop Instruction Format

15–13	12	11	10	9	8	7	6	5	4	3	2	1	0
N3	0	1	1	0	0	0	1	1	0	1	1	1	0
3													

Mnemonic
NOP *N3*

Revision History

Table I-1 lists the changes made since the previous version of this document.

Table I-1. Document Revision History

Reference	Additions/Modifications/Deletions
Section 2.9.14.4	Added Caution.
SPACKU4	Changed figure.

Products		**Applications**	
Amplifiers	amplifier.ti.com	Audio	www.ti.com/audio
Data Converters	dataconverter.ti.com	Automotive	www.ti.com/automotive
DLP® Products	www.dlp.com	Communications and Telecom	www.ti.com/communications
DSP	dsp.ti.com	Computers and Peripherals	www.ti.com/computers
Clocks and Timers	www.ti.com/clocks	Consumer Electronics	www.ti.com/consumer-apps
Interface	interface.ti.com	Energy	www.ti.com/energy
Logic	logic.ti.com	Industrial	www.ti.com/industrial
Power Mgmt	power.ti.com	Medical	www.ti.com/medical
Microcontrollers	microcontroller.ti.com	Security	www.ti.com/security
RFID	www.ti-rfid.com	Space, Avionics & Defense	www.ti.com/space-avionics-defense
RF/IF and ZigBee® Solutions	www.ti.com/lprf	Video and Imaging	www.ti.com/video
		Wireless	www.ti.com/wireless-apps