TMS320C621x/C671x DSP Two-Level Internal Memory Reference Guide

Literature Number: SPRU609
August 2002

Texas Instruments

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

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

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of that third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

Mailing Address:

Texas Instruments
Post Office Box 655303
Dallas, Texas 75265

Contents

Figures

Tables

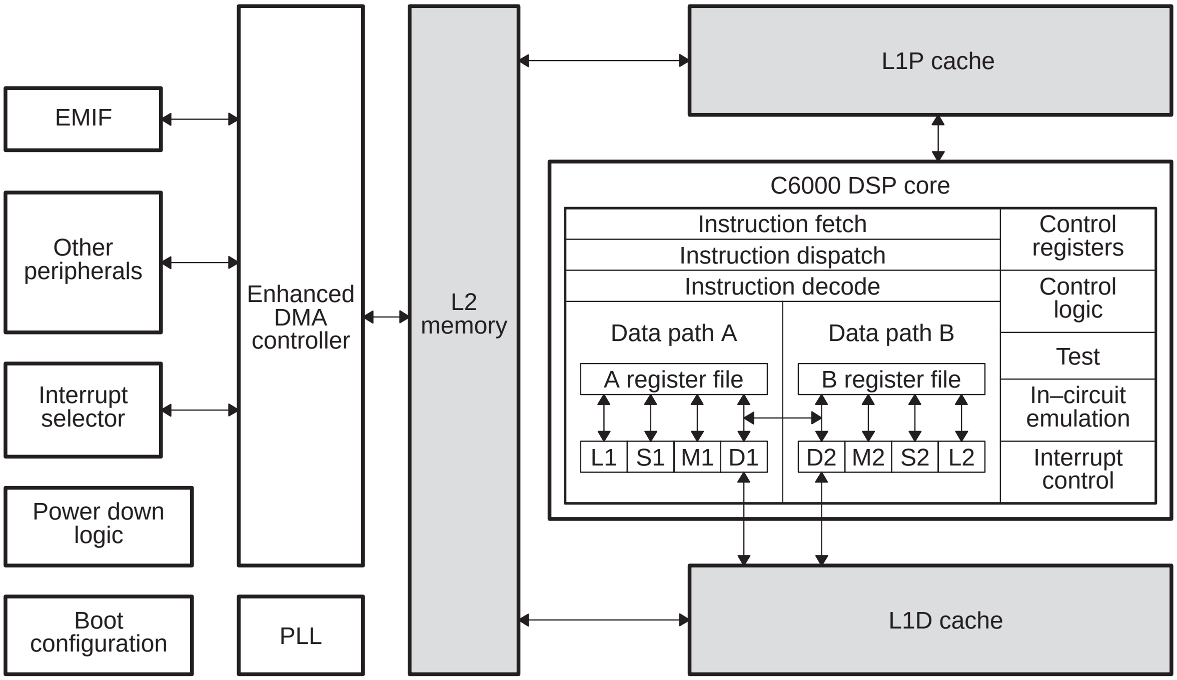
TMS320C621x/671x Two-Level Internal Memory

The TMS320C621x, TMS320C671x, and TMS320C64x™ digital signal processors (DSPs) of the TMS320C6000™ DSP family have a two-level memory architecture for program and data. The first-level program cache is designated L1P, and the first-level data cache is designated L1D. Both the program and data memory share the second-level memory, designated L2. L2 is configurable, allowing for various amounts of cache and SRAM. This document discusses the C621x/C671x two-level internal memory. For a discussion of the C64x two-level internal memory, see SPRU610.

1 Memory Hierarchy Overview

Figure 1 shows the block diagram of the C621x/C671x DSP. Table 1 summarizes the differences between the C621x/C671x and C64x internal memory. Figure 2 illustrates the bus connections between the CPU, internal memories, and the enhanced DMA (EDMA) of the C6000™ DSP.

Figure 1. TMS320C621x/C671x DSP Block Diagram



C6000, TMS320C64x, and TMS320C6000 are trademarks of Texas Instruments.

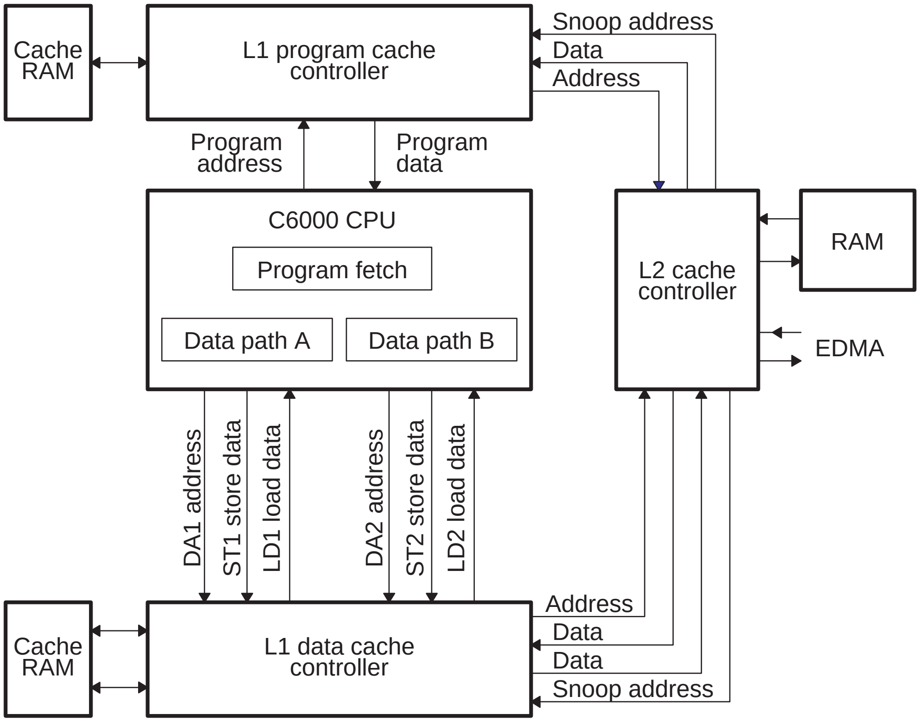
Table 1. TMS320C621x/C671x/C64x Internal Memory Comparison

	TMS320C621x/C671x	**TMS320C64x**
Internal memory structure	Two Level	
L1P size	4 Kbytes	16 Kbytes
L1P organization	Direct mapped	
L1P CPU access time	1 cycle	
L1P line size	64 bytes	32 bytes
L1P read miss action	1 line allocated in L1P	
L1P read hit action	Data read from L1P	
L1P write miss action	L1P writes not supported	
L1P write hit action	L1P writes not supported	
L1P –> L2 request size	2 fetches/L1P line	1 fetch/L1P line
L1P protocol	Read Allocate	Read Allocate; Pipelined Misses
L1P memory	Single-cycle RAM	
L1P –> L2 single request stall	5 cycles for L2 hit	8 cycles for L2 hit
L1P –> L2 minimum cycles between pipelined misses	Pipelined misses not supported	1 cycle
L1D size	4 Kbytes	16 Kbytes
L1D organization	2-way set associative	
L1D CPU access time	1 cycle	
L1D line size	32 bytes	64 bytes
L1D replacement strategy	2-way Least Recently Used	
L1D banking	64-bit-wide dual-ported RAM	8 × 32 bit banks
L1D read miss action	1 line allocated in L1D	
L1D read hit action	Data read from L1D	
L1D write miss action	No allocation in L1D, data sent to L2	
L1D write hit action	Data updated in L1D; line marked dirty	
L1D protocol	Read Allocate	Read allocate; Pipelined Misses
L1D –> L2 request size	2 fetches/L1D line	

Table 1. TMS320C621x/C671x/C64x Internal Memory Comparison (Continued)

	TMS320C621x/C671x	**TMS320C64x**
L1D –> L2 single request stall	4 cycles for L2 hit	6 cycles/L2 SRAM hit 8 cycles/L2 Cache hit
L1D –> L2 minimum cycles between pipelined misses	Pipelined misses not supported	2 cycles
L2 total size	Varies by part number. Refer to the data sheet for the specific device.	
L2 SRAM size	Varies by part number. Refer to the data sheet for the specific device.	
L2 Cache size	0/16/32/48/64 Kbytes	0/32/64/128/256 Kbytes
L2 organization	1/2/3/4-way set associative	4-way set associative Cache
L2 line size	128 bytes	
L2 replacement strategy	1/2/3/4-way Least Recently Used	4-way Least Recently Used
L2 banking	4 × 64 bit banks	8 × 64 bit banks
L2–L1P protocol	Coherency invalidates	
L2–L1D protocol	Coherency snoop-invalidates	Coherency snoops and snoop-invalidates
L2 protocol	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	
L2 read hit action	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.	
L2 write hit action	Data is written into hit L2 location	
L2 –> L1P read path width	256 bit	
L2 –> L1D read path width	128 bit	256 bit
L1D –> L2 write path width	32 bit	64 bit
L1D –> L2 victim path width	128 bit	256 bit
L2 –> EDMA read path width	64 bit	
L2 –> EDMA write path width	64 bit	

Figure 2. TMS320C621x/C671x Two-Level Internal Memory Block Diagram



2 Cache Terms and Definitions

Table 2 lists the terms used throughout this document that relate to the operation of the C621x/C671x two-level memory hierarchy.

Table 2. Terms and Definitions

Term	Definition
Allocation	The process of finding a location in the cache to store newly cached data. This process can include evicting data that is presently in the cache to make room for the new data.
Associativity	The number of line frames in each set. This is specified as the number of ways in the cache.
Capacity miss	A cache miss that occurs because the cache does not have sufficient room to hold the entire working set for a program. Compare with compulsory miss and conflict miss.
Clean	A cache line that is valid and whose contents match those held in the next lower level of memory. The opposite state for a valid cache line is dirty.
Coherence	Informally, a memory system is coherent if any read of a data item returns the most recently written value of that data item. This includes accesses by the CPU and the EDMA. Cache coherence is covered in more detail in section 7.1.
Compulsory miss	Sometimes referred to as a “first-reference miss”. A compulsory miss is a cache miss that must occur because the data has had no prior opportunity to be allocated in the cache. Typically, compulsory misses for particular pieces of data occur on the first access of that data. However, some cases can be considered compulsory even if they are not the first reference to the data. Such cases include repeated write misses on the same location in a cache that does not write-allocate, and cache misses to noncacheable locations. Compare with capacity miss and conflict miss.
Conflict miss	A cache miss that occurs due to the limited associativity of a cache, rather than due to capacity constraints. A fully-associative cache is able to allocate a newly cached line of data anywhere in the cache. Most caches have much more limited associativity (see set-associative cache), and so are restricted in where they may place data. This results in additional cache misses that a more flexible cache would not experience.
Direct-mapped cache	A direct-mapped cache maps each address in the lower levels of memory to a single location in the cache. Multiple locations may map to the same location in the cache. This is in contrast to a multi-way set-associative cache, which selects a place for the data from a set of locations in the cache. A direct-mapped cache can be considered a single-way set-associative cache.
Dirty	In a writeback cache, writes that reach a given level in the memory hierarchy may update that level, but not the levels below it. Thus, when a cache line is valid and contains updates that have not been sent to the next lower level, that line is said to be dirty. The opposite state for a valid cache line is clean.

Table 2. Terms and Definitions (Continued)

Term	Definition
DMA	Direct Memory Access. Typically, a DMA operation copies a block of memory from one range of addresses to another, or transfers data between a peripheral and memory. On the C621x/C671x DSP, DMA transfers are performed by the enhanced DMA (EDMA) engine. These DMA transfers occur in parallel to program execution. From a cache coherence standpoint, EDMA accesses can be considered accesses by a parallel processor.
Eviction	The process of removing a line from the cache to make room for newly cached data. Eviction can also occur under user control by requesting a writeback-invalidate for an address or range of addresses from the cache. The evicted line is referred to as the victim. When a victim line contains dirty data, the data must be written out to the next level memory to maintain coherency.
Execute packet	A block of instructions that begin execution in parallel in a single cycle. An execute packet may contain between 1 and 8 instructions.
Fetch packet	A block of 8 instructions that are fetched in a single cycle. One fetch packet may contain multiple execute packets, and thus may be consumed over multiple cycles.
First-reference miss	A cache miss that occurs on the first reference to a piece of data. First-reference misses are a form of compulsory miss.
Fully-associative cache	A cache that allows any memory address to be stored at any location within the cache. Such caches are very flexible, but usually not practical to build in hardware. They contrast sharply with direct-mapped caches and set-associative caches, both of which have much more restrictive allocation policies. Conceptually, fully-associative caches are useful for distinguishing between conflict misses and capacity misses when analyzing the performance of a direct-mapped or set-associative cache. In terms of set-associative caches, a fully-associative cache is equivalent to a set-associative cache that has as many ways as it does line frames, and that has only one set.
Higher-level memory	In a hierarchical memory system, higher-level memories are memories that are closer to the CPU. The highest level in the memory hierarchy is usually the Level 1 caches. The memories at this level exist directly next to the CPU. Higher-level memories typically act as caches for data from lower levels of memory.
Hit	A cache hit occurs when the data for a requested memory location is present in the cache. The opposite of a hit is a miss. A cache hit minimizes stalling, since the data can be fetched from the cache much faster than from the source memory. The determination of hit versus miss is made on each level of the memory hierarchy separately—a miss in one level may hit in a lower level.

Table 2. Terms and Definitions (Continued)

Term	Definition
Invalidate	The process of marking valid cache lines as invalid in a particular cache. Alone, this action discards the contents of the affected cache lines, and does not write back any dirty data. When combined with a writeback, this effectively updates the lower level of memory while completely removing the cached data from the given level of memory. Invalidates combined with writebacks are referred to as writeback-invalidates, and are commonly used for retaining coherence between caches.
Least Recently Used (LRU) allocation	For set-associative and fully-associative caches, least-recently used allocation refers to the method used to choose among line frames in a set when allocating space in the cache. When all of the line frames in the set that the address maps to contain valid data, the line frame in the set that was read or written the least recently (furthest back in time) is selected to hold the newly cached data. The selected line frame is then evicted to make room for the new data.
Line	A cache line is the smallest block of data that the cache operates on. The cache line is typically much larger than the size of data accesses from the CPU or the next higher level of memory. For instance, although the CPU may request single bytes from memory, on a read miss the cache reads an entire line’s worth of data to satisfy the request.
Line frame	A location in a cache that holds cached data (one line), an associated tag address, and status information for the line. The status information can include whether the line is valid, dirty, and the current state of that line’s LRU.
Line size	The size of a single cache line, in bytes.
Load through	When a CPU request misses both the first-level and second-level caches, the data is fetched from the external memory and stored to both the first-level and second-level cache simultaneously. A cache that stores data and sends that data to the upper-level cache at the same time is a load-through cache. Using a load-through cache reduces the stall time compared to a cache that first stores the data in a lower level and then sends it to the higher-level cache as a second step.
Long-distance access	Accesses made by the CPU to a noncacheable memory. Long-distance accesses are used when accessing external memory that is not marked as cacheable.
Lower-level memory	In a hierarchical memory system, lower-level memories are memories that are further from the CPU. In a C621x/C671x system, the lowest level in the hierarchy includes the system memory below L2 and any memory-mapped peripherals.
LRU	Least Recently Used. See Least Recently Used allocation for a description of the LRU replacement policy. When used alone, LRU usually refers to the status information that the cache maintains for identifying the least-recently used line in a set. For example, consider the phrase “accessing a cache line updates the LRU for that line.”

Table 2. Terms and Definitions (Continued)

Term	Definition
Memory ordering	Defines what order the effects of memory operations are made visible in memory. (This is sometimes referred to as consistency.) Strong memory ordering at a given level in the memory hierarchy indicates it is not possible to observe the effects of memory accesses in that level of memory in an order different than program order. Relaxed memory ordering allows the memory hierarchy to make the effects of memory operations visible in a different order. Note that strong ordering does not require that the memory system execute memory operations in program order, only that it makes their effects visible to other requestors in an order consistent with program order. Section 7.3 covers the memory ordering assurances that the C621x/C671x memory hierarchy provides.
Miss	A cache miss occurs when the data for a requested memory location is not in the cache. A miss may stall the requestor while the line frame is allocated and data is fetched from the next lower level memory. In some cases, such as a CPU write miss from L1D, it is not strictly necessary to stall the CPU. Cache misses are often divided into three categories: compulsory misses, conflict misses, and capacity misses.
Miss pipelining	The process of servicing a single cache miss is pipelined over several cycles. By pipelining the miss, it is possible to overlap the processing of several misses, should many occur back-to-back. The net result is that much of the overhead for the subsequent misses is hidden, and the incremental stall penalty for the additional misses is much smaller than that for a single miss taken in isolation.
Read allocate	A read-allocate cache only allocates space in the cache on a read miss. A write miss does not cause an allocation to occur unless the cache is also a write-allocate cache. For caches that do not write-allocate, the write data would be passed on to the next lower-level cache.
Set	A collection of line-frames in a cache that a single address can potentially reside. A direct-mapped cache contains one line-frame per set, and an N-way set-associative cache contains N line-frames per set. A fully-associative cache has only one set that contains all of the line-frames in the cache.
Set-associative cache	A set-associative cache contains multiple line-frames that each lower-level memory location can be held in. When allocating room for a new line of data, the selection is made based on the allocation policy for the cache. The C621x/C671x devices employ a Least Recently Used allocation policy for its set-associative caches.
Snoop	A method by which a lower-level memory queries a higher-level memory to determine if the higher-level memory contains data for a given address. The primary purpose of snoops is to retain coherency, by allowing a lower-level memory to request updates from a higher-level memory. A snoop operation may trigger a writeback, or more commonly, a writeback-invalidate. Snoops that trigger writeback-invalidates are sometimes called snoop-invalidates.

Table 2. Terms and Definitions (Continued)

Term	Definition
Tag	A storage element containing the most-significant bits of the address stored in a particular line. Tag addresses are stored in special tag memories that are not directly visible to the CPU. The cache queries the tag memories on each access to determine if the access is a hit or a miss.
Thrash	An algorithm is said to thrash the cache when its access pattern causes the performance of the cache to suffer dramatically. Thrashing can occur for multiple reasons. One possible situation is that the algorithm is accessing too much data or program code in a short time frame with little or no reuse. That is, its working set is too large, and thus the algorithm is causing a significant number of capacity misses. Another situation is that the algorithm is repeatedly accessing a small group of different addresses that all map to the same set in the cache, thus causing an artificially high number of conflict misses.
Touch	A memory operation on a given address is said to touch that address. Touch can also refer to reading array elements or other ranges of memory addresses for the sole purpose of allocating them in a particular level of the cache. A CPU-centric loop used for touching a range of memory in order to allocate it into the cache is often referred to as a touch loop. Touching an array is a form of software-controlled prefetch for data.
Valid	When a cache line holds data that has been fetched from the next level memory, that line-frame is valid. The invalid state occurs when the line-frame holds no data, either because nothing has been cached yet, or because previously cached data has been invalidated for whatever reason (coherence protocol, program request, etc.). The valid state makes no implications as to whether the data has been modified since it was fetched from the lower levels of memory; rather, this is indicated by the dirty or clean state of the line.
Victim	When space is allocated in a set for a new line, and all of the line-frames in the set that the address maps to contain valid data, the cache controller must select one of the valid lines to evict in order to make room for the new data. Typically, the least-recently used line is selected. The line that is evicted is known as the victim line. If the victim line is dirty, its contents are written to the next lower level of memory using a victim writeback.
Victim Buffer	A special buffer that holds victims until they are written back. Victim lines are moved to the victim buffer to make room in the cache for incoming data.
Victim Writeback	When a dirty line is evicted (that is, a line with updated data is evicted), the updated data is written to the lower levels of memory. This process is referred to as a victim writeback.

Table 2. Terms and Definitions (Continued)

Term	Definition
Way	In a set-associative cache, each set in the cache contains multiple line-frames. The number of line-frames in each set is referred to as the number of ways in the cache. The collection of corresponding line-frames across all sets in the cache is called a way in the cache. For instance, a 4-way set-associative cache has 4 ways, and each set in the cache has 4 line-frames associated with it, one associated with each of the 4 ways. As a result, any given cacheable address in the memory map has 4 possible locations it can map to in a 4-way set-associative cache.
Working set	The working set for a program or algorithm is the total set of data and program code that is referenced within a particular period of time. It is often useful to consider the working set on an algorithm-by-algorithm basis when analyzing upper levels of memory, and on a whole-program basis when analyzing lower levels of memory.
Write allocate	A write-allocate cache allocates space in the cache when a write miss occurs. Space is allocated according to the cache’s allocation policy (LRU, for example), and the data for the line is read into the cache from the next lower level of memory. Once the data is present in the cache, the write is processed. For a writeback cache, only the current level of memory is updated—the write data is not immediately passed to the next level of memory.
Writeback	The process of writing updated data from a valid but dirty cache line to a lower-level memory. After the writeback occurs, the cache line is considered clean. Unless paired with an invalidate (as in writeback-invalidate), the line remains valid after a writeback.
Writeback cache	A writeback cache will only modify its own data on a write hit. It will not immediately notify the next lower-level memory that the write occurred. The data will be written back at some future point, such as when the cache line is evicted, or when the lower-level memory snoops the address from the higher-level memory. It is also possible to directly initiate a writeback for a range of addresses using cache control registers. A write hit to a writeback cache causes the corresponding line to be marked as dirty—that is, the line contains updates that have yet to be sent to the lower levels of memory.
Writeback-invalidate	A writeback operation followed by an invalidation. See writeback and invalidate. On the C621x/C671x devices, a writeback-invalidate on a group of cache lines only writes out data for dirty cache lines, but invalidates the contents of all of the affected cache lines.

Table 2. Terms and Definitions (Continued)

Term	Definition
Write merging	Write merging combines multiple independent writes into a single, larger write. This improves the performance of the memory system by reducing the number of individual memory accesses it needs to process. For instance, on the C64x device, the L1D store buffer can merge multiple writes under some circumstances if they are to the same double-word address. In this example, the result is a larger effective store-buffer capacity and a lower bandwidth impact on L2.
Write-through cache	A write-through cache passes all writes to the lower-level memory. It never contains updated data that it has not passed on to the lower level. As a result, cache lines can never be dirty in a write-through cache. The C621x/C671x devices do not utilize write-through caches.

3 Level 1 Data Cache (L1D)

The level 1 data cache (L1D) services data accesses from the CPU. The following sections describe the parameters and operation of the L1D. The operation of L1D is controlled by various registers, as described in section 6, *Memory System Controls*.

3.1 L1D Parameters

The L1D is a 4K-byte cache. It is a two-way set associative cache with a 32-byte line size and 64 sets. It also features a 32-bit by 4-entry write buffer between L1D and the L2 memory.

Physical addresses map onto the cache in a straightforward manner. The physical address divides into three fields as shown in Figure 3. Bits 4–0 of the address specify an offset within the line. Bits 10–5 of the address select one of the 64 sets within the cache. Bits 31–11 of the address serve as the tag for the line.

Figure 3. L1D Address Allocation

31 11	10 5	4 0
Tag	Set Index	Offset

Because L1D is a two-way cache, each set contains two cache lines, one for each way. On each access, the L1D compares the tag portion of the address for the access to the tag information for both lines in the appropriate set. If the tag matches one of the lines and that line is marked valid, the access is a hit. If these conditions are not met, the access is a miss. Miss penalties are discussed in section 3.2, *L1D Performance*.

The L1D is a read-allocate-only cache. This means that new lines are allocated in L1D for read misses, but not for write misses. For this reason, a 4-entry write buffer exists between the L1D and L2 caches that captures data from write misses. The write buffer is described in section 3.2.3, *L1D Write Buffer*.

The L1D implements a least-recently used (LRU) line allocation policy. This means that on an L1D read miss, the L1D evicts the least-recently read or written line within a set in order to make room for the incoming data. Note that invalid lines are always considered least-recently used. If the contents of the selected line is dirty, then the victim line's data is written to L2 as a victim write-back prior to fetching the new data.

3.2 L1D Performance

3.2.1 L1D Memory Banking

The C621x/C671x DSP does not employ a banked memory structure in L1D. The L1D is implemented with a single bank of dual-ported, 64-bit memory. This allows two simultaneous accesses on each cycle with no stalls. This is in contrast to the C620x/C670x and C64x devices. The C64x devices employ a least-significant bit (LSB) based memory banking structure that only allows one access to each bank on each cycle.

3.2.2 L1D Miss Penalty

The L1D can service up to two data accesses from the CPU every cycle. Accesses that hit L1D complete without stalls.

Reads that miss L1D stall the CPU while the requested data is fetched. The L1D is a read-allocate cache, and so it will allocate a new line for the requested data, as described in section 3.1. An L1D read miss that hits L2 SRAM or L2 cache stalls the CPU for 4 cycles. This assumes there is no other memory traffic in L2 that delays the processing of requests from L1D. Section 5.4, *L2 Interfaces*, discusses interactions between the various requestors that access L2.

An L1D read miss that also misses L2 stalls the CPU while the L2 retrieves the data from external memory. Once the data is retrieved, it is stored in L2 and transferred to the L1D. The external miss penalty varies depending on the type and width of external memory used to hold external data, as well as other aspects of system loading. Section 5.2 describes how L2 handles cache misses on behalf of L1D.

If there are two read misses to the same line in the same cycle, only one miss penalty is incurred. Similarly, if there are two accesses in succession to the same line and the first one is a miss, the second access will not incur any additional miss penalty.

The process of allocating a line in L1D can result in a victim writeback. Victim writebacks move updated data out of L1D to the lower levels of memory. When updated data is evicted from L1D, the cache moves the data to the victim buffer. Once the data is moved to the victim buffer, the L1D resumes processing of the current read miss. Further processing of the victim writeback occurs in the background. Subsequent read and write misses, however, must wait for the victim writeback to be processed. As a result, victim writebacks can noticeably lengthen the time for servicing cache misses.

Write misses do not stall the CPU directly. Rather, write misses are queued in the write buffer that exists between L1D and L2. Although the CPU does not always stall for write misses, the write buffer can stall the CPU under various circumstances. Section 3.2.3 describes the effects of the write buffer.

3.2.3 L1D Write Buffer

The L1D does not write allocate. Rather, write misses are passed directly to L2 without allocating a line in L1D. A write buffer exists between the L1D cache and the L2 memory to capture these write misses. The write-buffer provides a 32-bit path for writes from L1D to L2 with room for four outstanding write requests.

Writes that miss L1D do not stall the CPU unless the write buffer is full. If the write buffer is full, a write miss will stall the CPU until there is room in the buffer for the write. The write buffer can also indirectly stall the CPU by extending the time for a read miss. Reads that miss L1D will not be processed as long as the write buffer is not empty. Once the write buffer has emptied, the read miss will be processed. This is necessary as a read miss may overlap an address for which a write is pending in the write buffer.

The L2 can process a new request from the write buffer once every 2 cycles, provided that the requested L2 bank is not busy. Section 5.3 describes the L2 banking structure and its impact on performance.

4 Level 1 Program Cache (L1P)

The level 1 program cache (L1P) services program fetches from the CPU. The following sections describe the parameters and operation of the L1P. The operation of L1P is controlled by various registers, as described in section 6, *Memory System Controls*.

4.1 L1P Parameters

The L1P is a 4K-byte cache. It is a direct-mapped cache with a 64-byte line size and 64 sets.

Physical addresses map onto the cache in a fixed manner. The physical address divides into three fields as shown in Figure 4. Bits 5–0 of the address specify an offset within a set. Bits 11–6 of the address select one of the 64 sets within the cache. Bits 31–12 of the address serve as the tag for the line.

Figure 4. L1P Address Allocation

31 12	11 6	5 0
Tag	Set Index	Offset

Because L1P is direct-mapped cache, each address maps to a fixed location in the cache. That is, each set contains exactly one line frame. On a cache miss, the cache allocates the corresponding line for the incoming data. Because L1P does not support writes from the CPU, the previous contents of the line are discarded.

4.2 L1P Miss Penalty

A program fetch that hits L1P completes in a single cycle without stalling the CPU. An L1P miss that hits in L2 stalls the CPU for 5 cycles, regardless of the parallelism of the surrounding code. This assumes there is no other memory traffic in L2 that delays the processing of requests from L1P. Section 5.4, *L2 Interfaces*, discusses iterations between the various requestors that access L2.

An L1P miss that misses in L2 stalls the CPU until the L2 retrieves the data from external memory and transfers the data to the L1P, which then returns the data to the CPU. This delay depends upon the type of external memory used to hold the external program, as well as other aspects of system loading.

5 Level 2 Unified Memory (L2)

The level 2 unified memory (L2) can operate as SRAM, cache, or both. It services cache misses from both L1P and L1D, as well as DMA accesses using the EDMA controller. The following sections describe the parameters and operation of the L2. The operation of L2 is controlled by various registers, as described in section 6, *Memory System Controls*.

5.1 L2 Cache and L2 SRAM

The L2 memory can operate as SRAM, as cache, or as both, depending on its mode. L2 SRAM refers to the portion of L2 mapped as SRAM and L2 Cache refers to the portion of L2 that acts as cache. For a given device, the total capacity of L2 SRAM and L2 Cache together is fixed, regardless of the L2 mode.

The total size of the L2 depends on the specific C621x/C671x device. The C6211, C6711, and C6712 devices provide 64K bytes of L2 memory. The C6713 device provides 256K bytes of L2 memory. For other C621x/C671x devices, consult the data sheet to determine the L2 size for the device.

After reset, the entire L2 memory is mapped as SRAM. The L2 SRAM is contiguous and always starts at address 0000 0000h in the memory map. The C6211, C6711, and C6712 devices map L2 SRAM over the address range 0000 0000h to 0000 FFFFh. The C6713 device maps L2 SRAM over the address range 0000 0000h to 0003 FFFFh.

The L2 Cache is enabled by the L2MODE field in the cache configuration register (CCFG). Enabling L2 Cache reduces the amount of available L2 SRAM. Section 6.1.3, *L2 Mode Selection Using L2MODE Field in CCFG*, discusses how the L2 memory map varies according to cache mode and the specific device being used.

The L2 Cache is a set associative cache whose capacity varies between 16K bytes and 64K bytes depending on its mode. The cache is organized as 128 sets with a 128-byte line size. The associativity varies between 1-way and 4-way as the capacity changes. Therefore, external physical memory addresses map onto the L2 Cache identically, regardless of the cache mode. The physical address divides into three fields as shown in Figure 5. Bits 6–0 of the address specify an offset within a line. Bits 13–7 of the address select one of the 128 sets within the cache. Bits 31–14 of the address serve as the tag for the line.

Figure 5. L2 Address Allocation (All L2 Cache Modes)

31 14	13 7	6 0
Tag	Set Index	Offset

5.2 L2 Operation

When L2 Cache is enabled, it services requests from L1P and L1D to external addresses. The operation of the L2 Cache is similar to the operation of both L1P and L1D. When a request is made, the L2 first determines if the address requested is present in the cache. If the address is present, the access is considered a cache hit and the L2 services the request directly within the cache. If the address is not present, the access results in a cache miss. On a miss, the L2 processes the request according to the cacheability of the affected address.

On a cache hit, the L2 updates the LRU status for the corresponding set in L2 cache. If the access is a read, the L2 returns the requested data. If the access is a write, the L2 updates the contents of the cache line and marks the line as dirty. L2 is a writeback cache, and so write hits in L2 are not immediately forwarded to external memory. The external memory will be updated when this line is later evicted, or is written back using the block-writeback control registers described in section 6.3.2, *Block Cache Operations*.

The L2 allocates a new line within the L2 Cache on a cache miss to a cacheable external memory location. Note that unlike L1D, the L2 allocates a line for both read and write misses. The L2 Cache implements a least-recently used policy to select a line within the set to allocate on a cache miss. If this line being replaced is valid, it is evicted as described below. Once space is allocated for the new data, an entire L2 line's worth of data is fetched via the EDMA into the allocated line.

Evicting a line from L2 requires several steps, regardless of whether the victim is clean or dirty. For each line in L2, L2 tracks whether the given line may also be cached in L1D. If it detects that the victim line might be present in L1D, it sends L1D a snoop-invalidate request. L1D responds by invalidating the corresponding line if it is indeed caching that line. If the line was present and dirty in L1D, the updated data is passed to L2 and merged with the L2 line that is being evicted. The combined result is written to external memory. If the victim line was not dirty in either L1D or L2, its contents are discarded. These actions ensure that the most recent writes to the affected cache line are written to external memory, but that clean lines are not needlessly written to external memory.

Note that L1P is not consulted when a line is evicted from L2. This allows program code to remain in L1P despite having been evicted from L2. This presumes that program code is never written to. In those rare situations where this is not the case, programs may use the cache controls described in section 6.3.2 to remove cached program code from L1P.

If the cache miss was a read, the data is stored in L2 Cache when it arrives. It is also forwarded directly to the requestor, thereby reducing the overall stall time. Because L1D and L1P cache lines are smaller than L2 cache lines, the L2 requests the missed portion first and the remaining portion last. The requested portion is allocated directly in the corresponding L1 cache, and the remaining portion is allocated in L2 only.

If the cache miss was a write, the incoming data is merged with the write data from L1D, and the merged result is stored in L2. In the case of a write, the line is not immediately allocated in L1D, as L1D does not write allocate.

A cache-miss to a non-cacheable location results in a long-distance access. A long distance read causes the L2 to issue a read transfer for the requested data to the EDMA controller. When the requested data is returned, the L2 forwards the data to the requestor. It does not allocate space for this data in the L2 Cache. A long distance read from L1D reads exactly the amount of data requested by the CPU. A long distance read from L1P reads exactly 32 bytes. The data read is not cached by L1D, L1P, or L2.

A long distance write causes the L2 to store a temporary copy of the written data. It then issues a write transfer for the write miss to the EDMA controller. Long distance writes can only originate from L1D via the L1D write buffer. Because the written data is stored in a special holding buffer, it is not necessary to stall the CPU while the long-distance write is being processed. Also, further writes to the L2 SRAM address space or on-chip peripherals may be processed while the long-distance access is being executed.

The L2 cache allows only one long distance access to be in progress at one time. A long distance write will potentially stall the CPU if a previous long distance write is still in progress. Likewise, a long distance read will not be processed until all previous long distance writes are complete.

The ordering constraints placed on long-distance accesses effectively make accesses to non-cacheable regions of memory strongly ordered. Program order for these accesses is retained, thus making these accesses useful for interfacing to peripherals or synchronizing with other CPUs.

5.3 L2 Bank Structure

The L2 memory is divided into four 64-bit banks that operate at the CPU’s clock rate, but pipelines accesses over two cycles. The L2 begins processing a new request on each cycle, although each bank can process new requests no faster than one every two cycles.

Because each bank requires two cycles to process a request, an access to a bank on one cycle blocks all accesses to that bank on the following cycle.

Thus, bank conflicts can occur between accesses on adjacent cycles. Repeated accesses to the same bank are processed at a rate of one every two cycles, regardless of the requestor.

The L2 must service requests from the following sources:

❑ L1P read miss that hits in L2.
❑ L1D read or write miss that hits in L2.
❑ EDMA reads and writes.
❑ Internal cache operations (victim writebacks, line fills, snoops)

The internal cache machinery is the fourth source of requests to the L2 memory. These requests represent data movement triggered by normal cache operation. Most of these requests are triggered by cache misses, or by user-initiated cache control operations.

Only one requestor may access the L2 in a given cycle. In the case of an L2 conflict, the L2 prioritizes requests in the above order. That is, L1P read hits have highest priority, followed by L1D, and so on.

L2 contention due to simultaneous accesses or bank conflicts can lengthen the time required for servicing L1P and L1D cache misses. The priority scheme above minimizes this effect by servicing CPU-initiated accesses before EDMA accesses and the background activities performed by the cache.

5.4 L2 Interfaces

The L2 services requests from the L1D cache, L1P cache, and the EDMA. The L2 provides access to its own memory, the peripheral configuration bus (CFGBUS), and the various cache control registers described in section 6, *Memory System Controls*. The following sections describe the interaction between L2 and the various requestors that it interfaces.

5.4.1 L1D/L1P-to-L2 Request Servicing

The L2 controller allows the L1P and L1D to access both L2 SRAM and L2 Cache. The L2 also acts as intermediary for long-distance accesses to addresses in external memory, and for accesses to on-chip peripherals. For each access, the address and the L2 mode determine the behavior.

Memory accesses to addresses that are mapped as L2 SRAM are serviced directly by L2 and are treated as cacheable by L1P and L1D. Memory accesses to addresses that are outside L2 SRAM but are on-chip are treated as non-cacheable. These accesses include accesses to on-chip peripherals and cache control registers.

L2 SRAM accesses by L1D do not trigger cache operations in either L1P or L2 Cache. Likewise, L1P accesses to L2 SRAM do not trigger cache operations in either L1D or L2 cache. This is consistent with the coherence policy outlined in section 7.1, *Memory System Coherence*. This contrasts with EDMA accesses to L2 SRAM, described in section 5.4.2.

L1P and L1D memory accesses to external addresses may be serviced by the L2 Cache, when the cache is enabled and caching is permitted on the given range of addresses. The cacheability of a given external address range is determined by the cache enable bit (CE) in the corresponding memory attribute register (MAR). Section 6.2, *Cacheability Controls*, describes MAR operation.

When the L2 is in all SRAM mode (L2MODE = 000b), it does not attempt to cache memory accesses directly. However, when servicing read requests for L1P and L1D, the L2 returns the cacheability status of the fetched data, thus allowing L1P and L1D to directly retain copies of memory that is marked as cacheable. Writes and writebacks from L1D for external addresses are sent to external memory directly when in this mode.

Accesses to noncacheable external locations are handled as long-distance accesses. Long-distance accesses are described in section 5.2, *L2 Operation*.

5.4.2 EDMA-to-L2 Request Servicing

The L2 controller handles EDMA access to L2 SRAM addresses. EDMA read and write requests to L2 SRAM result in specific actions to enforce the memory coherence policy described in section 7.1, *Memory System Coherence*.

The L2 controller tracks information on what portions of L2 SRAM might be held in L1D cache. When the EDMA reads from a location in L2 SRAM, the L2 controller *snoops* L1D for updated data, if necessary, before servicing the read for the EDMA. If L1D contains updated data, the update is written to L2 SRAM before allowing the EDMA read to proceed. The affected line in L1D is left valid but clean. No action is made relative to L1P.

When the EDMA writes to a location in L2 SRAM, the L2 controller sends an invalidate command to L1P and a snoop-invalidate command to L1D, as necessary. For L1P, the invalidate command is always sent since L2 does not track the state of L1P. If the address is present in L1P, the affected line is invalidated. As L1P cannot contain dirty data, L2 does not interact further with L1P. For L1D, the snoop-invalidate is sent only if L2 detects that L1D holds a copy of the address being written. If L1D contains updated data, the update is merged with the EDMA write and stored to L2, with the incoming data taking precedence over the data returned by L1D. The affected line in L1D is then marked invalid.

The end result of this system of snoops and invalidates is that coherence is retained between EDMA and CPU accesses to L2 SRAM. The example in section 7.1 illustrates this protocol in action.

5.4.3 L2 Request Servicing Using EDMA

The L2 controller relies on the EDMA to service requests on its behalf. Cache misses, victim writebacks, and long-distance accesses to external addresses cause the L2 controller to issue DMA transfer requests to the EDMA controller.

L2 requests are queued along other with other EDMA requests, and are serviced according to the priorities set by the EDMA. L2 requests are placed on the urgent EDMA priority queue. The priority and number of outstanding EDMA requests is fixed by the hardware. The EDMA is described in SPRU234, *TMS320C6000 DSP Enhanced Direct Memory Access (EDMA) Controller Peripheral Reference Guide*.

5.4.4 EDMA Access to Cache Controls

The L2 controller manages access to the memory-mapped cache control registers. While the L2 controller permits CPU access to these registers, it does not permit EDMA access to any of these registers. EDMA accesses to the cache control registers are dropped. EDMA writes are ignored and EDMA reads return undefined values.

As a result of these restrictions, cache control operations must only be initiated from the CPU. External devices that wish to trigger a cache control operation should do so using an interrupt handler or some other mechanism that triggers CPU writes to the appropriate registers on behalf of the requestor.

5.4.5 HPI and PCI Access to Memory Subsystem

The host port interface (HPI) and peripheral component interconnect (PCI) peripherals are not directly connected to the two-level memory subsystem. HPI and PCI requests are serviced indirectly by the EDMA. Therefore, HPI and PCI accesses follow the same coherence policies and are subject to the same restrictions as EDMA accesses.

Sections 5.4.2 through 5.4.4 describe the interaction between the EDMA and the two-level memory system. Section 7 describes the coherence and ordering policies for the memory system.

6 Memory System Controls

The two-level memory hierarchy is controlled by several memory-mapped control registers listed in Table 3. It is also controlled by the data cache control (DCC) and program cache control (PCC) fields in the CPU control and status register (CSR).

Note that in Table 3 the memory attribute registers (MAR) documented are renamed as compared to earlier versions of the documentation. The registers have been renamed to accommodate the larger set of MAR registers supported on the C64x devices. The addresses and function of MAR have not changed. The same MAR controls the cacheability of the same address range on both C621x/C671x and C64x devices. Table 4 lists the old names to the new names.

Table 3. Internal Memory Control Registers

Register Address	Acronym	Name
0184 0000h	CCFG	Cache configuration register
0184 4000h	L2WBAR	L2 writeback base address register
0184 4004h	L2WWC	L2 writeback word count register
0184 4010h	L2WIBAR	L2 writeback-invalidate base address register
0184 4014h	L2WIWC	L2 writeback-invalidate word count register
0184 4020h	L1PIBAR	L1P invalidate base address register
0184 4024h	L1PIWC	L1P invalidate word count register
0184 4030h	L1DWIBAR	L1D writeback-invalidate base address register
0184 4034h	L1DWIWC	L1D writeback-invalidate word count register
0184 5000h	L2WB	L2 writeback all register
0184 5004h	L2WBINV	L2 writeback-invalidate all register
0184 8000h to 0184 81FCh	MAR0 to MAR127	Reserved
0184 8200h	MAR128	Controls EMIF CE0 range 8000 0000h–80FF FFFFh
0184 8204h	MAR129	Controls EMIF CE0 range 8100 0000h–81FF FFFFh
0184 8208h	MAR130	Controls EMIF CE0 range 8200 0000h–82FF FFFFh
0184 820Ch	MAR131	Controls EMIF CE0 range 8300 0000h–83FF FFFFh

Table 3. Internal Memory Control Registers (Continued)

Register Address	Acronym	Name
0184 8210h to 0184 823Ch	MAR132 to MAR143	Reserved
0184 8240h	MAR144	Controls EMIF CE1 range 9000 0000h–90FF FFFFh
0184 8244h	MAR145	Controls EMIF CE1 range 9100 0000h–91FF FFFFh
0184 8248h	MAR146	Controls EMIF CE1 range 9200 0000h–92FF FFFFh
0184 824Ch	MAR147	Controls EMIF CE1 range 9300 0000h–93FF FFFFh
0184 8250h to 0184 827Ch	MAR148 to MAR159	Reserved
0184 8280h	MAR160	Controls EMIF CE2 range A000 0000h–A0FF FFFFh
0184 8284h	MAR161	Controls EMIF CE2 range A100 0000h–A1FF FFFFh
0184 8288h	MAR162	Controls EMIF CE2 range A200 0000h–A2FF FFFFh
0184 828Ch	MAR163	Controls EMIF CE2 range A300 0000h–A3FF FFFFh
0184 8290h to 0184 82BCh	MAR164 to MAR175	Reserved
0184 82C0h	MAR176	Controls EMIF CE3 range B000 0000h–B0FF FFFFh
0184 82C4h	MAR177	Controls EMIF CE3 range B100 0000h–B1FF FFFFh
0184 82C8h	MAR178	Controls EMIF CE3 range B200 0000h–B2FF FFFFh
0184 82CCh	MAR179	Controls EMIF CE3 range B300 0000h–B3FF FFFFh
0184 82D0h to 0184 83FCh	MAR180 to MAR255	Reserved

Table 4. Mapping of Old MAR Name to New MAR Name on C621x/C671x Device

Old MAR	New MAR	Register Address	Address Range Controlled
MAR0	MAR128	0184 8200h	CE0 range 8000 0000h–80FF FFFFh
MAR1	MAR129	0184 8204h	CE0 range 8100 0000h–81FF FFFFh
MAR2	MAR130	0184 8208h	CE0 range 8200 0000h–82FF FFFFh
MAR3	MAR131	0184 820Ch	CE0 range 8300 0000h–83FF FFFFh
MAR4	MAR144	0184 8240h	CE1 range 9000 0000h–90FF FFFFh
MAR5	MAR145	0184 8244h	CE1 range 9100 0000h–91FF FFFFh
MAR6	MAR146	0184 8248h	CE1 range 9200 0000h–92FF FFFFh
MAR7	MAR147	0184 824Ch	CE1 range 9300 0000h–93FF FFFFh
MAR8	MAR160	0184 8280h	CE2 range A000 0000h–A0FF FFFFh
MAR9	MAR161	0184 8284h	CE2 range A100 0000h–A1FF FFFFh
MAR10	MAR162	0184 8288h	CE2 range A200 0000h–A2FF FFFFh
MAR11	MAR163	0184 828Ch	CE2 range A300 0000h–A3FF FFFFh
MAR12	MAR176	0184 82C0h	CE3 range B000 0000h–B0FF FFFFh
MAR13	MAR177	0184 82C4h	CE3 range B100 0000h–B1FF FFFFh
MAR14	MAR178	0184 82C8h	CE3 range B200 0000h–B2FF FFFFh
MAR15	MAR179	0184 82CCh	CE3 range B300 0000h–B3FF FFFFh

6.1 Cache Mode Selection

The cache mode for the two-level memory hierarchy is determined by the cache configuration register (CCFG), and the data cache control (DCC) and program cache control (PCC) fields of the CPU control and status register (CSR). The CSR is shown in Figure 6.

Figure 6. CPU Control and Status Register (CSR)

31 24	23 16
CPU ID	Revision ID

15 10	9	8	7 5	4 2	1	0
PWRD	SAT	EN	PCC	DCC	PGIE	GIE

6.1.1 L1D Mode Selection Using DCC Field in CSR

The L1D only operates as a cache and cannot be memory mapped. The L1D does not support freeze or bypass modes. The only values allowed for the data cache control (DCC) field are 000b and 010b. All other values for DCC are reserved, as shown in Table 5.

Table 5. L1D Mode Setting Using DCC Field

Bit	Field	Value	Description
4–2	DCC		Data cache control bit.
		000	2-way cache enabled
		001	Reserved
		010	2-way cache enabled
		011	Reserved

6.1.2 L1P Mode Selection Using PCC Field in CSR

The L1P only operates as a cache and cannot be memory mapped. The L1P does not support freeze or bypass modes. The only values allowed for the program cache control (PCC) field are 000b and 010b. All other values for PCC are reserved, as shown in Table 6.

Table 6. L1P Mode Setting Using PCC Field

Bit	Field	Value	Description
7–5	PCC		Program cache control bit.
		000	Direct-mapped cache enabled
		001	Reserved
		010	Direct-mapped cache enabled
		011	Reserved

6.1.3 L2 Mode Selection Using L2MODE Field in CCFG

The L2 memory can function as mapped SRAM, as cache, or as some combination of both. The L2MODE field in the cache configuration register (CCFG) determines what portion of L2 is mapped as SRAM and what portion acts as cache. For some settings of L2MODE, either L2 SRAM or L2 Cache may not be present. The mode with no L2 Cache is referred to as all SRAM mode. The CCFG is shown in Figure 7 and described in Table 7.

The reset value of the L2MODE field is 000b, so the L2 is configured in all SRAM mode after reset. That is, there is no L2 Cache enabled and the entire L2 SRAM address range is available. The L2 still services external memory requests for the L1P and L1D in this mode. It will not, however, retain copies of the data itself.

By default, external memory is not marked as cacheable. Cacheability is controlled separately of the cache mode. Section 6.2 discusses the memory attribute registers (MAR) that control the cacheability of external memory.

As various amounts of L2 Cache are enabled, the amount of L2 SRAM available decreases. Addresses at the top of the L2 SRAM address space are converted to cache. For example, consider the C6211 DSP, which offers 64K bytes of L2 memory at addresses from 0000 0000h–0000 FFFFh. When the L2 is in 16K cache mode (L2MODE = 001b), the uppermost 16K-byte range of addresses, 0000 C000h–0000 FFFFh, is not available to programs.

Note:

Reads or writes to L2 address ranges that are configured as L2 Cache may result in undesired operation of the cache hierarchy. Programs must confine L2 accesses to L2 addresses that are mapped as L2 SRAM to ensure correct program operation.

To ensure correct operation when switching L2 cache modes, you must perform a series of operations. Table 8 specifies the required operations when either adding or removing L2 memory as mapped SRAM. Failure to follow these guidelines may result in data loss and undefined L2 operation.

Figure 7. Cache Configuration Register (CCFG)

31–16
Reserved
R-0

15–10	9	8	7–3	2–0
Reserved	IP	ID	Reserved	L2MODE
R-0	W-0	W-0	R-0	R/W-0

Legend: R = Read only; W = Write only; R/W = Read/write; -*n* = value after reset

Table 7. Cache Configuration Register (CCFG) Field Descriptions

Bit	Field	Value	Description		
31–10	Reserved	0	Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.		
9	IP		Invalidate L1P bit.		
		0	Normal L1P operation.		
		1	All L1P lines are invalidated.		
8	ID		Invalidate L1D bit.		
		0	Normal L1D operation.		
		1	All L1D lines are invalidated.		
7–3	Reserved	0	Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.		
2–0	L2MODE		L2 operation mode bits. Figure 8 shows how the various L2 cache modes affect the memory map for the C6211, C6711, and C6712 devices. Figure 9 shows the same information for the C6713 device.		
				C6211/C6711/C6712	**C6713**
		000	L2 cache disabled (All SRAM mode)	64K SRAM	256K SRAM
		001	1-way cache (16K L2 cache)	48K SRAM	240K SRAM
		010	2-way cache (32K L2 cache)	32K SRAM	224K SRAM
		011	3-way cache (48K L2 cache)	16K SRAM	208K SRAM
		100–110	Reserved		
		111	4-way cache (64K L2 cache)	0K SRAM	192K SRAM

Figure 8. C6211/C6711/C6712 L2 Cache Modes

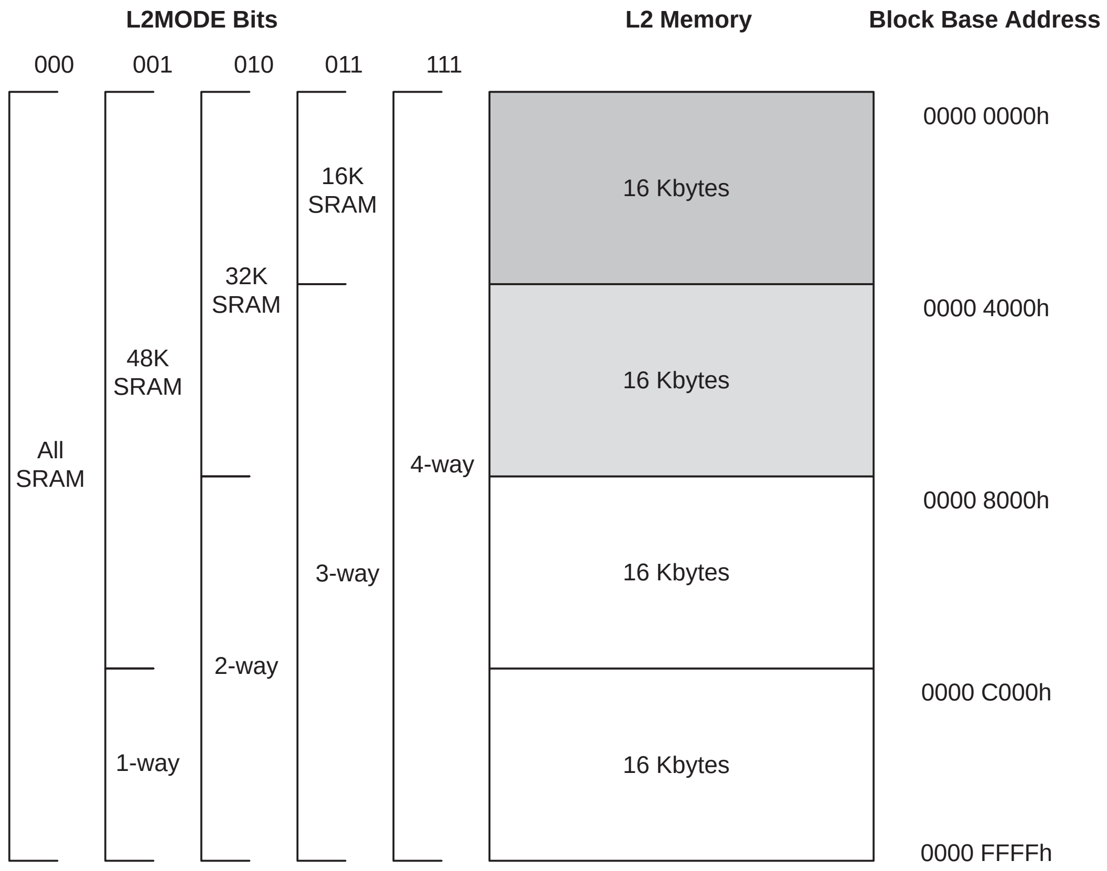


Figure 9. C6713 L2 Cache Modes

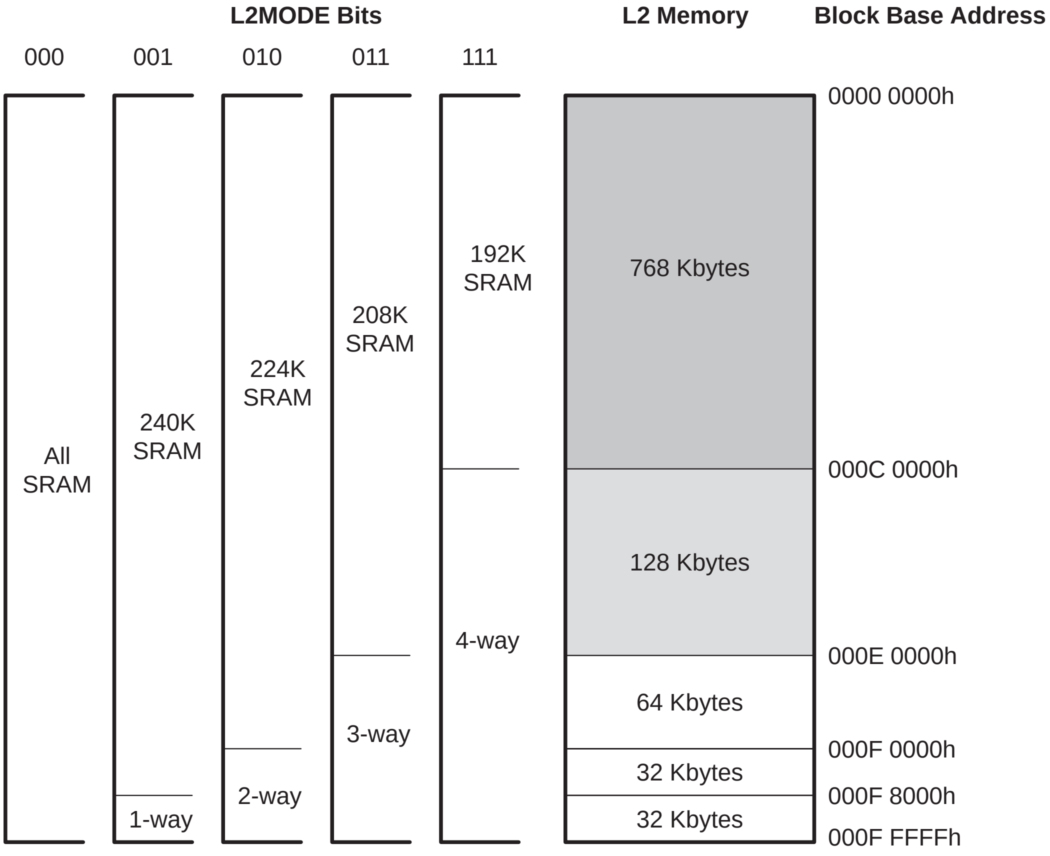


Table 8. L2 Mode Switch Procedure

To Switch From...	To...	Perform the Following Steps
All SRAM Mode	Mode with L2 Cache	1) Use EDMA to transfer any data needed out of the L2 SRAM space to be converted into cache. 2) Perform a block writeback-invalidate in L1D of L2 SRAM addresses that are about to become L2 Cache. 3) Wait for block writeback-invalidate to complete. 4) Perform block writeback-invalidates for any external address ranges that may be cached in L1D. (This step is unnecessary if no CE bit has been set to 1 in any MAR.) 5) Wait for the block writeback-invalidates from step 4 to complete. 6) Write to CCFG to change mode. 7) Force CPU to wait for CCFG modification by reading CCFG. 8) Execute 8 cycles of NOP.
Mode with mixed L2 SRAM and L2 Cache	Mode with less L2 mapped SRAM	1) Use EDMA to transfer any data needed out of the L2 SRAM space to be converted into cache. 2) Perform a block writeback-invalidate in L1D of L2 SRAM addresses that are about to become L2 Cache. 3) Wait for block writeback-invalidate to complete. 4) Perform global writeback-invalidate of L2 (L2WBINV). 5) Wait for L2WBINV to complete. 6) Write to CCFG to change mode. 7) Force CPU to wait for CCFG modification by reading CCFG. 8) Execute 8 cycles of NOP.
Any L2 mode	Mode with more L2 mapped SRAM	1) Perform global writeback-invalidate of L2 (L2WBINV). 2) Wait for L2WBINV to complete. 3) Write to CCFG to change mode. 4) Force CPU to wait for CCFG modification by reading CCFG. 5) Execute 8 cycles of NOP.

6.2 Cacheability Controls

The cacheability of external address ranges is controlled by the memory attribute registers (MAR). Each MAR controls the cacheability of a 16-Mbyte address range. MAR is shown in Figure 10 and described in Table 9. A listing of all MAR is in Table 4.

Figure 10. L2 Memory Attribute Register (MAR)

31 1	0
Reserved	CE
R-0	R/W-0

Legend: R = Read only; R/W = Read/write; -*n* = value after reset

Table 9. Memory Attribute Register Field Descriptions

Bit	Field	Value	Description
31–1	Reserved	0	Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.
0	CE		Cache enable bit.
		0	Memory range is not cacheable.
		1	Memory range is cacheable.

The cache enable (CE) bit in each MAR determines whether the L1D, L1P, and L2 are allowed to cache the corresponding address range. After reset, the CE bit in each MAR is cleared to 0, thereby disabling caching of external memory by default. This is in contrast to L2 SRAM, which is always considered cacheable.

To enable caching on a particular external address range, an application should set the CE bit in the appropriate MAR to 1. No special procedure is necessary. Subsequent accesses to the affected address range are cached by the two-level memory system.

To disable caching for a given address range, programs should follow the following sequence to ensure that all future accesses to the particular address range are not cached.

1) Ensure that all addresses within the affected range are removed from the L1 and L2 caches. This is accomplished in one of the following ways. Any one of the following operations should be sufficient.

   a) If L2 Cache is enabled, invoke a global writeback-invalidate using L2WBINV. Wait for the C bit in L2WBINV to read as 0. Alternately, invoke a block writeback-invalidate of the affected range using L2WIBAR/L2WIWC. Wait for L2WIWC to read as 0.

   b) If L2 is in all SRAM mode, invoke a block writeback-invalidate of the affected range using L1DWIBAR/L1DWIWC. Wait for L1DWIWC to read as 0.

   Note that the block-oriented cache controls can only operate on a 256-Kbyte address range at a time, so multiple block writeback-invalidate operations may be necessary to remove the entire affected address range from the cache. These cache controls are discussed in section 6.3.

2) Clear the CE bit in the appropriate MAR to 0.

6.3 Program-Initiated Cache Operations

The memory system provides a set of cache control operations. These allow programs to specifically request that certain data be written back or invalidated. The cache operations fall into two categories, global operations that operate on the entire cache, and block operations that operate on a specific range of addresses. The global operations are discussed in section 6.3.1 and the block operations are discussed in section 6.3.2.

The memory system can only perform one program-initiated cache operation at a time. This includes global operations, block operations, and mode changes. For this reason, the memory system may stall accesses to cache control registers while a cache control operation is in progress. Table 10 gives a summary of the available operations and their impact on the memory system.

Table 10. Summary of Program-Initiated Cache Operations

Type of Operation	Cache Operation	Register Usage	L1P Cache Effect	L1D Cache Effect	L2 Cache Effect
Global Operation	L2 Writeback All	L2WB	No effect.	Updated lines holding addresses also held in L2 Cache are written back. All lines corresponding to addresses held in L2 Cache are invalidated.[†]	Updated data is written back. All lines kept valid.
	L2 Writeback and Invalidate All	L2WBINV	Entire contents are discarded.[†]	Updated lines holding addresses also held in L2 Cache are written back. All lines corresponding to addresses held in L2 Cache are invalidated.[†]	Updated lines are written back. All lines are invalidated.
	L1P Invalidate All	IP bit in CCFG	Entire contents are discarded.	No effect.	No effect.
	L1D Invalidate All	ID bit in CCFG	No effect.	Entire contents are discarded. No updated data is written back.	No effect.
Block Operation	L2 Block Writeback	L2WBAR, L2WWC	No effect	Updated lines in block are written to L2. All lines in block are invalidated in L1D.[†]	Updated lines in block are written to external memory. Lines in block are kept valid in L2.
	L2 Block Writeback with Invalidate	L2WIBAR, L2WIWC	All lines in block are invalidated.[†]	Updated lines in block are written to L2. All lines in block are invalidated in L1D.[†]	Updated lines in block are written to external memory. All lines in block are invalidated in L2.
	L1P Block Invalidate	L1PIBAR, L1PIWC	All lines in block are invalidated.	No effect.	No effect.
	L1D Block Writeback with Invalidate	L1DWIBAR, L1DWIWC	All lines in block are invalidated.[‡]	Updated lines in block are written to L2. All lines in block are invalidated in L1D.	No effect.

[†] As described in section 6.3.3, these operations only operate on L1D and L1P when L2 Cache is enabled. When L2 Cache is enabled, these operations only affect those portions of L1P and L1D that are also held in L2 Cache.

[‡] In contrast to its behavior on C64x devices, L1DWIBAR/L1DWIWC cause the corresponding block to be invalidated in L1P.

6.3.1 Global Cache Operations

The global cache operations execute on the entire contents of a particular cache. Global operations take precedence over program accesses to the cache. Program accesses (either data or program fetches) to a particular cache are stalled while a global cache operation is active on that cache.

Global operations in L1D and L1P are initiated by the ID and IP bits in CCFG (Figure 7). The L1D and L1P only offer global invalidation. By writing a 1 to the ID or IP bit in CCFG, a program can invalidate the entire contents of the corresponding L1 cache. Upon initiation of the global invalidate, the entire contents of the corresponding cache is discarded — no updated data is written back. The command bit continues to read as 1 until the operation is complete.

> CAUTION
>
> **The L1D global-invalidate causes all updated data in L1D to be discarded, rather than written back to the lower levels of memory. This can cause incorrect operation in programs that expect the updates to be written to the lower levels of memory. Therefore, most programs use either the L1D block writeback-invalidate (described in section 6.3.2) or the global L2 operations, rather than the L1D global-invalidate.**

The L2 offers both global writeback and global writeback-invalidate operations. Global cache operations in L2 are initiated by writing a 1 to the C bit in either L2WB (Figure 11) or L2WBINV (Figure 12). Writing a 1 to the C bit of L2WB initiates a global writeback of L2; writing a 1 to the C bit of L2WBINV initiates a global writeback-invalidate of L2. The C bit continues to read as 1 until the cache operation is complete. Programs can poll to determine when a cache operation is complete.

Global operations on L2 have indirect effects on the L1 caches, as discussed in section 6.3.3.

Figure 11. L2 Writeback All Register (L2WB)

31	1	0
Reserved		C
R-0		R/W-0

Legend: R = Read only; R/W = Read/write; -*n* = value after reset

Figure 12. L2 Writeback-Invalidate All Register (L2WBINV)

31	1	0
Reserved		C
R-0		R/W-0

Legend: R = Read only; R/W = Read/write; -*n* = value after reset

6.3.2 Block Cache Operations

The block cache operations execute on a range of addresses that may be held in the cache. Block operations execute in the background, allowing other program accesses to interleave with the block cache operation.

Programs initiate block cache operations with two writes. The program first writes the starting address to one of the base address registers (BAR), shown in Figure 13. Next, the program writes the total number of words to operate on to the corresponding word count register (WC), shown in Figure 14. The cache operation begins as soon as the word count register is written with a non-zero value. The cache provides a set of BAR/WC pseudo-register pairs, one for each block operation the cache supports. The complete list of supported operations is shown in Table 10.

Notice that the word count field in WC is only 16-bits wide. This limits the block size to 65535 words (approximately 256K bytes). Larger ranges require multiple block commands to be issued to cover the entire range.

Although block operations specify the block in terms of a word address and word count, the block operations always operate on whole cache lines. Whole lines are always written back and/or invalidated in each affected cache. For this reason, programs should be careful to align arrays on cache-line boundaries, and to pad arrays to be a multiple of the cache line size. This is especially true when invoking the invalidate-only commands with respect to these arrays.

Figure 13. Block Cache Operation Base Address Register (BAR)

31 0
Base Address
R/W-0

Legend: R/W = Read/write; -*n* = value after reset

Figure 14. Block Cache Operation Word Count Register (WC)

31 16	15 0
Reserved	Word Count
R-0	R/W-0

Legend: R = Read only; R/W = Read/write; -*n* = value after reset

In current device implementations, the block command registers (xxBAR and xxWC) are implemented as a single register pair in the underlying hardware. They appear as pseudo-registers at multiple addresses in the address map. Therefore, programs ***must not*** interleave writes to different command register pairs. Writes to one xxBAR/xxWC pair should be performed atomically with respect to other xxBAR/xxWC writes. Depending on the nature of the program, it may be necessary to disable interrupts while writing to xxBAR and xxWC.

Despite this restriction, programs may attempt to initiate a new block command while a previous command is in progress. The cache controller will stall the write to xxBAR until the previous cache operation completes, thus preserving correct operation. To avoid stalling, programs may determine if a block cache operation has completed by polling WC. WC returns a non-zero value while the cache operation is in progress, and zero when the operation has completed. The non-zero value returned may vary between device implementations, so programs should only rely on the register being non-zero.

Programs should not assume that the value of xxBAR is retained between block cache operations. Programs should always write an address to xxBAR and a word count to xxWC for each cache operation. Also, programs should not assume that the various xxBAR and xxWC map to the same physical register. For each cache operation, programs should write to xxBAR and xxWC for that operation.

As indicated in Table 10, block writeback-invalidates in L1D also invalidate the corresponding range of L1P. The parallel invalidate in L1P is intended to simplify support for program loaders and other applications that write to addresses that will be subsequently executed from. Typically, data and program code address ranges are separate, so programs that do not write to addresses that will be executed from will see no effects due to this additional invalidation. The specific timing of L1P invalidates relative to L1D invalidates and program fetches is not well defined. Therefore, programs should wait for L1DWIWC to read as zero prior to branching to an address range that has been invalidated indirectly in this manner. (Note that the behavior of L1DIBAR/ L1DIWC differs on C64x devices, see the C64x documentation for details.)

Block cache operations in L2 can indirectly affect L1D and L1P, as noted in Table 10. Section 6.3.3 discusses these interactions in detail.

Note:

Reads or writes to the addresses within the block being operated on while a block cache operation is in progress may cause those addresses to not be written back or invalidated as requested. To avoid this, programs should not access addresses within the range of cache lines affected by a block cache operation while the operation is in progress. Programs may consult the appropriate xxWC to determine when the block operation is complete.

6.3.3 Effect of L2 Commands on L1 Caches

Cache operations in L2 indirectly operate on the L1D and L1P cache. As a result, the L2 cache operations have *no effect on any of the caches* when L2 is in all SRAM mode. This is true for both the global and block commands. Otherwise, when L2 Cache is enabled, program-initiated cache operations in L2 may operate on the corresponding contents of the L1D and L1P.

Under normal circumstances, the L1D cache is *inclusive* in L2, and L1P is not. Inclusive implies that the entire contents of L1D are also held either in L2 SRAM or L2 Cache. The L2 cache operations are designed with these properties in mind.

Because L1P is not inclusive in L2, the L2 cache operations that invalidate lines in L2 send explicit invalidation commands to L1P. A global writeback-invalidate of L2 (L2WBINV) triggers a complete invalidation of L1P. Block invalidate and writeback-invalidate operations in L2 blindly send invalidate commands to L1P for the corresponding L1P cache lines. This ensures that L1P always fetches the most recent contents of memory after the cache operation is complete.

Because L1D is normally inclusive in L2, the L2 relies on normal cache protocol operation to operate on L1D indirectly. Writebacks and invalidates triggered for lines in L2 result in snoop-invalidate commands sent to L1D when L2 detects that L1D is also holding a copy of the corresponding addresses. Therefore, the L2 global writeback (L2WB) and L2 global writeback-invalidate (L2WBINV) causes all external addresses held in L1D to be written back and invalidated from L1D. Likewise, block operations in L2 cause the corresponding range in L1D to be written to L2 and invalidated from L1D using the indirect snoop-invalidates.

One result of this is that L2 SRAM addresses cached in L1D are not affected by program-initiated cache operations in L2, as L2 Cache never holds copies of L2 SRAM. To remove L2 SRAM addresses from L1D, programs must use the L1D block cache operations directly. Ordinarily, direct removal of L2 SRAM addresses from L1D is required only when changing L2 Cache modes. The coherence policy described in section 7.1, *Memory System Coherence*, makes unnecessary most of the need for programs to manually write back portions of L1D to L2 SRAM.

Another result is nonintuitive behavior when L1D is *not* inclusive in L2. L1D is inclusive in L2 under normal circumstances, and so most programs do not need to be concerned about this situation. Indeed, the recommended L2 cache mode-change procedure in section 6.1.3 ensures that the memory system is never in this state. When *not* following the procedure precisely, it is possible for L1D to hold copies of external memory that are not held in L2. This noninclusive state is achieved in the following rare sequence:

1) The program enabled caching for an external address range while L2 was in all SRAM mode.

2) The program read directly from this external address range.

3) The program then enabled L2 Cache without first removing the external address range from L1D with a block cache operation.

To prevent this situation, programs should ensure no external addresses are cached in L1D when enabling L2 Cache. This is accomplished by not enabling caching for external addresses prior to enabling L2 Cache, or by removing external addresses from L1D using the block cache operations described in section 6.3.2.

7 Memory System Policies

This section discusses the various policies of the memory system, such as coherence between CPU and EDMA or host accesses, and the order in which memory updates are made.

7.1 Memory System Coherence

Cache memories work by retaining copies of data from lower levels of the memory hierarchy in the hierarchy's higher levels. This provides a performance benefit as higher levels of the memory hierarchy may be accessed more quickly than the lower levels, and so the lower levels need not be consulted on every access. Because many accesses to memory are captured at higher levels of the hierarchy, the opportunity exists for the CPU and other devices to see a different picture of what is in memory.

A memory system is coherent if all requestors into that memory see updates to individual memory locations occur in the same order. A requestor is a device such as a CPU or EDMA. A coherent memory system assures that all writes to a given memory location are visible to future reads of that location from any requestor, so long as no intervening write to that location overwrites the value. If the same requestor is writing and reading, the results of a write are immediately visible. If one requestor writes and a different requestor reads, the reader may not see the updated value immediately, but it will be able to see updates after a sufficient period of time. Coherence also implies that all writes to a given memory location appear to be serialized. That is, that all requestors see the same order of writes to a memory location, even when multiple requestors are writing to one location.

The two-level cache memory system is coherent with various qualifications. The C621x/C671x devices provide a set of coherence assurances to the programmer that are designed to simplify and improve the performance of the hardware, while still providing a reasonable programming model.

Memory system coherence only applies to cacheable data. For noncacheable data, the memory system merely passes the access on to its final destination without retaining any intermediate copies, and thus coherence is not an issue. Therefore, this discussion focuses only on cacheable memory, which is generally confined to various forms of RAM and ROM.

The coherence model of the C621x/C671x memory system is expressed in terms of requestors and where the memory is located. The two-level memory system must support requests from three sources: CPU program fetches, CPU data accesses, and EDMA accesses. (HPI and PCI accesses to the two-level memory system are handled using the EDMA.) Cacheable areas of interest include both internal and external RAM. Since ROM is typically not writable, we consider only RAM. Table 11 shows where the memory system assures coherency.

Table 11. Coherence Assurances in the Two-Level Memory System

Requestor	CPU Program Fetch	CPU Data Access	EDMA Access
CPU program fetch	Coherent	Software-managed, L2 level only	On-chip SRAM only
CPU data access	Software-managed, L2 level only	Coherent	On-chip SRAM only
EDMA access	On-chip SRAM only	On-chip SRAM only	Coherent

Notice that the hardware ensures that accesses by the CPU and EDMA to internal SRAM addresses are coherent, but external addresses are not. Software must ensure external addresses accessed by the EDMA are not held in cache when the EDMA accesses them. Failure to do so can result in data corruption on the affected range of addresses. See section 6.3, *Program-Initiated Cache Operations*, for the steps required to ensure particular ranges of addresses are not held in cache.

Also notice that CPU program and data accesses are not coherent. The reason that these are not considered coherent is that the L1P does not query the L1D when it makes a program fetch, and thus CPU writes captured in L1D may not be visible to L1P for an arbitrarily long period of time. Therefore, programs that write to locations that are subsequently executed from must ensure that the updated data is written from L1D to at least L2 before execution. The L1DWIBAR/L1DWIWC and L1PIBAR/L1PIWC described in section 6.3 can be used for this purpose.

Although the memory system is coherent, it does not assure that all requestors see updates to cacheable memory occurring in the same order. This is of primary importance when a given buffer of memory is accessed by the CPU and an EDMA or host port access. Unless care is taken, it is possible for an EDMA or host port access to see a mixture of old and new data if the access occurs while the CPU updates are being processed by the memory system. Section 7.3 discusses the order in which CPU accesses are made visible to the memory system.

7.2 EDMA Coherence in L2 SRAM Example

As indicated in section 7.1, the memory system provides coherence assurance on CPU and EDMA accesses to L2 SRAM. It does not provide coherence between CPU and EDMA accesses to external memory. This section illustrates how the L2 SRAM coherence assurances are enforced by the hardware within the context of an example.

A common DSP programming technique is to process streams of data with a double-buffering scheme. In this setup, a large set of data external to the device is processed by transferring in small chunks and operating on them. The results are then transferred to external memory. These transfers are generally executed by the EDMA controller. In pseudo-code, a typical processing sequence might be as shown in Figure 15.

Because the EDMA operates independently of the CPU, it is possible to overlap the EDMA transfers with data processing. Double-buffering allows this by having two sets of input and output buffers: one for processing and one for transfers-in-progress. The CPU and EDMA alternate sets of buffers, “ping-ponging” between them. (For this reason, double-buffering is sometimes referred to as ping-pong buffering.) In pseudo-code, a double-buffered processing sequence might be as shown in Figure 16. Here, PING and PONG are pairs of pointers to the double-buffers. PING points to the buffers that the CPU will process, and PONG points to the buffers the EDMA is filling.

Graphically, the time sequence for the internal input and output buffers would look as shown in Figure 17. EDMA reads are one step ahead of the processing and EDMA writes are one step behind. In Figure 17, step 2 operates in parallel with steps 3 and 4; steps 5 and 6 overlap with steps 7 and 8.

Figure 18 shows the read/process/write pipeline in software, with the EDMA transfers overlapping processing.

Figure 15. Streaming Data Pseudo-Code

```
while (data left to process)
{
   Read next block to internal input buffer


   Process internal input buffer, placing result in internal output buffer


   Write the internal output buffer to the external output
}
```

Figure 16. Double Buffering Pseudo-Code

```
Read first block to internal PING input buffer.
while (data left to process)
{
   Start reading next block to internal “PONG” input buffer.

   Wait for PING input and output buffers to be ready.

   Process internal PING input buffer, placing result in internal
   PING output buffer.

   Swap PING and PONG buffer pointers.

   Start writing out PONG input buffer.
}
```

Figure 17. Double-Buffering Time Sequence

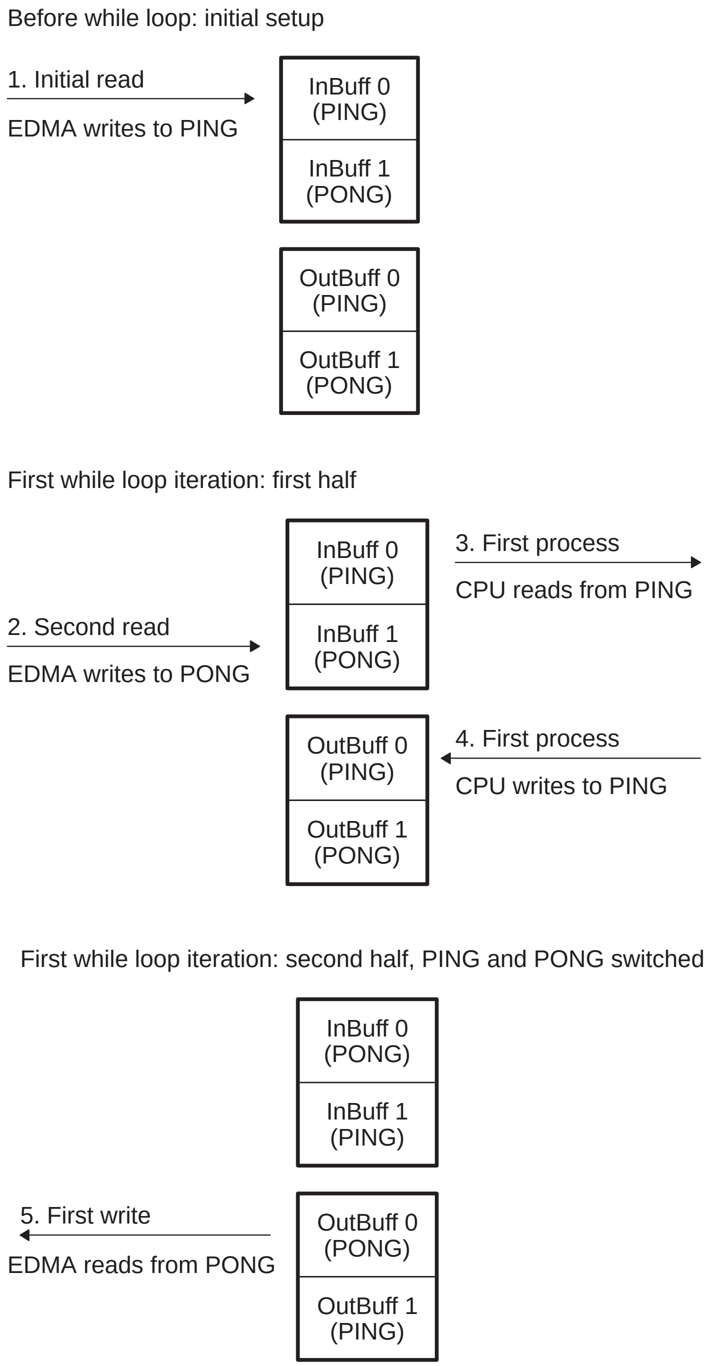
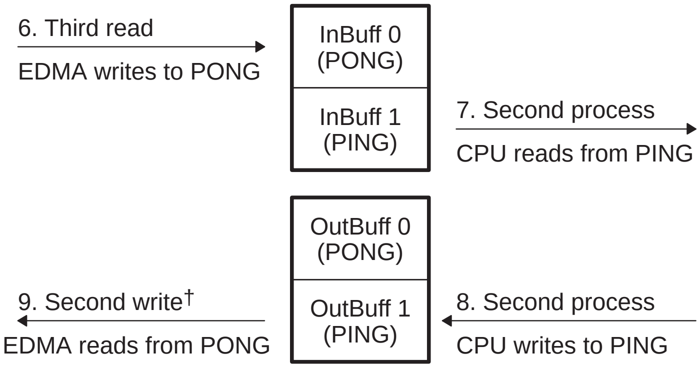


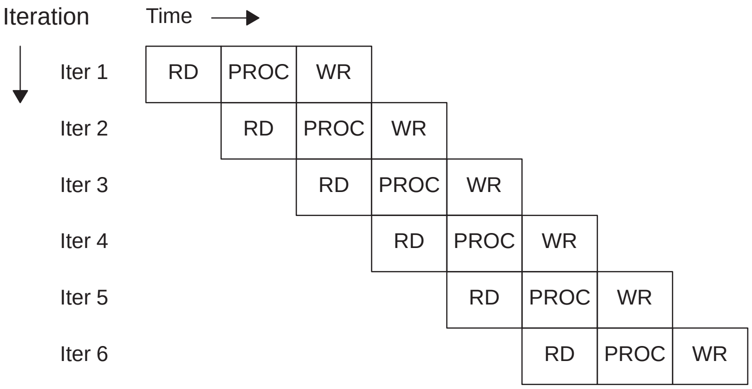
Figure 17. Double-Buffering Time Sequence (Continued)

Second while loop iteration



† PING and PONG labels switch before step 9.

Figure 18. Double Buffering as a Pipelined Process



Iteration	Time →							
Iter 1	RD	PROC	WR					
Iter 2		RD	PROC	WR				
Iter 3			RD	PROC	WR			
Iter 4				RD	PROC	WR		
Iter 5					RD	PROC	WR	
Iter 6						RD	PROC	WR

The memory system's coherence directly supports this programming technique. A system of snoops and invalidates (as discussed in section 5.4.2, *EDMA-to-L2 Request Servicing*) keeps L2 SRAM and L1D in sync without programmer intervention. In the context of this example, the memory system performs the following operations in response to each of the numbered steps in Figure 17.

1) The EDMA writes to InBuff 0. As the EDMA writes to InBuff 0, the L2 controller sends snoop-invalidates for the corresponding lines in L1D, if it detects that L1D may be caching these addresses. Dirty data from L1D is merged with the data written by the EDMA, so that L2 contains the most up-to-date data. The corresponding lines are invalidated in L1D.

2) This step is similar to step 1, EDMA writes to InBuff 1.

3) The CPU reads InBuff 0. These lines were snoop-invalidated from L1D in step 1. Therefore, these accesses miss L1D, forcing L1D to read the new data from L2 SRAM.

4) The CPU writes to OutBuff 0. (In practice, this step may overlap or interleave with step 3.) Suppose initially OutBuff 0 is present in L1D. In this case, the writes hit L1D and mark the lines as dirty. If OutBuff 0 were not present in L1D, the writes would go directly to L2 SRAM.

5) The EDMA reads OutBuff 0. The EDMA reads in L2 trigger a snoop for each cache line held in L1D. Any dirty data for the line is written to L2 before the EDMA's read is processed. Thus, the EDMA sees the most up-to-date data. The line is subsequently invalidated in L1D. (On C64x devices, the line is marked clean and is left valid.)

6) The EDMA writes to InBuff 0. This step proceeds identically to step 1. That is, InBuff 0 is snoop-invalidated from L1D as needed. The EDMA writes are processed after any dirty data is written back to L2 SRAM.

7) The CPU reads InBuff 1. These lines were snoop-invalidated in step 6. Therefore, these accesses miss L1D, forcing L1D to read the new data from L2 SRAM.

8) The CPU writes to OutBuff 1. This step proceeds identically to step 4.

9) The EDMA reads OutBuff 1. This step proceeds identically to step 5.

Notice that the system of snoops and snoop-invalidates automatically keeps the EDMA and CPU synchronized for the input and output buffers. Further, double-buffering allows EDMA and CPU accesses to occur in parallel.

7.3 Memory Access Ordering

7.3.1 Program Order of Memory Accesses

To optimize throughput, power, and ease of programming, the C6000 DSP architecture supports a combination of strongly ordered and relaxed ordered memory models. These terms are defined relative to the order of memory operations implied by a particular program sequence. This ordering is referred to as the program order of memory accesses.

The C6000 DSP cores may initiate up to two parallel memory operations per cycle. The program order of memory accesses defines the outcome of memory accesses in terms of a hypothetical serial implementation of the architecture. That is, it describes the order that the parallel memory operations are processed such that time-sequence terms such as earlier and later are used precisely with respect to a particular sequence of operations.

Program order is defined with respect to instructions within an execute packet and with respect to a sequence of execute packets. Memory operations (including those that are issued in parallel) are described as being earlier or later with respect to each other. The terms earlier and later are strictly opposites: if X is not earlier than Y, then X is later than Y.

Memory accesses initiated from different execute packets have the same temporal ordering as the execute packets themselves. That is, in the defined program order, memory operations issued on cycle *i* are always earlier than memory accesses issued on cycle *i* + 1, and are always later than those issued on cycle *i* – 1.

For accesses issued in parallel, the type of operations (reads or writes), and the data address ports that execute the operations determine the ordering. Table 12 describes the ordering rules.

Table 12. Program Order for Memory Operations Issued From a Single Execute Packet

Data Address 1 (DA1) Operation	Data Address 2 (DA2) Operation	Program Order of Accesses
Load	Load	DA1 is **earlier** than DA2
Load	Store	DA1 is **earlier** than DA2
Store	Load	DA2 is **earlier** than DA1
Store	Store	DA1 is **earlier** than DA2

The data address port for a load or store instruction is determined by the datapath that provides the data (as opposed to the address) for the memory operation. Load and store instructions that operate on data in the A datapath use DA1. Load and store instructions that operate on data in the B datapath use DA2. Note that the datapath that provides the data to be operated on determines whether DA1 or DA2 is used. The datapath that provides the address for the access is irrelevant.

(The C64x DSP supports nonaligned memory accesses to memory using the LDNW, STNW, LDNDW, and STNDW instructions. The memory system does not assure that these memory accesses will be atomic. Rather, it may divide the accesses for these instructions into multiple operations. The program order of memory accesses does not define the order of the individual memory operations that comprise a single nonaligned access. The program order only defines how the entire nonaligned access is ordered relative to earlier and later accesses. So, although the complete nonaligned access does follow the program order defined above with respect to the CPU itself, other requestors may see the nonaligned memory access occur in pieces.)

The above definition describes the memory system semantics. The memory system assures that the semantics of the program order of memory accesses will be retained for CPU accesses relative to themselves. The memory system may, however, relax the ordering of operations as they are executed within the memory hierarchy so long as the correct semantics are retained. It may also allow other requestors to the memory system to see the accesses occur in an order other than the original program order. Section 7.3.2 describes this in detail.

7.3.2 Strong and Relaxed Memory Ordering

The program order of memory accesses (as described in section 7.3.1) describes the desired semantics of a particular sequence of memory accesses. In most circumstances, it is not necessary for the memory system to execute the memory accesses in this exact order at all levels of memory hierarchy in order to retain these semantics. Indeed, for performance and power reasons, it is extremely advantageous for the memory hierarchy to relax the order of accesses.

When communicating with peripherals and when coordinating with other devices (the EDMA, other CPUs) that are accessing the same memory, retaining program order potentially becomes more important. Given these diverse needs, the memory hierarchy supports both strongly ordered and relaxed orderings for memory operations.

Memory system coherence implies that writes to a single memory location are serialized for all requestors, and that all requestors see the same sequence of writes to that location. Coherence does not make any implications about the ordering of accesses to different locations, or the ordering of reads with respect to other reads of the same location. Rather, the memory system ordering rules (strong or relaxed) describe the ordering assurances applied to accesses to different locations.

A sequence of memory operations are said to be strongly ordered if it is not possible to observe a different sequence of memory operations from some place in the system. For instance, if one requestor writes to location X earlier than it writes to a different location Y, a second requestor must not see Y updated before it sees X updated for the accesses to be considered strongly ordered. In this example, regardless of the order in which the memory system processes writes from the first requestor and reads from the second requestor, the second requestor must not be able to observe the write to Y occurring before the write to X. Strong ordering does not require the writes to be serialized, however. The writes may occur in parallel or even in opposite order so long as it is not possible for the second CPU to observe Y being updated before X.

(Nonaligned accesses on C64x devices (those issued by the LDNW, STNW, LDNDW, and STNDW instructions) may be strongly ordered relative to other accesses. Because nonaligned accesses are not assured to be atomic, strong ordering assurances apply only to the nonaligned access as a whole. No ordering assurances are made between the individual operations that comprise a single nonaligned access, should the memory system divide the nonaligned access into multiple memory operations.)

The memory hierarchy provides strong ordering for all noncacheable long-distance accesses. Such accesses are typically to peripherals and external memory for which the corresponding MAR cache enable bit is not set.

For cacheable locations in external memory, the memory hierarchy provides only a relaxed ordering. Thus, it is possible for other requestors to observe updates occurring in memory in a different order than the original program order of memory accesses. Despite this relaxed ordering, the CPU still sees the desired memory system semantics as described in section 7.3.1.

For cacheable locations in L2 SRAM, the C621x/C671x devices provide strong ordering for CPU accesses to locations that are within the same L1D cache line, and for locations that are not present in L1D. On the C621x/C671x devices, this means that data accesses to addresses whose upper 27 bits are equal are strongly ordered. (On the C64x device, this means that data accesses to addresses whose upper 26 bits are equal are strongly ordered.)

For locations in L2 SRAM that are not within the same cache line, strong ordering is provided only on writes and only as long as addresses involved are not present in L1D. This can be ensured by using the L1DWIBAR and L1DWIWC control registers described in section 6.3.2, *Block Cache Operations*. In all other cases, a relaxed ordering is provided for CPU accesses to L2 SRAM.

The L2 provides limited ordering assurances for EDMA access to L2 SRAM. EDMA reads to L2 are not reordered relative to other EDMA reads. EDMA writes to L2 are not reordered relative to other EDMA writes. Reads and writes, however, may be reordered relative to each other.

Index