Application Report
SPRAA84A–October 2005–Revised October 2005

Migrating from TMS320C64x to TMS320C64x+

Steven Gorwood *DSP Software Applications*

ABSTRACT

This document describes migration 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 and to briefly describe new features. Functionality in the devices that is identical is not included. For detailed information about either device, see the *TMS320C64x/C64x+ DSP CPU and Instruction Set Reference Guide* (SPRU732). For more detailed information about the internal DMA (IDMA) and interrupts, see the *TMS320C64x+ Megamodule Peripherals Reference Guide* (SPRU871). For more information about cache operations, see the *TMS320C64x+ DSP Cache User's Guide* (SPRU862). For more information about the enhanced DMA (EDMA) version 3.0, see the *EDMA v3.0 (EDMA3) Migration Guide for TMS320DM644x DMSoC* (SPRAAA6).

Contents

List of Figures

List of Tables

Trademarks

TMS320C64x, C64x+, TMS320C6000 are trademarks of Texas Instruments.

1 TMS320C64x+ DSP Overview

1.1 Architecture Overview

The TMS320C64x+™ digital signal processor (DSP) core is an extension of the TMS320C64x™ DSP core. The C64x+™ DSP adds increased functionality in several areas:

- Enhanced multiplication capability
- New instructions resulting in more efficient code execution and increased code compactness.
- Two level privilege system supports the use of higher capability operating systems.
- Software and hardware exceptions for error recovery and diagnosis.
- Flexible level 1 memory architecture that can be configured either as cache, as RAM or as mixed cache and RAM.

2 TMS320C64x+ DSP Changes to Existing TMS320C64x DSP Functionality

The C64x+ DSP core is based on the C64x DSP core and, unless new features available on the C64x+ DSP are used, most C64x software should run without problems after being recompiled for the new core and after being adjusted for memory map differences. This section describes the C64x+ DSP changes that may affect existing code. For detailed information, refer to the *TMS320C64x/C64x+ DSP CPU and Instruction Set Reference Guide* (SPRU732) and the data manual for your specific device.

Changes exist in the following areas that may affect existing code:

- Instruction Set
- Registers
- Interrupts
- DMA Operations
- Timing Changes
- Circular Addressing

2.1 Modified Instructions

On the C64x CPU, the following instructions compute the intermediate result to a 32-bit precision. On the C64x+ CPU, the following instructions compute the intermediate result to a 33-bit precision. As a result, some calculations that saturate on the C64x CPU will not saturate on the C64x+ CPU. See Appendix C for more details on the modified instructions.

- **DOTPNRSU2:** Dot product with negate, shift, and round of a signed 16-bit packed value and an unsigned packed 16-bit value
- **DOTPNRUS2:** Dot product with negate, shift, and round of an unsigned 16-bit packed value and an unsigned packed 16-bit value
- **DOTPRSU2:** Dot product with shift, and round of a signed 16-bit packed value and an unsigned packed 16-bit value
- **DOTPRUS2:** Dot product with shift, and round of an unsigned 16-bit packed value and a signed packed 16-bit value

2.2 Modified Registers

The function of the following registers has been modified on the C64x+ CPU:

- control status register (**CSR**)
- interrupt service table pointer register (**ISTP**)

2.2.1 Control Status Register (CSR)

The control status register (**CSR**) contains control and status bits. The field definitions for CSR are unchanged, and the function of the register will be unchanged so long as the program remains in the (default) supervisor mode. If the CPU is changed into the (new) user mode, some of the fields are protected from modification. This should not affect existing code. See Section 3.8 for more information on privilege.

The C64x+ CPU adds the following changes to the functionality of **CSR**:

- The CPU ID for the C64x+ CPU is 10h.
- The following bits are only writable in supervisor mode: **PWRD**, **PCC**, and **DCC**. See Section 3.8 for more information on supervisor mode.
- The **GIE** bit is the same physical bit as the **GIE** bit in the new task state register (**TSR**). See Section 3.2.1 for more information about **TSR**.
- The **PCC** and **DCC** bits have no effect on the C64x+ CPU.

2.2.2 Interrupt Service Table Pointer Register (ISTP)

The interrupt service table pointer register (**ISTP**) contains a pointer to the start of the current interrupt service table (IST). **ISTP** has a default value of 0000 0000h on the C64x CPU. This default value for **ISTP** is different for different devices using the C64x+ CPU. See the device-specific data manual for the default value of **ISTP**.

2.3 Interrupts

The interrupts on the C64x+ CPU function similarly to the C64x CPU, with the following exceptions:

- On the C64x CPU, the default location of the interrupt service table (**IST**) is located at 0000 0000h and the default value stored in the interrupt service table pointer register (**ISTP**) is 0. On the C64x+ CPU, the default location of the **IST** and the default value in **ISTP** is normally not 0. The default values vary from device to device, so refer to the device-specific data manual for the correct values.
- On the C64x CPU, the timing associated with interrupts is slightly different. See Section 2.6 for timing change information.
- The C64x+ CPU introduces two new instructions: **DINT** and **RINT**. These instructions support the atomic disabling and re-enabling of interrupts.
- On the C64x+ CPU, the task state register (**TSR**) is copied to the interrupt task state register (**ITSR**) on a maskable interrupt. Neither **TSR** nor **ITSR** existed on the C64x CPU. See Section 3.2.1 for more information about **TSR** and **ITSR**.
- On the C64x+ CPU, **TSR** is copied to the NMI/exception task state register (**NTSR**) on a non-maskable interrupt. Neither **TSR** nor **NTSR** existed on the C64x CPU. See Section 3.2.1 for more information about **TSR** and **NTSR**.
- The C64x+ CPU introduces exceptions. See Section 3.4 for more information on exceptions.

2.3.1 Entry to Interrupt

On the C64x CPU, the **GIE** bit in **CSR** is copied to the **PGIE** bit in **CSR** when an interrupt occurs.

On the C64x+ CPU, in addition to the above, **TSR** is copied to **ITSR**.

2.3.2 Return From Interrupt

On the C64x CPU, the **PGIE** bit in **CSR** is copied to the **GIE** bit in **CSR** when a return from interrupt occurs.

On the C64x+ CPU, in addition to the above, **ITSR** is copied to **TSR**. If the interrupt was an NMI (non-maskable interrupt) or exception, **NTSR** is copied to **TSR**.

2.4 Internal Memory and Coherence Operations

There have been significant changes to the treatment of internal memory on the C64x+ CPU. This section summarizes the changes that may affect existing code. See Section 3.7 for a more complete treatment of the changes.

2.4.1 L1D Cache is Not Contained in L2 Cache

On the C64x CPU, L1D cache is contained in the L2 cache and coherence between L2 and L1D cache is maintained by evicting data from L1D when the corresponding data is evicted from L2.

On the C64x+ CPU, L1D is not contained in L2 cache. If data is evicted from L2, the corresponding data in L1D cache is still available for use. This should result in faster execution of code in many cases.

2.4.2 L2 is Not Located at 0000 0000h

On the C64x CPU, L2 is always located with an origin of 0000 0000h.

On the C64x+ CPU, L2 will not be located at 0000 0000h. The location of L2 will vary between devices. See your device-specific data manual for information.

2.4.3 Coherency Between L2 Cache and L1P Cache is Not Maintained

On the C64x CPU, writes to L2 will automatically invalidate the corresponding region in L1P cache.

On the C64x+ CPU, writes to L2 will not invalidate the corresponding region in L1P. This will need to be done manually. This will primarily be an issue when overlayed code is swapped into and out of L2.

2.4.4 Non-Cacheable Code Fetches

On the C64x CPU, when caching of external spaces is disabled using the **MAR** bits, L1P will not cache that memory range.

On the C64x+ CPU, L1P will cache the memory range regardless of the values of the **MAR** bits.

2.4.5 L2WB and L2WBINV When L2 is in All RAM Mode

On the C64x CPU, L2 writeback (L2WB) and L2 writeback invalidate (L2WBINV) have no effect if L2 is in all RAM mode.

On the C64x+ CPU, these operations work regardless of L2 mode. If no L2 cache is configured, these operations will still writeback the contents of L1D to external memory and invalidate L1D or L1P.

2.5 DMA Functions

DMA functions are split between two separate transfer mechanisms on the C64x+ CPU:

- Internal DMA (IDMA) services transfers between internal memory spaces. See Section 3.5 for more information about IDMA.
- Enhanced DMA (EDMA) services transfers between internal and external memory spaces and between external memory spaces.

The EDMA controller handles all data transfers between the level 2 (L2) and level 1 (L1D and L1P) cache/memory controller and the device peripherals including external memory.

There are significant differences between the C64x EDMA and the C64x+ EDMA:

- On the C64x DSP, the EDMA control registers are always located at a specific location that do not change from device to device. On the C64x+ DSP, the location of the EDMA control registers changes from device to device.
- On the C64x DSP, the EDMA controller is capable of 1D and 2D DMA transfers. On the C64x+ DSP, the EDMA controller is capable of 1D, 2D, and 3D DMA transfers.
- On the C64x DSP, QDMA is a separate controller with a separate set of control registers. On the C64x+ DSP, QDMA is part of the EDMA controller and uses the same control registers.

- On the C64x DSP, the parameter RAM (PaRAM) entry for an EDMA transfer is 6-words long. On the C64x+ DSP, the PaRAM entry for an EDMA transfer is 8-words long.
  See the *EDMA v3.0 (EDMA3) Migration Guide for TMS320DM644x DMSoC* (SPRAAA6).

2.6 Timing Changes

2.6.1 Execution Time

Code compiled for the C64x+ DSP will probably run faster than the same code compiled for the C64x DSP due to changes in the instruction set.

2.6.2 Interrupt Latency

On the C64x CPU, the interrupt latency (that is, the shortest interval between the point of an interrupt and the first cycle of the interrupt) is 7 cycles.

On the C64x+ CPU, the interrupt latency is 9 cycles.

2.6.3 Interrupt Overhead

On the C64x CPU, the interrupt overhead (that is, the shortest round trip time for taking an interrupt and returning from the interrupt) is 11 cycles.

On the C64x+ CPU, the interrupt overhead is 13 cycles.

2.6.4 Branch to Execute Packets That Span Fetch Packets

On the C64x+ CPU, a branch to an execute packet that spans two fetch packets causes a stall while the second fetch packet is fetched. Since the assembler/compiler will not create code that has a branch to this kind of execute packet, the only case (other than legacy code that has not been recompiled) in which this should happen is on a return from an interrupt or exception.

2.7 Using Nonaligned Loads and Stores in Circular Addressing Mode

A new restriction to the size of circular address buffers exists on the C64x+ CPU when using the nonaligned load or nonaligned store operations (that is, **LDNDW**, **LDNW**, **STNDW**, or **STNW** instructions).

The size of the circular buffer must be at least 32 bytes when using the nonaligned load or nonaligned store instructions.

3 New Features on the TMS320C64x+ CPU

This section covers features that are available on the C64x+ CPU that are not available on the C64x CPU. In summary:

- New instructions support increased code efficiency and speed.
- New 16-bit compact instructions support increased code compactness.
- New privilege modes support secure operating systems.
- New support for exceptions provides new error handling capabilities.
- New SPLOOP facility provides improved code compactness and interruptibility for pipelined loops.
- Changes to internal memory increase the flexibility of internal memory usage.

3.1 New Instructions

The following new instructions are available on the C64x+ CPU. See Appendix D for more details on the new instructions. See Appendix A for the instruction compatibility between the C64x and C64x+ DSPs.

Expanded arithmetic functions to support FFT and DCT algorithms:

- **ADDSUB:** Parallel ADD and SUB 32-bit operations on common inputs
- **ADDSUB2:** Parallel ADD2 and SUB2 dual 16-bit operations on common inputs
- **SSUB2:** Subtract two signed 16-bit integers on upper and lower register halves with saturation
- **SADDSUB:** Parallel SADD and SSUB 32-bit operations on common inputs
- **SADDSUB2:** Parallel SADD2 and SSUB2 dual 16-bit operations on common inputs

The CALLP instruction improves the efficiency of subroutine calls:

- **CALLP:** Call using a displacement

Improved complex multiplication support:

- **CMPY:** Complex multiply of two pairs of signed, packed 16-bit values
- **CMPYR:** Complex multiply of two pairs of signed, packed 16-bit values with rounding. Rounding is done by adding 8000h to the result.
- **CMPYR1:** Complex multiply of two pairs of signed, packed 16-bit values with rounding. Rounding is down by adding 4000h to the result.

Double dot product instructions improve the throughput of FIR loops:

- **DDOTP4:** Double dot product of signed, packed 16-bit values and signed, packed 8-bit values
- **DDOTPH2:** Double dot product of two pairs of signed, packed 16-bit values
- **DDOTPH2R:** Double dot product of two pairs of signed, packed 16-bit values with rounding
- **DDOTPL2:** Double dot product of two pairs of signed, packed 16-bit values
- **DDOTPL2R:** Double dot product of two pairs of signed, packed 16-bit values with rounding

The C64x+ CPU adds instructions to disable and restore interrupts without having to directly manipulate the **GIE** bit:

- **DINT:** Disable interrupts
- **RINT:** Restore previous interrupt state

Improved data movement:

- **DMV:** Move two independent registers to a register pair

Parallel packing:

- **DPACK2:** Parallel PACK2 and PACKH2 operations
- **DPACKX2:** Parallel PACKLH2 instructions
- **RPACK2:** Pack two 16-bit MSBs into upper and lower register halves after shifting with saturation
- **SHFL3:** 3-way interleave on three 16-bit values into 48-bit result

Enhanced Galois Field Multiply:

- **GMPY:** Galois Field Multiply of a 32-bit value with a 9 bit value
- **XORMPY:** Galois Field Multiply with zero polynomial

Expanded multiply instructions improve the orthogonality of the multiplication instruction set:

- **MPY2IR:** Parallel 16-bit by 32-bit multiplies to produce a rounded 32-bit result
- **MPY32:** Multiply 32-bit signed by 32-bit signed to produce 32-bit or 64-bit result
- **MPY32SU:** Multiply 32-bit signed value by 32-bit unsigned value to produce signed 64-bit result
- **MPY32U:** Multiply two 32-bit unsigned values to produce 64-bit unsigned result
- **MPY32US:** Multiply 32-bit unsigned value by 32-bit signed value to produce signed 64-bit result modified instructions
- **SMPY32:** Multiply two 32-bit signed values with 64-bit result with left shift and saturation. Only most-

significant 32-bits is written

The software pipeline loop (SPLOOP) facility provides a more efficient way of writing highly optimized loops. The following instructions have been added to support the SPLOOP facility:

- **SPKERNEL:** SPLOOP code boundary
- **SPKERNELR:** SPLOOP code boundary with reload
- **SPLOOP:** Software pipelined loop
- **SPLOOPD:** Software pipelined loop with delayed test for termination
- **SPLOOPW:** Software pipelined loop with delayed testing and no epilog
- **SPMASK:** Software pipelined loop load/execution control
- **SPMASKR:** Software pipelined loop load/execution control

The exception mechanism on the C64x+ CPU is intended to support error detection and program redirection to error handling routines. Two new instructions have been added to add the capability of triggering an exception under program control. These instructions are also used to communicate between privileged and non-privileged tasks.

- **SWE:** Software exception
- **SWENR:** Software exception with no return

Although the **ADDAB**, **ADDAH**, and **ADDAW** instructions existed on the C64x CPU, the C64x+ CPU adds a new form for these instructions that expands the flexibility of their use. On the C64x CPU, they are used to add the contents of two registers. The C64x+ CPU adds a new form that supports the addition of an unsigned 15-bit immediate constant to the contents of either B14 or B15:

- **ADDAB:** Add using byte-addressing mode
- **ADDAH:** Add using halfword-addressing mode
- **ADDAW:** Add using word-addressing mode

On the C64x CPU, the **BNOP** instruction used either the .S1 or .S2 execution unit. On the C64x+ CPU, a new unitless form of the instruction is available that does not use either unit. This should not impact existing code, but will expand scheduling flexibility on new code.

- **BNOP:** Branch using a displacement with NOP

On the C64x CPU, the **MIN2** and **MAX2** instructions use either the .L1 or .L2 functional unit. On the C64x+ CPU, the instructions use one of the .L1, L2, .S1, or .S2 functional units. The increased number of functional units will add scheduling flexibility on new code (resulting in faster execution and smaller code size for the same functionality).

- **MAX2:** Maximum operation on signed, packed 16-bit values
- **MIN2:** Minimum operation on signed, packed 16-bit values

3.2 New Registers

The C64x+ DSP adds 12 new control registers that are not present on the C64x DSP.

- The **TSR** (task state register), **ITSR** (interrupt task state register), and **NTSR** (NMI/exception task state register) store information about the current state of the current execution environment.
- The **ILC** (inner loop counter) and **RILC** (reload inner loop counter) are used for loop control during software pipeline loop (SPLOOP) operation.
- The **IERR** (internal exception report register) contains flags indicating the cause on internal exceptions.
- The **EFR** (exception flag register) contains flags indicating pending exceptions.
- The **ECR** (exception clear register) is used to clear flags in **EFR**.
- The **REP** (restricted entry point register) specifies the target of the **SWENR** instruction.
- The **SSR** (saturation status register) provides saturation flags for each functional unit.
- The **GPLYA** and **GPLYB** are used to store the 32-bit polynomial used by the new Galois Multiply (**GMPY**) instruction.
- The 32-bit **TSCL** (time stamp counter low register) and the 32-bit **TSCH** (time stamp counter high register) are used to read the (new) 64-bit free-running counter.

3.2.1 TSR, ITSR, and NTSR Registers

Several new features on the C64x+ CPU have state information that must be tracked and restored across the context switch associated with interrupts or exceptions. The task state register (**TSR**) stores information such as:

- Is an SPLOOP currently running? If so, it will need to be restarted after any interrupt.
- Are exceptions and/or interrupts enabled?
- What is the privilege state (user mode or supervisor mode)?
- Is an exception or interrupt currently processing?
- Are interrupts currently architecturally blocked? An example of this is the period when the processor is in the delay slots of a branch instruction.

The interrupt task state register (**ITSR**) is used to store the contents of **TSR** in the event of an interrupt. The NMI/exception task state register (**NTSR**) is used to store the contents of **TSR** and the conditions under which an exception occurred in the event of a nonmaskable interrupt (NMI) or exception.

3.2.2 ILC and RILC Registers

The SPLOOP is a new feature that provides hardware support for coding software pipelined loops. The inner loop counter register (**ILC**) and reload inner loop counter register (**RILC**) are provided to store the desired number of loop iterations during an SPLOOP execution.

ILC is used by the SPLOOP module to store the loop counter. **RILC** is used by the SPLOOP module to reload **ILC** with a default value in the case of nested pipelined loops.

3.2.3 EFR, ECR, IERR, and REP Registers

Several registers have been added to support the exception mechanism on the C64x+ CPU. Exceptions provide a mechanism for detecting, reporting, and handling errors. See Section 3.4 for more information.

The exception flag register (**EFR**) contains four flags that indicate the general source of pending exceptions (that is, whether the exception was internal, external, triggered by the deliberate execution of an **SWE** or **SWENR** instruction, or was a non-maskable interrupt). **EFR** bits can only be cleared by writing a 1 to the equivalent bit in **ECR**.

The exception clear register (**ECR**) is used to clear the bits in **EFR**.

The internal exception report register (**IERR**) contains flags that indicate specific type of an exception (that is, resource conflict, privilege violation, etc).

The value stored in the restricted entry point address register (**REP**) is used by the **SWENR** instruction to determine the target of the change of control when an **SWENR** instruction is executed. **REP** should be pre-initialized in supervisor mode before any **SWENR** instruction is issued. This is used when a user mode task terminates and returns to the supervisor mode task that spawned it.

3.2.4 SSR Register

On the C64x CPU, the saturation in an instruction results in setting a single bit (**SAT**) in the control status register (**CSR**). It is not possible to determine which of several instructions executing in parallel on different functional units is the cause of the saturation.

The C64x+ CPU adds the saturation status register (**SSR**) to address this issue. **SSR** contains 8 bits (one bit for each functional unit) that are individually set by saturation by each functional unit.

The bits in **SSR** can only be cleared by a reset or by the **MVC** instruction.

The functionality of the **SAT** bit in **CSR** is unchanged and a saturation by any functional unit will set both the **SAT** bit in **CSR** and a corresponding bit in **SSR**.

For example: The **SSHL** is a saturating shift left instruction. If the command: SSHL .S1 A2, 2, A1, results in a saturated result, both the **SAT** bit in **CSR** and the **S1** bit in **SSR** are set.

3.2.5 GPLYA and GPLYB Registers

The C64x+ CPU adds a new instruction (**GMPY**) that is not available on the C64x CPU. The **GMPY** instruction performs a Galois field multiply between a 32-bit value and a 9-bit value.

The C64x CPU has the **GMPY4** instruction that does a Galois field multiply of packed 8-bit data.

The **GMPY** instruction uses the 32-bit **GPLYA** to store the polynomial, when it is executed on the A side using the .M1 unit; and uses the 32-bit **GPLYB** to store the polynomial, when it is executed on the B side using the .M2 unit. Because two registers are used to store the polynomial, two **GMPY** instructions can be executed with different polynomials on the same instruction cycle, resulting in increased scheduling flexibility.

3.2.6 TSCL and TSCH Registers

Many applications need a simple low latency method for measuring intervals. The C64x+ CPU has a 64-bit free running counter that is read by reading the time stamp counter low register (**TSCL**) and the time stamp counter high register (**TSCH**):

- **TSCL** is used to read the lower 32-bits of the free-running 64-bit counter.
- **TSCH** is used to read the upper 32-bits of the free-running 64-bit counter.

The counter is enabled by writing to **TSCL** using the **MVC** instruction. The value written by the **MVC** instruction will be ignored. Once started, the counter will increment each CPU clock and cannot be stopped.

The value of the upper 32-bits of the counter is only written to **TSCH** when the values of the lower 32-bits are read using the **MVC** instruction. Only this snapshot is available for reading.

3.3 Compact Instructions

On the C64x CPU, all instructions are 32-bits wide; the C64x+ CPU introduces a set of 16-bit compact instructions that can replace their 32-bit equivalents to reduce code size with no impact on functionality or speed. The compiler will use these 16-bit instructions where it finds potential code size savings.

See Appendix A for a list of the available compact instructions on the C64x+ CPU.

3.3.1 Compact Instruction Header

All fetch packets on the C64x CPU are of a form similar to that shown in Figure 1. Each instruction is a uniform 32-bits wide.

The C64x+ CPU adds compact (16-bit) versions of many of the most useful instructions. The availability of compact instructions is enabled by the replacement of the eighth word of the fetch packet with a 32-bit header word. The header word:

- Identifies the fetch packet as a special packet that may contain compact instructions.
- Identifies which of the words in the fetch packet contain compact instructions.
- Identifies which of the instructions in the fetch packet will execute in parallel.
- Contains some decoding information that supplements the information contained in the 16-bit compact opcode.

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 as shown in Figure 1.

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

- The compact instructions do not have a dedicated predication field.
- Only a 3-bit address field is available.
- Only a subset of the functionality of the 32-bit instruction is available.

Figure 1. Fetch Packet Types

Standard C64x CPU Fetch Packet		Header-Based Fetch Packet		
Word		Word		
0	32-bit opcode	0	16-bit opcode	16-bit opcode
1	32-bit opcode	1	32-bit opcode	
2	32-bit opcode	2	16-bit opcode	16-bit opcode
3	32-bit opcode	3	32-bit opcode	
4	32-bit opcode	4	16-bit opcode	16-bit opcode
5	32-bit opcode	5	32-bit opcode	
6	32-bit opcode	6	16-bit opcode	16-bit opcode
7	32-bit opcode	7	**Header**	

3.4 Exceptions

The exception mechanism on the C64x+ CPU is intended to support error detection and program redirection to error handling routines.

There are three types of exceptions on the C64x+ CPU.

- One externally generated maskable exception
- One externally generated non-maskable exception
- Internally non-maskable exceptions

The two external exceptions are triggered by two inputs from outside the CPU boundary. The significance of these two signals changes from device to device. See your device-specific data manual for more information.

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. An internal exception can also be forced by executing an **SWE** or **SWENR** instruction. Registers associated with exceptions are listed in Table 1.

Table 1. Summary of Exception Control Registers

Acronym	Register Name	Description
ECR	Exception Clear Register	Used to clear pending exception flags
EFR	Exception Flag Register	Contains pending exception flags
IER	Interrupt Enable Register	Contains NMI exception enable (**NMIE**) bit
IERR	Internal Exception Report Register	Indicates the cause of the internal exception
ISTP	Interrupt Service Table Pointer Register	Contains pointer to the beginning of the interrupt service fetch packet
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.
NTSR	Nonmaskable Interrupt/Exception Task State Register	Stores the contents of **TSR** upon taking an exception
REP	Restricted Entry Point Address Register	Contains the address to where the **SWENR** instruction transfers control
TSR	Task State Register	Contains the global exception enable (**GEE**) and exception enable (**XEN**) bits.

3.4.1 Enabling Exceptions

At reset, exceptions are disabled. Exceptions are globally enabled by setting the **GEE** bit in **TSR**. Once enabled, exceptions cannot be disabled.

The **XEN** bit in **TSR** and the **NMIE** bit in **IER** is used to enable or disable the external maskable exception.

3.4.2 Taking Exceptions

The following actions happen when the CPU begins processing an exception:

- One or more bits are set in the exception flags register (**EFR**). The exception service routine can examine these bits to determine that type of the exception.
- Zero or more bits are set in the internal exception report register (**IERR**). The exception service routine can examine these bits to determine that cause of the exception.
- The contents of **TSR** are copied to **NTSR**.
- The return address is placed in **NRP**.
- Control is transferred to the NMI fetch packet in the interrupt service table pointed to by **ISTP**.

In general, the exception service routine for an exception is too large to fit in a single fetch packet, so the NMI fetch packet normally contains a branch to a larger routine that decodes the cause of the exception and acts accordingly.

3.4.3 Returning From Exceptions

After processing an exception is complete, the return to the application code is done by executing a branch to the NMI return pointer register (**NRP**); that is, execute a **B NRP** instruction. **NRP** contains the return pointer that directs the CPU to the proper location to continue program execution.

Before returning from an exception, the bits in **EFR** should be cleared by writing 1s to the corresponding bits in **ECR**.

Since exceptions can interrupt an instruction at any point, it is not always possible to safely return from an exception. Prior to returning, executing the **B NRP** instruction, the exception routine should check the status of the **IB**, **GIE**, and **SPLX** bits in **NTSR** to ensure that the interrupted program can be resumed normally. All three bits should be 0 to safely return to the interrupted code. Example 1 shows code for how to safely return from an exception.

Example 1. Returning From Exception Code

```
 STNDW B1:B0,*SP-- ;save B0 and B1 to the stack
|| MVC NTSR, B0 ;Read NTSR
 EXTU B0,16,30,B1 ;B1 = NTSR.IB and NTSR.SPLIX
|| AND B0,1,B0 ;B0 = NTSR.GIE
 OR B0,B1,B0 ;are all 0?
 [!B0] B NRP ;if B0 = 0, return
 LDNDW *SP++,B1:B0 ;restore B0 and B1
 NOP 4 ;delay slots
cant_restore:
;code to handle non-restartable case starts here
```

3.4.4 SWE and SWENR Instructions

The software exception (**SWE**) and software exception with no return (**SWENR**) instructions are used to trigger an exception through software.

When an **SWE** instruction is executed, the exception detection logic is signaled and an exception is taken setting the **SXF** bit in the exception flag register (**EFR**). Execution of this instruction will result in transfer of control to the exception service routine and change the operating mode to supervisor mode.

The **SWE** instruction is intended as a mechanism for user mode tasks to invoke system calls that operate in supervisor mode. If the **SXF** bit is the only bit set in **EFR**, the exception can be interpreted as a system service request. An appropriate calling convention using the general purpose register may be adopted to provide parameters for the call.

The **SWENR** instruction causes a software exception to be taken similarly to that caused by the **SWE** instruction. Unlike the **SWE** instruction, no provision is made for returning from the exception. **TSR** is not copied to **NTSR** and no return address is placed in **NRP**. In addition, unlike the **SWE** instruction (that transfers control to the NMI vector in the interrupt service table), the **SWENR** instruction will transfer control to the address stored in the restricted entry point address register (**REP**).

The **SWENR** instruction is intended to provide a mechanism for tasks operating in user mode to terminate and return control to the supervisor mode routine which invoked it.

If another exception (internal or external) is recognized simultaneously with the **SWENR** instruction raised exception, then the other exception(s) take priority and normal exception behavior occurs, that is, **NTSR** and **NRP** are used and execution is directed to the NMI vector in the interrupt service table.

See Section 3.8 for more information on privilege, user mode and supervisor mode.

3.5 Internal DMA (IDMA)

The internal DMA (IDMA) controller allows rapid data transfers between all local memories. It allows a fast way to page data sections to any memory-mapped RAM local to the CPU. The key advantage of the IDMA is that it allows for transfers between slow (L2) and fast (L1D) data memory, which provides lower latency than the cache controller. In addition, the transfers take place in the background of CPU operation, so that stalls due to cache can be removed from the system.

The IDMA consists of two channels. The two channels are fully orthogonal to each other allowing concurrent operation. The channels are:

- IDMA0 is intended for quick programming of configuration registers located in the external configuration space.
- IDMA1 is intended for transferring data between local memories.

See the *TMS320C64x+ Megamodule Peripherals Reference Guide* (SPRU871) for more information about IDMA.

3.5.1 IDMA Channel 0

IDMA channel 0 is intended to be used for short (up to 16 32-word windows) transfers to and from configuration space. IDMA channel 0 is controlled by a set of five memory-mapped registers as shown in Table 2.

- IDMA channel 0 status register (IMDA0_STAT) provides that activity state of the channel. There are two bits to denote whether a transfer is in progress and whether a transfer is pending.
- IDMA channel 0 mask register (IDMA0_MASK) allows unwanted registers within the transfer window to be blocked from access, facilitating multiple read/write transactions be completed with a single transfer command by the CPU.
- IDMA channel 0 source address register (IDMA0_SOURCE) specifies the source address of the IDMA transfer.
- IDMA channel 0 destination address register (IDMA0_DEST) specifies the destination address of the IDMA transfer.
- IDMA channel 0 window count register (IDMA0_COUNT) specifies the number of contiguous 32-word windows to be accessed during the data transfer. In addition, the register allows a CPU interrupt to be enabled to notify the CPU that a transfer has completed.

IDMA transfers are submitted automatically when the CPU writes to the transfer control registers. The CPU must write to all of the channels registers, in sequential incrementing order, for an IDMA transfer to trigger. The transfer will begin once IDMA0_COUNT is written. The CPU may either monitor IDMA0_STAT or enable the CPU interrupt in IDMA0_STAT to determine when the transfer is complete.

Table 2. IDMA Channel 0 Registers

Register	Address	Description
IDMA0_STAT	0182 0000h	IDMA Channel 0 Status Register
IDMA0_MASK	0182 0004h	IDMA Channel 0 Mask Register
IDMA0_SOURCE	0182 0008h	IDMA Channel 0 Source Address Register
IDMA0_DEST	0182 000Ch	IDMA Channel 0 Destination Address Register
IDMA0_COUNT	0182 0010h	IDMA Channel 0 Window Count Register

3.5.2 IDMA Channel 1

IDMA channel 1 is intended for transferring data between local memories. IDMA channel 1 is controlled by a set of four memory-mapped registers as shown in Table 3.

- IDMA channel 1 status register (IMDA1_STAT) provides that activity state of the channel. There are two bits to denote whether a transfer is in progress and whether a transfer is pending.
- IDMA channel 1 source address register (IDMA1_SOURCE) specifies the source address of the IDMA transfer.
- IDMA channel 1 destination address register (IDMA1_DEST) specifies the destination address of the IDMA transfer.
- IDMA channel 1 window count register (IDMA1_COUNT) specifies the transfer length in bytes. In addition, the register allows a CPU interrupt to be enabled and identifies the priority level (relative to CPU and DMA accesses) to be specified.

IDMA transfers are submitted automatically when the CPU writes to the transfer control registers. The CPU must write to all of the channels registers, in sequential incrementing order, for an IDMA transfer to trigger. The transfer will begin once IDMA1_COUNT is written. The CPU should monitor IDMA1_STAT to determine when the transfer is complete.

Table 3. IDMA Channel 1 Registers

Register	Address	Description
IDMA1_STAT	0182 0100h	IDMA Channel 1 Status Register
IDMA1_SOURCE	0182 0108h	IDMA Channel 1 Source Address Register
IDMA1_DEST	0182 010Ch	IDMA Channel 1 Destination Address Register
IDMA1_COUNT	0182 0110h	IDMA Channel 1 Window Count Register

3.6 SPLOOP

The eight functional units on the C64x and C64x+ DSPs provide a powerful environment for complex computational loops. A technique called software pipelining is used to take advantage of this parallelism where several loop iterations are actually executed in parallel.

Example 2 shows a simple three instruction loop. Example 3 shows an optimized implementation of the three instruction loop.

By the third cycle, the loop is executing 3 instructions per machine cycle. This portion of the loop is called the loop kernel. On the C64x and C64x+ CPU, you can potentially execute up to eight instructions per cycle within the kernel (one for each of the eight functional units).

The loop instructions prior to the kernel as the loop is being prepared are called the prolog. The loop instructions after the kernel as the loop completes are called the epilog.

The SPLOOP facility loads the instructions to the SPLOOP buffer and executes the loop iterations from the SPLOOP buffer instead of program memory. Example 4 shows an SPLOOP version of the code.

Example 2. Simple Three Instruction Loop

```
For (I=0; i<n; I++) {
 instruction 1;
 instruction 2;
 instruction 3;
}
```

Example 3. Optimized Three Instruction Loop

```
;First cycle of prolog
 Instruction 1 of first loop iteration on functional unit 1

;Second cycle of prolog
Instruction 1 of second loop iteration on functional unit 1
Instruction 2 of first loop iteration on functional unit 2

;Here is the loop kernel
loop here (–3) times {
Instruction 1 of ith loop iteration on functional unit 1
Instruction 2 of (I-1) th loop iteration on functional unit 2
Instruction 3 of (I-2) th loop iteration on functional unit 3
}

;First cycle of epilog
Instruction 2 of tenth loop iteration on functional unit 2
Instruction 3 of ninth loop iteration on functional unit 3

;Second cycle of epilog
Instruction 3 of tenth loop iteration on functional unit 3
```

Example 4. SPLOOP Version of Three Instruction Loop

```
 MVC IterationCount,ILC
 NOP 3
 SPLOOP 1
 Instruction 1
 Instruction 2
 Instruction 3
 SPKERNEL
```

The **MVC** instruction loads the desired number of iterations into the inner loop count register (**ILC**). The beginning of the looped code is marked with the **SPLOOP** instruction. The end of the looped code is marked with the **SPKERNEL** instruction. Since the code is executing from a buffer, no branch is required to branch back to the beginning of the loop.

Using the SPLOOP facility provides the following advantages:

- SPLOOP implementations normally are much more economical in terms of code size because the prolog and epilog do not need to be coded explicitly.
- In general, non-SPLOOP implementations are not fully interruptible. SPLOOP implementations are interruptible
- Executing a loop from the SPLOOP buffer uses slightly less energy than fetching the instructions from program memory.

3.6.1 SPLOOP Registers

Two new register have been added to the C64x+ core to support the SPLOOP facility.

- The inner loop count register (**ILC**) contains the desired number of iterations. It should be loaded four instruction cycles prior to the **SPLOOP** instruction or on the same cycle as the **SPLOOPD** instruction. It is not required when using the **SPLOOPW** instruction.
- The reload inner loop count register (**RILC**) is used to reload **ILC** when a nested loop is reloaded. When required, it should be initialized at the same time as **ILC**.

3.6.2 SPLOOP Instructions

Seven new instructions have been added to the C64x+ instruction set to support the SPLOOP facility.

- **SPLOOP**, **SPLOOPD**, and **SPLOOPW** instructions are used to mark the beginning of an SPLOOP block of code. Each instruction has a parameter (the iteration interval) that specifies the interval (in instruction cycles) between successive iterations of the loop. SPLOOP assumes that the inner loop count register (**ILC**) was preloaded at least four instruction cycles previously. The **SPLOOPD** instruction permits **ILC** to be loaded on the same instruction cycle as the **SPLOOPD** instruction, but requires that the loop be long enough that the 4 cycle delay elapses before the termination condition becomes true and the value loaded to **ILC** needs to be adjusted to account for the delayed load. The **SPLOOPW** instruction does not use **ILC**, it is intended to be used when a do/while loop structure is desired.
- **SPKERNEL** and **SPKERNELR** instructions are used to mark the end of an SPLOOP block of code. The **SPKERNELR** instruction is used when a reload of a nested loop is required.
- The **SPMASK** and **SPMASKR** instructions are used to either:
  - Inhibit the loading of instructions into the SPLOOP buffer when they are executed on the first iteration of a loop.
  - Mask the presence of an instruction within the SPLOOP buffer so that it can be replaced with a different instruction in the epilog portion of the loop cycle or between nested loops.
- The **SPMASKR** instruction is used when a delayed reload of a nested loop is required.

3.6.3 Converting An Existing Loop to SPLOOP

The compiler will utilize the SPLOOP where appropriate and it is not expected that an SPLOOP will often be coded by hand. On the (hopefully rare) cases where an existing loop needs to be converted to an SPLOOP form, the following general steps are used:

1. Isolate the prolog or loop setup code.
2. Initialize the inner loop count register (**ILC**) with the desired number of loop iterations. When a nested loop is being coded, it will also be necessary to initialize the reload inner loop count register (**RILC**). Write the loop as single scheduled iteration.
3. Determine the minimum iteration interval. In general, this will be determined by the usage of functional units within the loop. Use this value as the **SPLOOP(D/W)** instruction operand.
4. Start the loop with an **SPLOOP**, **SPLOOPD**, or **SPLOOPW** instruction.
5. Finish the loop with an **SPKERNEL** or **SPKERNELR** command.
6. Place a single, scheduled iteration of the loop kernel between the **SPLOOPD** or **SPLOOPW** instruction used to mark the beginning of the loop and the **SPKERNELR** or **SPMASKR** instruction used to mark the end of the loop.
7. If a nested loop is being coded, use either the **SPKERNELR** or **SPMASKR** instruction (but not both) to specify the reload point.
8. If a nested loop is being coded, the outer loop code must contain a conditional branch to the first execute packet following the **SPKERNEL** or **SPKERNELR** instruction (so that the program counter is set correctly for the next execution of the outer loop code).
9. Use the **SPMASK** or **SPMASKR** instruction to merge setup code with the first iteration of the loop or to merge setup code before a loop reload (when coding a nested loop).

3.7 Internal Memory

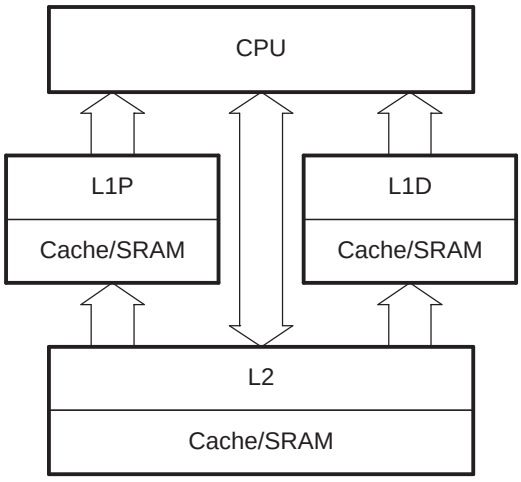
The configuration of internal memory on the C64x+ CPU is more configurable than the internal memory on the C64x CPU; providing the capability to tailor the performance of the device to the particular applications needs.

See the *TMS320C64x+ Megamodule Peripherals Reference Guide* (SPRU871) for more detailed information about internal memory.

Internal memory on the C64x+ CPU is divided into three types: L1P, L1D, and L2 (see Figure 2).

- L1P is program memory. It can be configured as level 1 cache, RAM, or some combination of the two.
- L1D is data memory. It can be configured as level 1 cache, RAM or some combination of the two.
- L2 is level 2 cache or RAM or some combination of the two.

Figure 2. TMS320C64x+ CPU Internal Memory



3.7.1 L1P

On the C64x CPU, L1P is always a 16 Kbyte direct mapped cache.

On the C64x+ CPU, L1P consists of two contiguous regions that may have different characteristics. Some characteristics may depend upon the specific device definition. Some may be varied under program control at program run time.

The following L1P characteristics are device dependent. Consult your device-specific data manual for more information.

L1P size will depend upon the device. Maximum L1P size is 1024 KB. The maximum size of each of the two L1P regions is 512 KB.

- L1P region 0 may start at different addresses on different devices. Region 0 may be 0 KB in size (thus disabling it), or some non-zero size up to a maximum of 512 KB in size.
- L1P region 1 will be located contiguous with region 0 and will have non-zero size up to a maximum size of 512 KB.
- The two L1P regions may have different access times depending on the type of memory used for each region
- L1P will be configured either as all RAM/ROM or as maximal cache at boot depending on the device.

The following L1P characteristics are selectable at program run time under software control:

- L1P cache size may be adjusted to be 0 KB, 4 KB, 8 KB, 16 KB, or 32 KB by writing to L1PCFG.
- L1P RAM converts to cache starting at the top of the L1P memory map and working downwards. That is, the highest addresses of L1P region 1 are the first to become cache.

The differences between C64x and C64x+ CPU L1P memory is summarized in Table 4.

Table 4. L1P Comparison Between C64x and C64x+ CPUs

Internal Memory Structure	TMS320C64x CPU	TMS320C64x+ CPU
L1P size	16 KB	Device dependent. Will be a multiple of 16 KB with a maximum of 1024 KB
L1P memory types	Cache	May be RAM, ROM, or Cache
L1P configuration at RESET	All Cache	Device dependent. May be either all RAM/ROM or Maximal Cache.
L1P RAM/ROM Address	N/A	Device dependent. Must be on 1 MB boundary.
L1P cache organization	Direct Mapped	Direct Mapped
L1P CPU access time	1 cycle	1 -> 4 cycles depending on memory type
L1P line size	32 bytes	32 bytes
L1P read miss policy	1 line allocated to L1P	1 line allocated to L1P
L1P read hit policy	Data read from L1P	Data read from L1P
L1P write miss policy	L1P writes not supported	L1P writes not supported
L1P write hit policy	L1P writes not supported	L1P writes not supported
L1P protocol	Read Allocate; Pipelined Misses	Read Allocate; Pipelined Misses
L1P Memory	Single cycle RAM	RAM or ROM
L1P -> L2 request size	1 fetch/L1P line	1 fetch/L1P line
L1P -> L2 request CPU stall	8 cycles	Varies with device

3.7.1.1 Initializing L1P RAM

On the C64x CPU, L1P is always configured as cache, so initializing the contents of L1P is not an issue.

On the C64x+ CPU, L1P may be configured either completely or partially as RAM.

L1P cannot be directly written to by the CPU using the normal load/store operations. Either IDMA or EDMA should be used to write data to the L1P SRAM.

3.7.1.2 Non-Cacheable Code Fetches

On the C64x CPU, when caching of external spaces is disabled using the **MAR** bits, L1P will not cache that memory range.

On the C64x+ CPU, L1P will cache the memory range regardless of the values of the **MAR** bits.

3.7.2 L1D

On the C64x CPU, L1D is always a 16 KB 2-way set-associative cache.

On the C64x+ CPU, L1D consists of two contiguous regions that may have different characteristics. Some characteristics may depend upon the specific device definition. Some may be varied under program control at program run time.

The following L1D characteristics are device dependent. Consult your device-specific data manual for more information:

- Maximum L1D size is 1024 KB. The maximum size of each of the two L1D regions is 512 KB.
- L1D region 0 will always start on a 1MB boundary, but may be configured to be at different addresses on different devices. Region 0 may be 0 KB in size (thus disabling it), or some multiple of 16 KB up to 512 KB in size.
- L1D region 1 will be located contiguous with region 0 and will have a size of some multiple of 16 KB up to a maximum size of 512 KB.
- L1D will be configured either as all RAM or as maximal cache at boot depending on the device.

The following L1D characteristics are selectable at program run time under software control:

- L1D cache size may be adjusted to be 0 KB, 4 KB, 8 KB, 16 KB, or 32 KB by writing to L1DCFG.
- L1D RAM converts to cache starting at the top of the L1D memory map and working downwards. That is, the highest addresses of L1D region 1 are the first to become cache.

On the C64x CPU, L2 includes L1D. This means that whatever data is contained within L1D cache is also contained in L2. The maximum amount of data contained in cache is therefore the size of L1P plus the size of L2. This could cause the L1D contents to be invalidated if L1P activity consumed a lot of the L2 cache.

On the C64x+ CPU, L2 and L1D are independent. The maximum amount of data contained in cache is therefore the size of L1P plus the size of L1D plus the size of L2. Because L1D is not contained in L2, evicting pages from L2 will not cause the data in L1D to become invalidated.

The differences between C64x and C64x+ CPU L1D memory is summarized in Table 5.

Table 5. L1D Comparison Between C64x and C64x+ CPUs

Internal Memory Structure	TMS320C64x CPU	TMS320C64x+ CPU
L1D size	16 KB	Device dependent. Will be a multiple of 16 KB with a maximum of 1024 Kbytes.
L1D organization	2-way set-associative	2-way set-associative
L1D CPU access time	1 cycle	1 cycle
L1D line size	64 bytes	64 bytes
L1D replacement strategy	2-way Least Recently Used	2-way Least Recently Used
L1D banking	8 × 32 bit banks	8 × 32 bit banks
L1D read miss policy	1 line allocated to L1D	1 line allocated to L1D
L1D read hit policy	Data read from L1D	Data read from L1D
L1D write miss policy	No allocation in L1D, data sent to L2	No allocation in L1D, data sent to L2
L1D write hit policy	Data is written into L1D location	Data is written into L1D location
L1D -> L2 request size	2 fetches/L1D line	2 fetches/L1D line
L1D protocol	Read Allocate; Pipelined Misses	Read Allocate; Pipelined Misses
L1D Memory	Single cycle RAM	Single cycle RAM
L1D -> L2 request CPU stall	6 cycles / RAM 8 cycles /L2 Cache hit	Depends on device

3.7.3 L2

On the C64x CPU, L2 consists of a single region of memory that can be configured as RAM, cache, or a combination of RAM and cache. It is always located with an origin of 0000 0000h.

On the C64x+ CPU, L2 consists of two regions that may have different characteristics. Some characteristics may depend upon the specific device definition. Some may be varied under program control at program run time. The two regions are not necessarily contiguous in memory and, in general, will not be located at 0000 0000h.

The following L2 characteristics are device dependent. Consult your device-specific data manual for more information:

- L2 size will be a multiple of 16 Kbytes. Maximum L2 size is 8192 KB. The maximum size of each of the two L2 regions is 4096 KB.
- L2 region 0 will always start on a 1 MB boundary, but may be configured to be at different addresses on different devices. Region 0 may be 0 KB in size (thus disabling it), or some multiple of 16 KB up to 4096 KB in size.
- L2 region 1 will have a size of some multiple of 16 KB up to a maximum size of 4096 KB
- The two L2 regions may have different access times depending on the type of memory used for each region.
- L2 will be configured as all RAM after device reset.

The following L2 characteristics are selectable at program run time under software control:

- L2 cache size may be adjusted to be 0 KB, 32 KB, 64 KB, 128 KB, or 256 Kbytes by writing to L2CFG.
- L2 RAM converts to cache starting at the top of the L2 region 0 memory map and working downwards. That is, the highest addresses of L2 region 0 are the first to become cache. Cache size cannot be greater than the region 0 size, and so some modes may not be available on certain devices.

L2 region 0 is typically used for the following purposes:

- L2 RAM
- L2 Cache

L2 region 1 is typically used for the following purposes:

- L2 RAM
- L2 ROM
- Shared L2 memory

The differences between C64x and C64x+ CPU L2 memory is summarized in Table 6.

Table 6. L2 Comparison Between C64x and C64x+ CPUs

Internal Memory	TMS320C64x CPU	TMS320C64x+ CPU
L2 Origin	0000 0000h	Varies. Refer to datasheet.
Number of L2 regions	1	2
L2 total size	Varies. Refer to datasheet.	Varies. Refer to datasheet.
L2 Cache Size	0/32/64/128/256 KB	0/32/64/128/256 KB
L2 RAM Size	Varies. Refer to datasheet.	Varies. Refer to datasheet.
L2 organization	4-way set-associative.	4-way set-associative.
L2 line size	128 bytes	128 bytes
L2 replacement strategy	4-way Least Recently Used	4-way Least Recently Used
L2 banking	8 × 64 bit banks	2 × 128 bit banks or 4 × 128 bit banks or 1 × 256 bit banks
L2 -> L1P protocol	Coherency invalidates	None
L2 -> L1D protocol	Coherency snoops and snoop invalidates	Coherency snoop reads and snoop writes
L2 protocol	Read and write allocate	Read and write allocate
L2 read miss action	Data is read via EDMA into newly allocated line in L2; requested data is passed to the requesting L1.	Data is read via EDMA into newly allocated line in L2; requested data is passed to the requesting L1.
L2 read hit action	Data read from L2	Data read from L2
L2 write miss action	Data is read via EDMA into newly allocated line in L2; write data is then written to the newly allocated line.	Data is read via EDMA into newly allocated line in L2; write data is then written to the newly allocated line.
L2 write hit action	Data is written into hit L2 location	Data is written into hit L2 location
L2-> L1P read path width	256 bit	256 bit
L2 -> L1D read path width	256 bit	256 bit
L1D -> L2 victim path width	64 bit	128 bit

3.7.4 Cache Operations

Cache operations available on the C64x CPU:

- L2 global writeback
- L2 global writeback and invalidate
- L2 block writeback
- L2 block writeback and invalidate
- L1P global invalidate
- L1D global invalidate
- L1P block invalidate
- L1D block writeback and invalidate

Eight new operations are added on the C64x+ CPU:

- L2 global invalidate
- L1D global writeback
- L1D global writeback and invalidate
- L1D block writeback
- L1D block invalidate
- L2 freeze
- L1P freeze
- L1D freeze

Each of these operations is initiated by writing to memory-mapped registers. See Appendix B for a list of the memory-mapped registers used by cache operations.

3.7.4.1 L1P and L1D Global Invalidate Operation

There are now two mechanisms for invalidating-without-writeback of the contents of L1P and L1D:

- The mechanism used on the C64x CPU (writing to the **IP** and **ID** bits in **CCFG** is retained), although the register is renamed to **L2CFG**.
- New memory-mapped registers are added (**L1PINV** and **L1DINV**) to control this function.

All new software should switch to using **L1DINV** and **L1PINV** for this function. Controlling this operation using the **IP** and **ID** bits is deprecated and the function may be removed from future devices.

3.7.4.2 L2 and L1D Global Invalidate Operation Not Safe

NOTE: It is generally not safe to globally invalidate either L2 or L1D, unless care is taken, modified data and stack contents may be lost.

3.7.4.3 L2WB and L2WBINV When L2 is in All RAM Mode

On the C64x CPU, L2 writeback (**L2WB**) and L2 writeback invalidate (**L2WBINV**) has no effect if L2 is in all RAM mode. On the C64x+ CPU, these operations work regardless of L2 mode. If no L2 cache is configured, these operations will still writeback the contents of L1D to external memory and invalidate L1D or L1P.

3.7.4.4 L2 Cache Coherence Operations and L1D

On the C64x CPU, the L2 global cache coherency commands fail to remove addresses from L1D if they are not also cached in L2. This might happen if data were cached in L1D prior to enabling L2.

On the C64x+ CPU, when an L2 global writeback commands will cause the updated L1D data to be written to L2/external memory even if the data is not currently cached in L2. A global L2 invalidate command will also invalidate L1D.

3.7.4.5 L2, L1P, and L1D Cache Freeze Operation

The C64x+ CPU adds the capability to freeze the L2, L1P, or L1D cache. During a cache freeze, the cache retains its current state. A program read of a frozen cache is identical to a read on an enabled cache except that on a cache miss, the data read from the external memory interface is not stored in the cache. The freeze mode can be used to ensure that critical program data is not overwritten in the cache.

L1P freeze is selected by the **OPER** bit in **L1PCC**. L1D freeze is selected by the **OPER** bit in **L1DCC**. L2 freeze is selected by the **L2CC** bit in **L2CFG**.

3.7.5 Local and Global Addresses

The C64x+ CPU is designed with the goal that it might be used in multi-CPU devices. To support this functionality, internal memory on the C64x+ CPU (L1P, L1D, and L2) effectively exists at two different address ranges in the memory map – a *global* address range and a local address range. EDMA operations always use the global address, not the local address.

- The global addresses are always in the range 1*N*x0 0000h to 1*N*xF FFFFh, where *N* is the CPU identifier.
- The local address is always the global address ANDed with 00FF FFFFh.

3.7.6 Changing Cache Size

Although the process is easier on the C64x+ CPU than on the C64x CPU, care must be exercised when changing the size of cache in L2, L1P, or L1D.

NOTE: Any change to cache size will force a global writeback/invalidate of the affected cache.

When changing from a smaller cache size to a larger cache size, some of the memory that is configured as RAM will be converted to cache and any data residing in that RAM will be lost. The program should:

1. Copy any needed data out of the affected range of L2, L1D, or L1P RAM.
2. Wait for the completion of any IDMA or EDMA issued in step 1.
3. Write the desired cache size change to the:
   a. **L2MODE** bit in **L2CFG** when changing L2 size.
   b. **L1PMODE** bit in **L1PCFG** when changing L1P size.
   c. **L1DMODE** bit in **L1DCFG** when changing the L1D size.
4. Read back the register that was used to change the cache size (**L2CFG**, **L1PCFG**, or **L1DCFG**). This will stall the CPU until the mode change completes.

When changing from a larger cache to a smaller cache is easier, since data is not lost in the change. The program should:

1. Write the desired cache size change to the:
   a. **L2MODE** bit in **L2CFG** when changing L2 size.
   b. **L1PMODE** bit in **L1PCFG** when changing L1P size.
   c. **L1DMODE** bit in **L1DCFG** when changing the L1D size
2. Read back the register that was used to change the cache size (**L2CFG**, **L1PCFG**, or **L1DCFG**). This will stall the CPU until the mode change completes.

3.7.7 Memory Protection

Each memory page on the C64x+ CPU has a permission tag that describes which other tasks, processors, or services have permission to read, write, or execute on that memory page.

All memory access requests (via CPU, IDMA, or EDMA) carry with the request a bit that defines whether the requestor was in supervisor mode or user mode and a field that describes the privilege associated with the requestor.

The permission tag specifies which memory access requests are honored based on the privilege associated with the requestor and whether the requestor was in supervisor mode or user mode.

If an invalid access is detected by the memory page, it will signal an exception.

Refer to the device-specific data manual for more information.

3.8 Privilege

The C64x+ CPU includes support for a form of protected-mode operation with a two-level system of privileged program execution. The C64x+ CPU privilege system is designed to support several objectives:

- Support the emergence of higher capability operating systems on the TMS320C6000™ DSP family architecture.
- Support more robust end-equipment, especially in conjunction with exceptions.
- Provide protection to support system features such as memory protection.

The support of 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
- 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 (pre-privilege) programs that do not comprehend a privilege system. Legacy programs run fully compatible with the understanding that undefined or illegal operations may behave differently on the C64x+ devices 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.

3.8.1 Execution Modes

There are two execution modes:

- Supervisor Mode
- User Mode

3.8.1.1 Supervisor Mode

Reset forces the C64x+ CPU into supervisor mode. A task operating in supervisor mode has full access to all system resources. Supervisor mode serves two purposes:

- It is the compatible execution mode.
- It is the privileged execution mode where all functions are available.

3.8.1.2 User Mode

An application executing in user mode has restricted access to system resources. An exception results in any attempt to access restricted resources while in user mode. The following registers are not available for use in user mode:

- Exception clear register (**ECR**)
- Exception flags register (**EFR**)
- Interrupt clear register (**ICR**)
- Interrupt enable register (**IER**)
- Interrupt flags register (**IFR**)
- Interrupt service table pointer register (**ISTP**)
- Interrupt task state register (**ITSR**)
- Restricted entry point address register (**REP**)
- The **PGIE**, **PWRD**, **PCC**, and **DCC** bits in the control status register (**CSR**) cannot be written in user mode.
- The only bits in the task state register (**TSR**) that can be written in user mode are the **GIE** and **SGIE** bits. No other bits in **TSR** can be written in user mode.

Data or program memory use may be restricted in user mode. The details will be different for different devices. Refer to the device-specific data manual for details.

3.8.2 Execution Mode Transitions

A task executing in supervisor mode can spawn a task executing in user mode, by placing the address of the entry point of the user mode task into the nonmaskable interrupt return pointer register (**NRP**), setting the nonmaskable interrupt/exception task state register (**NTSR**) to the desired state and executing a branch to that address using the **B NRP** instruction. The same effect can be accomplished using the **B IRP** instruction after initializing the interrupt return pointer register (**IRP**) and the interrupt task state register (**ITSR**) to the desired values. Prior to this, the restricted entry point address register (**REP**) should be initialized with the desired return address so that the user mode task can terminate with a return to supervisor mode using the **SWENR** instruction.

A maskable interrupt will place the CPU into supervisor mode and place the previous state into **ITSR**. Exiting the interrupt using the **B IRP** instruction will place the CPU into the original state.

A non-maskable interrupt or an exception will place the CPU into supervisor mode and place the previous state into **NTSR**. Exiting the interrupt or exception (if the exception is returnable) using the **B NRP** instruction will place the CPU into the original state.

A task executing in user mode may initiate a change to supervisor mode by executing either a **SWE** or **SWENR** instruction. The **SWE** instruction can be used when a privileged service is required. Any code necessary to interpret a user mode request should reside in the exception service routine. The **SWENR** instruction can be used to permanently return to supervisor mode from a user mode task, if **REP** was previously initialized to the correct return address while in supervisor mode.

Instruction Compatibility

The C64x and C64x+ DSPs share an instruction set. All of the instructions valid for the C64x DSP are also valid for the C64x+ DSP. Table 7 shows the instruction compatibility between the C64x and C64x+ DSPs. Some instructions are also available in compact form, see Section 3.3 for more information on compact instructions.

Table 7. Instruction Compatibility Between C64x and C64x+ DSPs

Instruction	TMS320C64x DSP	TMS320C64x+ DSP	
		Standard Form	Compact Form
ABS	√	√	
ABS2	√	√	
ADD	√	√	√
ADDAB	√	√	
ADDAB (with constant)		√	
ADDAD	√	√	
ADDAH	√	√	
ADDAH (with constant)		√	
ADDAW	√	√	
ADDAW (with constant)		√	√
ADDK	√	√	√
ADDKPC	√	√	
ADDSUB		√	
ADDSUB2		√	
ADDU	√	√	
ADD2	√	√	
ADD4	√	√	
AND	√	√	√
ANDN	√	√	
AVG2	√	√	
AVGU4	√	√	
B displacement	√	√	
B register	√	√	
B IRP	√	√	
B NRP	√	√	
BDEC	√	√	
BITC4	√	√	
BITR	√	√	
BNOP displacement	√	√	√
BNOP displacement (unitless)		√	
BNOP register	√	√	
BPOS	√	√	
CALLP		√	√

Table 7. Instruction Compatibility Between C64x and C64x+ DSPs (continued)

Instruction	TMS320C64x DSP	TMS320C64x+ DSP	
		Standard Form	Compact Form
CLR	√	√	√
CMPEQ	√	√	√
CMPEQ2	√	√	
CMPEQ4	√	√	
CMPGT	√	√	√
CMPGT2	√	√	
CMPGTU	√	√	√
CMPGTU4	√	√	
CMPLT	√	√	√
CMPLT2	√	√	
CMPLTU	√	√	√
CMPLTU4	√	√	
CMPY		√	
CMPYR		√	
CMPYR1		√	
DDOTP4		√	
DDOTPH2		√	
DDOTPH2R		√	
DDOTPL2		√	
DDOTPL2R		√	
DEAL	√	√	
DINT		√	
DMV		√	
DOTP2	√	√	
DOTPN2	√	√	
DOTPNRSU2	√	√	
DOTPNRUS2	√	√	
DOTPRSU2	√	√	
DOTPRUS2	√	√	
DOTPSU4	√	√	
DOTPUS4	√	√	
DOTPU4	√	√	
DPACK2		√	
DPACKX2		√	
EXT	√	√	√
EXTU	√	√	√
GMPY		√	
GMPY4	√	√	
IDLE	√	√	
LDB memory	√	√	√
LDBU memory	√	√	√
LDDW	√	√	√
LDH memory	√	√	√
LDHU memory	√	√	√
LDNDW	√	√	√
LDNW	√	√	√

Table 7. Instruction Compatibility Between C64x and C64x+ DSPs (continued)

Instruction	TMS320C64x DSP	TMS320C64x+ DSP	
		Standard Form	Compact Form
LDW memory	√	√	√
LMBD	√	√	
MAX2	√	√	
MAX2 (Also runs on .S1 and .S2 units)		√	
MAXU4	√	√	
MIN2	√	√	
MIN2 (Also runs on .S1 and .S2 units)		√	
MINU4	√	√	
MPY	√	√	√
MPYH	√	√	√
MPYHI	√	√	
MPYHIR	√	√	
MPYHL	√	√	√
MPYHLU	√	√	
MPYHSLU	√	√	
MPYHSU	√	√	
MPYHU	√	√	
MPYHULS	√	√	
MPYHUS	√	√	
MPYIH	√	√	
MPYIHR	√	√	
MPYIL	√	√	
MPYILR	√	√	
MPYLH	√	√	√
MPYLHU	√	√	
MYLI	√	√	
MPYLIR	√	√	
MPYLSHU	√	√	
MPYLUHS	√	√	
MPYSU	√	√	
MPYSU4	√	√	
MPYU	√	√	
MPYU4	√	√	
MPYUS	√	√	
MPYUS4	√	√	
MPY2	√	√	
MPY2IR		√	
MPY32		√	
MPY32SU		√	
MPY32U		√	
MPY32US		√	
MV	√	√	√
MVC	√	√	√
MVD	√	√	
MVK	√	√	√
MVKH	√	√	

Table 7. Instruction Compatibility Between C64x and C64x+ DSPs (continued)

Instruction	TMS320C64x DSP	TMS320C64x+ DSP	
		Standard Form	Compact Form
MVKL	√	√	
MVKLH	√	√	
NEG	√	√	√
NOP	√	√	√
NORM	√	√	
NOT	√	√	
OR	√	√	√
PACK2	√	√	
PACKH2	√	√	
PACKH4	√	√	
PACKHL2	√	√	
PACKLH2	√	√	
PACKL4	√	√	
RINT		√	
ROTL	√	√	
RPACK2		√	
SADD	√	√	√
SADD2	√	√	
SADDSUB		√	
SADDSUB2		√	
SADDSU2	√	√	
SADDUS2	√	√	
SADDU4	√	√	
SAT	√	√	
SET	√	√	√
SHFL	√	√	
SHFL3		√	
SHL	√	√	√
SHLMB	√	√	
SHR	√	√	√
SHR2	√	√	
SHRMB	√	√	
SHRU	√	√	√
SHRU2	√	√	
SMPY	√	√	√
SMPYH	√	√	√
SMPYHL	√	√	√
SMPYLH	√	√	√
SMPY2	√	√	
SMPY32		√	
SPACK2	√	√	
SPACKU4	√	√	
SPKERNEL		√	√
SPKERNELR		√	
SPLOOP		√	√
SPLOOPD		√	√

Table 7. Instruction Compatibility Between C64x and C64x+ DSPs (continued)

Instruction	TMS320C64x DSP	TMS320C64x+ DSP	
		Standard Form	Compact Form
SPLOOPW		√	
SPMASK		√	√
SPMASKR		√	√
SSHL	√	√	√
SSHVL	√	√	
SSHVR	√	√	
SSUB	√	√	√
SSUB2		√	
STB memory	√	√	√
STDW	√	√	√
STH memory	√	√	√
STNDW	√	√	√
STNW	√	√	√
STW memory	√	√	√
SUB	√	√	√
SUBAB	√	√	
SUBABS4	√	√	
SUBAH	√	√	
SUBAW	√	√	√
SUBC	√	√	
SUBU	√	√	
SUB2	√	√	
SUB4	√	√	
SWAP2	√	√	
SWAP4	√	√	
SWE		√	
SWENR		√	
UNPKHU4	√	√	
UNPKLU4	√	√	
XOR	√	√	√
XORMPY		√	
XPND2	√	√	
XPND4	√	√	
ZERO	√	√	

Internal Memory Control Registers

Table 8 compares the memory-mapped registers used to control the L1P, L1D, and L2 internal memory areas and cache for the C64x CPU and C64x+ CPU.

Table 8. Control Register Comparisons Between C64x and C64x+ CPUs

Memory Address	TMS320C64x CPU	TMS320C64x+ CPU	Purpose
0184 0000h	CCFG	L2CFG	Configure L2 cache mode
0184 0020h		L1PCFG	Configure L1 cache mode
0184 0024h		L1PCC	Sets/releases Freeze mode for L1P
0184 0040h		L1DCFG	Configure L1D cache mode
0184 0044h		L1DCC	Sets/releases Freeze mode for L1D
0184 2000h	L2ALLOC0		L2 allocation register 0
0184 2004h	L2ALLOC1		L2 allocation register 1
0184 2008h	L2ALLOC2		L2 allocation register 2
0184 200Ch	L2ALLOC3		L2 allocation register 3
0184 4000h	L2WBAR	L2WBAR	L2 block writeback base address
0184 4004h	L2WWC	L2WWC	L2 block writeback word count
0184 4010h	L2WIBAR	L2WIBAR	L2 block writeback /invalidate base address
0184 4014h	L2WIWC	L2WIWC	L2 block writeback /invalidate word count
0184 4018h	L2IBAR	L2IBAR	L2 block invalidate base address
0184 401Ch	L2IWC	L2IWC	L2 block invalidate word count
0184 4020h	L1PIBAR	L1PIBAR	L1P block invalidate base address
0184 4024h	L1PIWC	L1PIWC	L1P block invalidate word count
0184 4030h	L1DWIBAR	L1DWIBAR	L1D block writeback/invalidate base address
0184 4034h	L1DWIWC	L1DWIWC	L1D block writeback/invalidate word count
0184 4040h		L1DWBAR	L1D block writeback base address
0184 4044h		L1DWWC	L1D block writeback word count
0184 4048h	L1DIBAR	L1DIBAR	L1D block invalidate base address
0184 404Ch	L1DIWC	L1DIWC	L1D block invalidate word count
0184 5000h	L2WB	L2WB	L2 global writeback
0184 5004h	L2WBINV	L2WBINV	L2 global writeback/invalidate
0184 5008h		L2INV	L2 global invalidate
0184 5028h		L1PINV	L1P global invalidate
0184 5040h		L1DWB	L1D global writeback
0184 5044h		L1DWBINV	L1D global writeback/invalidate
0184 5048h		L1DINV	L1D global invalidate

Modified Instructions

This appendix lists the instructions that are functionally different on the C64x+ CPU from the equivalent instructions on the C64x CPU.

C.1 DOTPNRSU2

The **DOTPNRSU2** instruction 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 and additional round and shift step.

On the C64x CPU, the intermediate results prior to the round and shift step are maintained to a 32-bit precision, so that saturation is possible in the round and shift step.

On the C64x+ CPU, the intermediate results prior to the round and shift step are maintained to a 33-bit precision, so that saturation during the round and shift step will not happen.

C.2 DOTPNRUS2

The **DOTPNRUS2** instruction 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 and additional round and shift step.

On the C64x CPU, the intermediate results prior to the round and shift step are maintained to a 32-bit precision, so that saturation is possible in the round and shift step.

On the C64x+ CPU, the intermediate results prior to the round and shift step are maintained to a 33-bit precision, so that saturation during the round and shift step will not happen.

C.3 DOTPRSU2

The **DOTPRSU2** instruction returns the dot-product between two pairs of packed 16-bit values. This instruction takes the result of the dot-product and performs and additional round and shift step.

On the C64x CPU, the intermediate results prior to the round and shift step are maintained to a 32-bit precision, so that saturation is possible in the round and shift step.

On the C64x+ CPU, the intermediate results prior to the round and shift step are maintained to a 33-bit precision, so that saturation during the round and shift step will not happen.

C.4 DOTPRUS2

The **DOTPRUS2** instruction returns the dot-product between two pairs of packed 16-bit values. This instruction takes the result of the dot-product and performs and additional round and shift step.

On the C64x CPU, the intermediate results prior to the round and shift step are maintained to a 32-bit precision, so that saturation is possible in the round and shift step.

On the C64x+ CPU, the intermediate results prior to the round and shift step are maintained to a 33-bit precision, so that saturation during the round and shift step will not happen.

New Instructions

This appendix lists the instructions that are either new or have a new form on the C64x+ CPU that are not available on the C64 CPU.

D.1 ADDAB (with constant)

The **ADDAB** instruction adds two 32-bit integers.

On the C64x CPU, the two 32-bit values are contained in registers.

On the C64x+ CPU, a new form of this instruction is available:

```
ADDAB (.unit) B14/B15, ucst15, dst
```

This form adds a 15-bit unsigned constant to the base register (B14 or B15) and writes the result to a register (*dst*). This form of this instruction operates unconditionally and cannot be predicated.

D.2 ADDAH (with constant)

The **ADDAH** instruction adds two 32-bit integers. One of the two integers is left-shifted by one bit prior to the addition.

On the C64x CPU, the two 32-bit values are contained in registers.

On the C64x+ CPU, a new form of this instruction is available:

```
ADDAH (.unit) B14/B15, ucst15, dst
```

This form left-shifts a 15-bit unsigned constant, then adds the result to the base register (B14 or B15) and writes the result to a register (*dst*). This form of this instruction operates unconditionally and cannot be predicated.

D.3 ADDAW (with constant)

The **ADDAW** instruction adds two 32-bit integers. One of the two integers is left-shifted by two bits prior to the addition.

On the C64x CPU, the two 32-bit values are contained in registers.

On the C64x+ CPU, a new form of this instruction is available:

```
ADDAW (.unit) B14/B15, ucst15, dst
```

This form left-shifts a 15-bit unsigned constant by two bits, then adds the result to the base register (B14 or B15) and writes the result to a register (*dst*). This form of this instruction operates unconditionally and cannot be predicated.

D.4 ADDSUB

The **ADDSUB** instruction adds and subtracts two 32-bit integers in parallel and writes the results to a register pair (*dst_o:dst_e*):

```
ADDSUB (.unit) src1, src2, dst_o:dst_e
```

The following is performed in parallel:

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

The equivalent operation on the C64x CPU is:

```
ADD src1,src2,dst_e SUB src1,src2,dst_o
```

D.5 ADDSUB2

The **ADDSUB2** instruction adds and subtracts packed 16-bit integers in parallel and writes the results to a register pair (*dst_o:dst_e*):

```
ADDSUB2 (.unit) src1, src2, dst_o:dst_e
```

The following is performed in parallel:

1. The least-significant 16-bits of *src2* is added to least-significant 16-bits of *src1* and the (16-bit) result placed in the least-significant 16-bits of *dst_o*.
2. The most-significant 16-bits of *src2* is added to most-significant 16-bits of *src1* and the (16-bit) result placed in the most-significant 16-bits of *dst_o*.
3. The least-significant 16-bits of *src2* is subtracted from the least-significant 16-bits of *src1* and the (16-bit) result placed in the least-significant 16-bits of *dst_o*.
4. The most-significant 16-bits of *src2* is subtracted from the most-significant 16-bits of *src1* and the (16-bit) result placed in the most-significant 16-bits of *dst_o*.

The equivalent operation on the C64x CPU is:

```
ADD2 src1,src2,dst_e SUB2 src1,src2,dst_o
```

D.6 BNOP (without unit)

The **BNOP** instruction executes a relative branch followed by zero or more **NOP** instructions.

On the C64x CPU, the **BNOP** instruction operates on the .S1 or .S2 functional units.

On the C64x+ CPU, a new form of this instruction is available that does not use any of the 8 functional units:

```
BNOP  src2, src1
```

NOTE: The **BNOP** instruction does not affect the limit of 8 parallel instructions during any one cycle.

D.7 CALLP

The **CALLP** instruction executes a branch to a relative offset within a 21-bit range and the return address is placed in either the A3 or B3 register:

```
CALLP (.unit) label, A3/B3
```

If the **CALLP** executes on the .S1 unit, the return address is placed in A3. If **CALLP** executes on the .S2 unit, the return address is placed in B3.

D.8 CMPY

The **CMPY** instruction performs a complex multiply between two pairs of signed and packed 16-bit values:

```
CMPY (.unit) src1, src2, dst_o:dst_e
```

The values in *src1* and *src2* are treated as signed and packed 16-bit quantities.

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*.

The equivalent operation on the C64x CPU is:

```
DOTP2 src1,src2,dst_o SWAP2 src1,tmp DOTPNR2 src1,src2,dst_e
```

D.9 CMPYR

The **CMPYR** instruction performs a complex multiply (with rounding) between two pairs of signed and packed 16-bit values:

```
CMPYR (.unit) src1, src2, dst
```

The values in *src1* and *src2* are treated as signed and packed 16-bit quantities.

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 halfword 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 halfword of *dst*.

D.10 CMPYR1

The **CMPYR1** instruction performs a complex multiply (with rounding and left shift) between two pairs of signed and packed 16-bit values:

```
CMPYR1 (.unit) src1, src2, dst
```

The values in *src1* and *src2* are treated as signed and packed 16-bit quantities.

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^{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 halfword 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^{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 halfword of *dst*.

D.11 DDOTP4

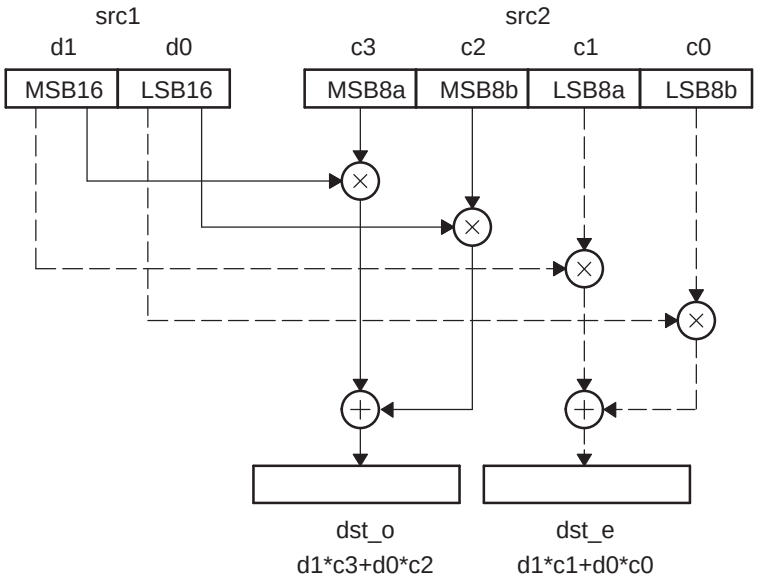
The **DDOTP4** instruction performs two **DDOTP2** operations simultaneously (see Figure 3):

```
DDOTP4 (.unit) src1, src2, dst_o:dst_e
```

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*.

Figure 3. DDOTP4 Instruction



D.12 DDOTPH2

The **DDOTPH2** instruction returns two dot-products between two pairs of signed and packed 16-bit values. The signed results are written to a 64-bit register pair (see Figure 4):

```
DDOTPH2 (.unit) src1_o:src1_e, src2, dst_o:dst_e
```

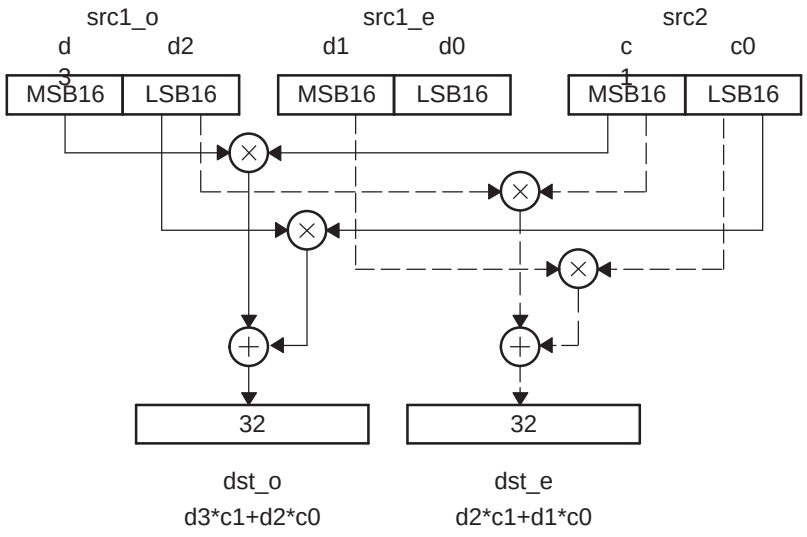
The product of the lower halfwords of *src1_o* and *src2* is added to the product of the upper halfwords of *src1_o* and *src2*. The result is written to *dst_o*.

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 written to *dst_e*.

The equivalent operation on the C64x CPU is:

```
PACKLH2 src1_O,src1_e,tmp DOTP2 src2,tmp,dst_o DOTP2 src2,src1_o,dst_e
```

Figure 4. DDOTPH2 Instruction



D.13 DDOTPH2R

The **DDOTPH2R** instruction returns two dot-products between two pairs of signed and packed 16-bit values. The signed results are rounded, shifted right by 16 and packed into a 32-bit register (*dst*):

```
DDOTPH2R (.unit) src1_o:src1_e, src2, dst
```

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*.

D.14 DDOTPL2

The **DDOTPL2** instruction returns two dot-products between two pairs of signed and packed 16-bit values. The signed results are written to a 64-bit register pair (see Figure 5):

```
DDOTPL2 (.unit) src1_o:src1_e, src2, dst_o:dst_e
```

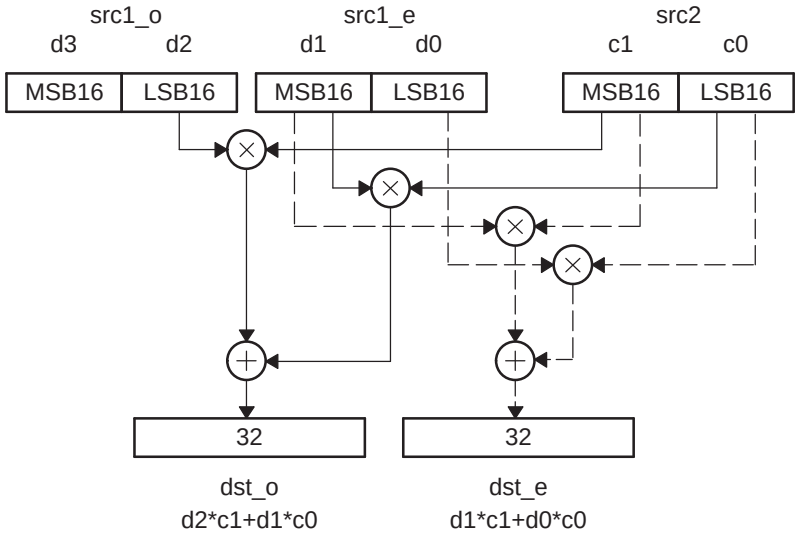
The product of the lower halfwords of *src1_e* and *src2* is added to the product of the upper halfwords of *src1_e* and *src2*. The result is written to *dst_e*.

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 written to *dst_o*.

The equivalent operation on the C64x CPU is:

```
PACKLH2 src1_O,src1_e,tmp DOTP2 src2,tmp,dst_o DOTP2 src2,src1_e,dst_e
```

Figure 5. DDOTPL2 Instruction



D.15 DDOTPL2R

The **DDOTPL2R** instruction returns two dot-products between two pairs of signed and packed 16-bit values. The signed results are rounded, shifted right by 16 and packed into a 32-bit register (*dst*):

```
DDOTPL2R (.unit) src1_o:src1_e, src2, dst
```

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*.

D.16 DINT

The **DINT** instruction disables interrupts in the current cycle:

```
DINT
```

On the C64x CPU, it is necessary to directly manipulate the **GIE** bit in the control status register (**CSR**) to globally disable or enable interrupts. The new **DINT** and **RINT** instructions removes this requirement on the C64x+ CPU and provides the ability to atomically enable or disable interrupts in a single instruction cycle.

The **DINT** instruction copies the contents of the **GIE** bit in the task state register (**TSR**) into the **SGIE** bit in **TSR**, and clears the **GIE** bit in both **TSR** and **CSR**. No functional unit is used by this instruction.

See Section D.28 for information about the **RINT** instruction.

D.17 DMV

The **DMV** instruction moves the contents of two independent registers to a register pair (*dst_o:dst_e*):

```
DMV (.unit) src1, src2, dst_o:dst_e
```

The contents of *src1* are written to *dst_o* and the contents of *src2* are written to *dst_e*.

The equivalent operation on the C64x CPU is:

```
MV src1,dst_o MV src2,dst_e
```

D.18 DPACK2

The **DPACK2** instruction executes a **PACK2** instruction in parallel with a **PACKH2** instruction (see Figure 6):

```
DPACK2 (.unit) src1, src2, dst_o:dst_e
```

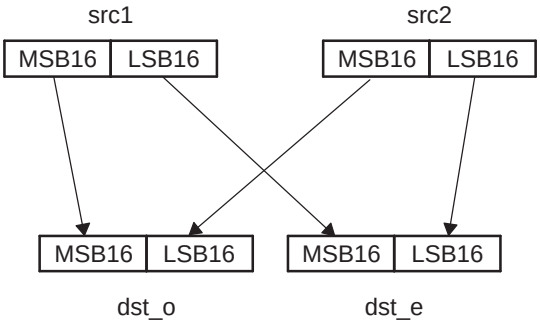
The lower halfword of *src1* is copied to the upper halfword of *dst_e*. The lower halfword of *src2* is copied to the lower halfword of *dst_e*.

The upper halfword of *src1* is copied to the upper halfword of *dst_o*. The upper halfword of *src2* is copied to the lower halfword of *dst_o*.

The equivalent operation on the C64x CPU is:

```
PACK2 src1,src2,dst_e PACKH2 src1,src2,dst_o
```

Figure 6. DPACK2 Instruction



D.19 DPACKX2

The **DPACKX2** instruction executes two **PACKLH2** instructions in parallel (see Figure 7):

```
DPACKX2 (.unit) src1, src2, dst_o:dst_e
```

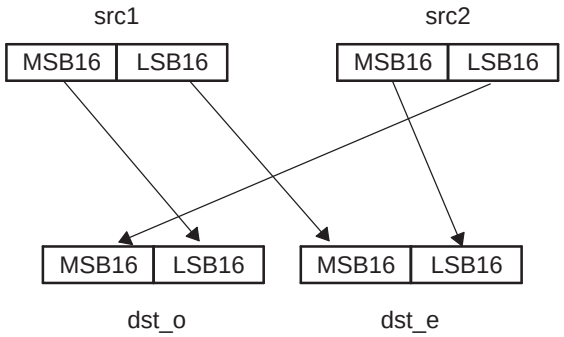
The lower halfword of *src1* is copied to the upper halfword of *dst_e*. The upper halfword of *src2* is copied to the lower halfword of *dst_e*.

The upper halfword of *src1* is copied to the lower halfword of *dst_o*. The lower halfword of *src2* is copied to the upper halfword of *dst_o*.

The equivalent operation on the C64x CPU is:

```
PACKLH2 src1,src2,dst_e PACKLH2 src2,src1,dst_o
```

Figure 7. DPACKX2 Instruction



D.20 GMPY

The **GMPY** instruction performs a Galois field multiply where *src1* is 32-bits and *src2* is limited to 9 bits. The polynomial comes from either the GPLYA or GPLYB control register depending on which side (A or B) the instruction operates:

```
GMPY (.unit) src1, src2, dst
```

D.21 MAX2 (operates on .S unit)

The **MAX2** instruction performs a maximum operation on signed and packed 16-bit values. For each pair of signed 16-bit values, **MAX2** places the larger value in the destination register.

On the C64x CPU, this instruction operates on the .L1 and .L2 functional units only.

On the C64x+ CPU, this instruction also operates on the .S1 and .S2 functional units in addition to the .L1 and .L2 functional units:

```
MAX2 (.unit) src1, src2, dst
```

D.22 MIN2 (operates on .S unit)

The **MIN2** instruction performs a minimum operation on signed and packed 16-bit values. For each pair of signed 16-bit values, **MIN2** places the smaller value in the destination register.

On the C64x CPU, this instruction operates on the .L1 and .L2 functional units only.

On the C64x+ CPU, this instruction also operates on the .S1 and .S2 functional units in addition to the .L1 and .L2 functional units:

```
MIN2 (.unit) src1, src2, dst
```

D.23 MPY2IR

The **MPY2IR** instruction performs two 16-bit by 32-bit multiplies:

```
MPY2IR (.unit) src1, src2, dst_o:dst_e
```

The most-significant bits of *src1* (treated as a signed 16-bit value) are multiplied by the 32-bit signed value in *src2*. The result is rounded to a 32-bit result by adding 2^{14} to it and right shifted by 15 bits then written to *dst_o*.

The least-significant bits of *src1* (treated as a signed 16-bit value) are multiplied by the 32-bit signed value in *src2*. The result is rounded to a 32-bit result by adding 2^{14} to it and right shifted by 15 bits then written to *dst_e*.

D.24 MPY32

The **MPY32** instruction performs a signed 32-bit by signed 32-bit multiply:

```
MPY32 (.unit) src1, src2, dst
```

The signed 32-bit in *src1* is multiplied by the signed 32-bit value in *src2*. The lower 32-bits of the 64-bit result are written to *dst*.

D.25 MPY32SU

The **MPY32SU** instruction performs a signed 32-bit by unsigned 32-bit multiply into a 64-bit signed result:

```
MPY32SU (.unit) src1, src2, dst_o:dst_e
```

The signed 32-bit in *src1* is multiplied by the signed 32-bit value in *src2*. The signed 64-bit value is written to the *dst_o:dst_e* register pair.

D.26 MPY32U

The **MPY32U** instruction performs an unsigned 32-bit by unsigned 32-bit multiply into a 64-bit unsigned result:

```
MPY32U (.unit) src1, src2, dst_o:dst_e
```

The unsigned 32-bit in *src1* is multiplied by the unsigned 32-bit value in *src2*. The unsigned 64-bit value is written to the *dst_o:dst_e* register pair.

D.27 MPY32US

The **MPY32US** instruction performs an unsigned 32-bit by signed 32-bit multiply into a 64-bit signed result:

```
MPY32US (.unit) src1, src2, dst_o:dst_e
```

The unsigned 32-bit in *src1* is multiplied by the signed 32-bit value in *src2*. The signed 64-bit value is written to the *dst_o:dst_e* register pair.

D.28 RINT

The **RINT** instruction restores the previous interrupt enable state after a **DINT** instruction:

```
RINT
```

On the C64x CPU, it is necessary to directly manipulate the **GIE** bit in the control status register (**CSR**) to globally disable or enable interrupts. The new **DINT** and **RINT** instructions removes this requirement on the C64x+ CPU and provides the ability to atomically enable or disable interrupts in a single instruction cycle.

The **RINT** instruction copies the contents of the **SGIE** bit in the task state register (**TSR**) into the **GIE** bit in **TSR** and **CSR**, and clears the **SGIE** bit in **TSR**. If the **GIE** bit is restored to 1, interrupts are enabled. No functional unit is used by this instruction.

See Section D.16 for information about the **DINT** instruction.

D.29 RPACK2

The **RPACK2** instruction packs the most-significant halfwords from two registers into the upper and lower register halves of the destination register (after left shifting and saturation of the source registers):

```
RPACK2 (.unit) src1, src2, dst
```

src1 and *src2* are left shifted with saturation. The 16 most-significant bits of *src1* are placed in the upper halfword of *dst*. The 16 most-significant bits of *src2* are placed in the lower halfword of *dst*.

The equivalent operation on the C64x CPU is:

```
PACK2 src1,src2,dst_e PACKH2 src1,src2,dst_o
```

D.30 SADDSUB

The **SADDSUB** instruction performs a simultaneous **SADD** and **SSUB** operation on common inputs:

```
SADDSUB (.unit) src1, src2, dst_o:dst_e
```

The contents of *src1* are added to the contents of *src2* with saturation. The result is placed in *dst_o*.

The contents of *src2* are subtracted (with saturation) from the contents of *src1*. The result is placed in *dst_e*.

D.31 SADDSUB2

The **SADDSUB2** instruction performs a simultaneous **SADD2** and **SSUB2** operation on common inputs:

```
SADDSUB2 (.unit) src1, src2, dst_o:dst_e
```

The signed and packed 16-bit values stored in the upper and lower halves of *src2* are added (with saturation) to the signed and packed 16-bit values stored in the upper and lower halves of *src1*. The results are written as signed and packed 16-bit values to *dst_e*.

The signed and packed 16-bit values stored in the upper and lower halves of *src2* are subtracted (with saturation) from the signed and packed 16-bit values stored in the upper and lower halves of *src1*. The results are written as signed and packed 16-bit values to *dst_o*.

The equivalent operation on the C64x CPU is:

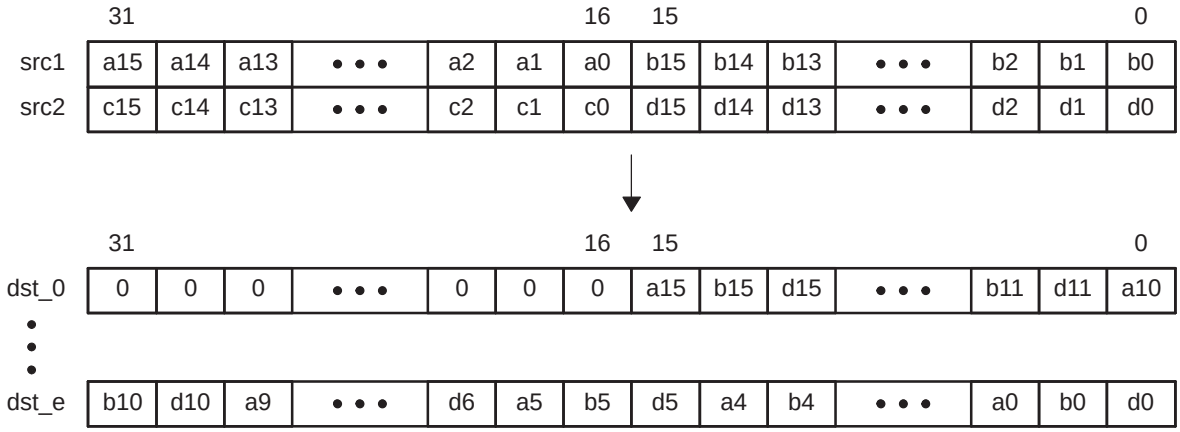
```
SADD2 src1,src2,dst_ SSUB2 src1,src2,dst_o
```

D.32 SHFL3

The **SHFL3** instruction performs a three-way interleave of three 16-bits value into a 48-bit result (see Figure 8):

```
SHFL3 (.unit) src1, src2, dst_o:dst_e
```

Figure 8. SHFL3 Instruction



D.33 SMPY32

The **SMPY32** instruction multiplies two signed 32-bit values together:

```
SMPY32 (.unit) src2, src1, dst
```

The two signed 32-bit values stored in *src1* and *src2* are multiplied together. The 64-bit result is left shifted by one bit with saturation. The 32 most-significant bits of the shifted value are stored in *dst*.

D.34 SPKERNEL

The **SPKERNEL** instruction marks the end of an SPLOOP block:

```
SPKERNEL (fstg, fcyc)
```

The **SPKERNEL** instruction is used when the loop does not need to be reloaded in to the buffer (that is, when the loop is not nested). The *fstg* and *fcyc* parameters indicate when the post epilog code starts. See Section 3.6 for more information.

D.35 SPKERNELR

The **SPKERNELR** instruction marks the end of an SPLOOP block:

```
SPKERNELR
```

The **SPKERNELR** instruction is used when the SPLOOP needs to be reloaded for a nested loop. See Section 3.6 for more information.

D.36 SPLOOP

The **SPLOOP** instruction marks the beginning of an SPLOOP block:

```
SPLOOP ii
```

The **SPLOOP** instruction is used to mark the beginning of an SPLOOP block. The *ii* parameter is the iteration interval and specifies how frequently new iterations of the loop are started. The value of the inner loop counter register (**ILC**) should be initialized at least three instruction cycles before the **SPLOOP** instruction. See Section 3.6 for more information.

D.37 SPLOOPD

The **SPLOOPD** instruction marks the beginning of an SPLOOP block:

```
SPLOOPD ii
```

The **SPLOOPD** instruction is used to mark the beginning of an SPLOOP block. The *ii* parameter is the iteration interval and specifies how frequently new iterations of the loop are started. Unlike the **SPLOOP** instruction, the value of the inner loop counter register (**ILC**) does not need to be initialized three instruction cycles before the **SPLOOPD** instruction; but the loop will execute at least four cycles before terminating. See Section 3.6 for more information.

D.38 SPLOOPW

The **SPLOOPW** instruction marks the beginning of an SPLOOP block:

```
SPLOOPW ii
```

The **SPLOOPW** instruction is used to mark the beginning of an SPLOOP block when a while loop is desired. The *ii* parameter is the iteration interval and specifies how frequently new iterations of the loop are started. Unlike the **SPLOOP** and **SPLOOPD** instructions, the **SPLOOPW** instruction does not use the inner loop counter register (**ILC**) to determine when to exit the loop. Instead, the predication condition on the instruction is used to determine the termination condition. See Section 3.6 for more information.

D.39 SPMASK

The **SPMASK** is used to inhibit the execution or loading into the buffer of specific instructions:

```
SPMASK unitmask
```

The **SPMASK** instruction is used to inhibit the execution of specified instruction from the SPLOOP buffer or to inhibit the loading of specified instructions into the SPLOOP buffer (although the instructions will execute normally). The *unitmask* parameter is used to specify which instructions are affected by this instruction. See Section 3.6 for more information.

D.40 SPMASKR

The **SPMASKR** is used to inhibit the execution or load into the buffer of specific instructions:

```
SPMASKR unitmask
```

The **SPMASKR** instruction is used to inhibit the execution of specified instruction from the SPLOOP buffer or to inhibit the loading of specified instructions into the SPLOOP buffer (although the instructions will execute normally). In addition, the **SPMASKR** instruction controls the reload point of nested loops.

The *unitmask* parameter is used to specify which instructions are affected by this instruction. See Section 3.6 for more information.

D.41 SSUB2

The **SSUB2** instruction subtracts two signed 16-bit integers that are stored in the upper and lower register halves:

```
SSUB2 (.unit) src1, src2, dst
```

The two signed 16-bit values stored in the upper and lower register halves in *src2* are subtracted from the two signed 16-bit values stored in the upper and lower register halves of *src1*. The result is placed in the upper and lower register halves of *dst*.

D.42 SWE

The **SWE** instruction causes an internal exception to be taken:

```
SWE
```

The **SWE** instruction is used as a mechanism for user mode programs to request supervisor mode services. Execution of the **SWE** instruction results in an internal exception being recognized. The **SXF** bit in the exception flag register (**EFR**) is set to 1. The **HWE** bit in the NMI/exception task state register (**NTSR**) is cleared to 0. If exceptions have been globally enabled, this causes an exception be recognized before execution of the next execute packet. The address of that next execute packet is placed in the nonmaskable interrupt return pointer register (**NRP**).

D.43 SWENR

The **SWENR** instruction causes an internal exception to be taken:

```
SWENR
```

The **SWENR** instruction 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:

- The task state register (**TSR**) is not copied into the NMI/exception task state register (**NTSR**).
- No return address is placed in the nonmaskable interrupt return pointer register (**NRP**), it remains unmodified.
- The **IB** bit in **TSR** is set to 1.
- Unlike the **SWE** instruction (which forces a branch to the exception vector pointed to by the interrupt service table pointer register (**ISTP**)), the **SWENR** instruction forces a branch to the address stored in the restricted entry point address register (**REP**).

D.44 XORMPY

The **XORMPY** instruction performs a Galois field multiply with zero-value polynomial:

```
XORMPY (.unit) src1, src2, dst
```

The **XORMPY** instruction performs a Galois field multiply, where *src1* is 32 bits and *src2* is limited to 9 bits. The **XORMPY** instruction is identical to the **GMPY** instruction executed with a zero-value polynomial.

IMPORTANT NOTICE FOR TI DESIGN INFORMATION AND RESOURCES

Texas Instruments Incorporated ('TI") technical, application or other design advice, services or information, including, but not limited to, reference designs and materials relating to evaluation modules, (collectively, "TI Resources") are intended to assist designers who are developing applications that incorporate TI products; by downloading, accessing or using any particular TI Resource in any way, you (individually or, if you are acting on behalf of a company, your company) agree to use it solely for this purpose and subject to the terms of this Notice.

TI's provision of TI Resources does not expand or otherwise alter TI's applicable published warranties or warranty disclaimers for TI products, and no additional obligations or liabilities arise from TI providing such TI Resources. TI reserves the right to make corrections, enhancements, improvements and other changes to its TI Resources.

You understand and agree that you remain responsible for using your independent analysis, evaluation and judgment in designing your applications and that you have full and exclusive responsibility to assure the safety of your applications and compliance of your applications (and of all TI products used in or for your applications) with all applicable regulations, laws and other applicable requirements. You represent that, with respect to your applications, you have all the necessary expertise to create and implement safeguards that (1) anticipate dangerous consequences of failures, (2) monitor failures and their consequences, and (3) lessen the likelihood of failures that might cause harm and take appropriate actions. You agree that prior to using or distributing any applications that include TI products, you will thoroughly test such applications and the functionality of such TI products as used in such applications. TI has not conducted any testing other than that specifically described in the published documentation for a particular TI Resource.

You are authorized to use, copy and modify any individual TI Resource only in connection with the development of applications that include the TI product(s) identified in such TI Resource. NO OTHER LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE TO ANY OTHER TI INTELLECTUAL PROPERTY RIGHT, AND NO LICENSE TO ANY TECHNOLOGY OR INTELLECTUAL PROPERTY RIGHT OF TI OR ANY THIRD PARTY IS GRANTED HEREIN, including but not limited to any patent right, copyright, mask work right, or other intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information regarding or referencing third-party products or services does not constitute a license to use such products or services, or a warranty or endorsement thereof. Use of TI Resources may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

TI RESOURCES ARE PROVIDED "AS IS" AND WITH ALL FAULTS. TI DISCLAIMS ALL OTHER WARRANTIES OR REPRESENTATIONS, EXPRESS OR IMPLIED, REGARDING TI RESOURCES OR USE THEREOF, INCLUDING BUT NOT LIMITED TO ACCURACY OR COMPLETENESS, TITLE, ANY EPIDEMIC FAILURE WARRANTY AND ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, AND NON-INFRINGEMENT OF ANY THIRD PARTY INTELLECTUAL PROPERTY RIGHTS.

TI SHALL NOT BE LIABLE FOR AND SHALL NOT DEFEND OR INDEMNIFY YOU AGAINST ANY CLAIM, INCLUDING BUT NOT LIMITED TO ANY INFRINGEMENT CLAIM THAT RELATES TO OR IS BASED ON ANY COMBINATION OF PRODUCTS EVEN IF DESCRIBED IN TI RESOURCES OR OTHERWISE. IN NO EVENT SHALL TI BE LIABLE FOR ANY ACTUAL, DIRECT, SPECIAL, COLLATERAL, INDIRECT, PUNITIVE, INCIDENTAL, CONSEQUENTIAL OR EXEMPLARY DAMAGES IN CONNECTION WITH OR ARISING OUT OF TI RESOURCES OR USE THEREOF, AND REGARDLESS OF WHETHER TI HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

You agree to fully indemnify TI and its representatives against any damages, costs, losses, and/or liabilities arising out of your non-compliance with the terms and provisions of this Notice.

This Notice applies to TI Resources. Additional terms apply to the use and purchase of certain types of materials, TI products and services. These include; without limitation, TI's standard terms for semiconductor products http://www.ti.com/sc/docs/stdterms.htm), evaluation modules, and samples (http://www.ti.com/sc/docs/sampterms.htm).

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