Engineer-to-Engineer Note EE-271

ANALOG DEVICES

Technical notes on using Analog Devices DSPs, processors and development tools
Visit our Web resources http://www.analog.com/ee-notes and http://www.analog.com/processors or e-mail processor.support@analog.com or processor.tools.support@analog.com for technical support.

Using Cache Memory on Blackfin® Processors

Contributed by Kunal Singh & Andreas Pellkofer

Rev 2 – May 13, 2009

Introduction

This application note discusses cache memory management for Analog Devices Blackfin® processor family. The document introduces popular cache schemes and then discusses the Blackfin instruction cache and the data cache in detail.

The described features are available on all Blackfin processors. Example code is provided with this application note to demonstrate cache memory management. It applies to all derivatives of the Blackfin processor family. This document assumes that the reader is familiar with basic cache terminology.

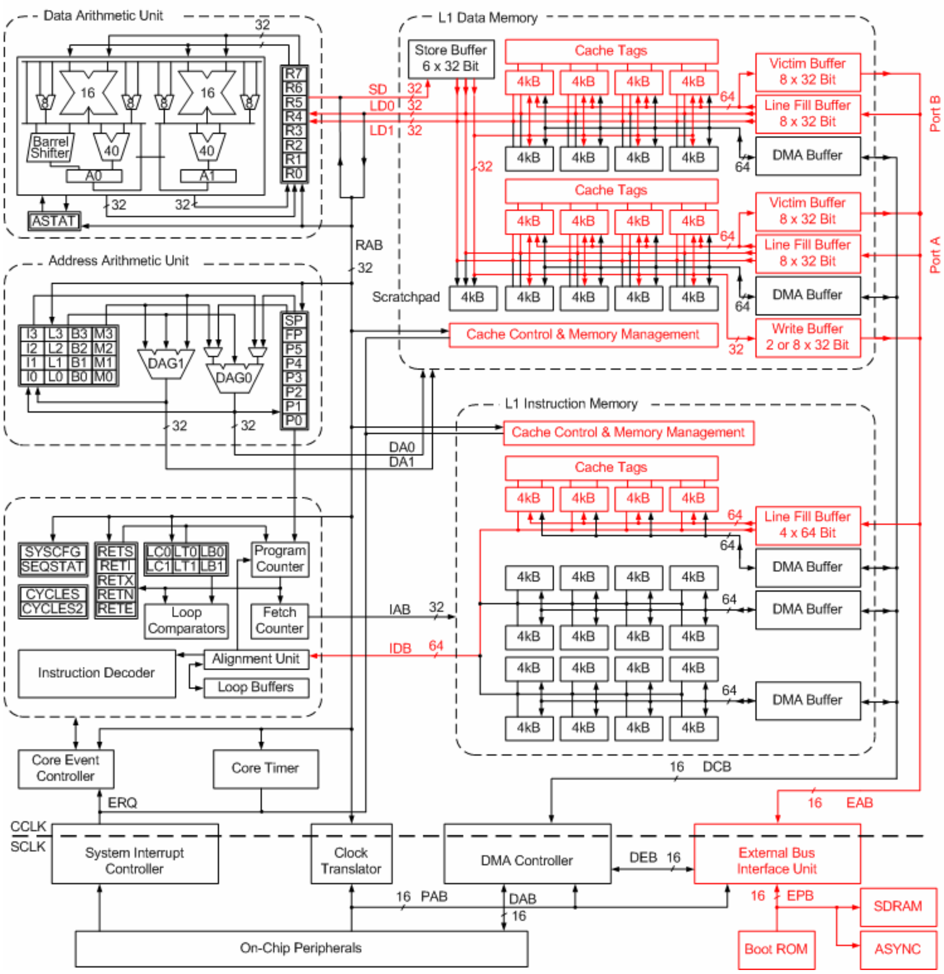


Figure 1. ADSP-BF533 block diagram

Contents

Introduction 1

Contents 2

Cache Memory Concepts 3

Memory Configuration 3

Cache Terminology 3

Block Placement 4

Block Replacement 4

Block Identification 5

Write Strategies..6

Blackfin Cache Model...6

Blackfin Instruction Cache Configuration ..8

Blackfin Data Cache Configuration 11

Memory Protection and Cache Unit..................15

Cacheability Protection Lookaside Buffers (CPLBs) 15

Memory Pages and Page Attributes..................17

CPLB Status Registers..18

CPLB-Related Sequencer Exceptions 20

Page Descriptor Table..20

Coherency Considerations.....................................20

Cache and Instruction Pipeline.......................20

Handling the Instruction and Data Cache........21

Enabling the Cache 21

Instruction Memory Control Register (IMEM_CONTROL) 21

Data Memory Control Register (DMEM_CONTROL) 21

Cache Control Instructions 23

Accessing the Cache Memory 23

Accessing the Instruction Cache 23

Accessing the Instruction SRAM.......................24

Accessing the Data Cache.....................................24

Performance Considerations 25

Instruction Cache: Optimized Conditions ..25

Instruction Cache: Bad Concept.......................25

Data Cache 26

Core Clock vs. System Clock...............................26

Refreshing SDRAM Memory 26

SDRAM Performance 27

Conclusions 27

VisualDSP++® Compiler Support 28

Header Files 28

CPLB Control..28

CPLB Installation 28

Exception Handling...28

Using the Project Wizard 29

Summary of MMRs 30

Cache Configuration for Blackfin Derivatives 30

Conclusion...30

Appendix..31

References...31

Table of Figures...31

List of Tables 32

Document History...32

Cache Memory Concepts

This section discusses the cache memory model in general. The actual Blackfin memory model is discussed in the next section.

Memory Configuration

Systems that require huge amounts of memory generally employ a memory configuration with different memory levels. Memory at the highest level (*internal L1 memory*) provides the highest performance at the highest cost. Lower-level memories have multiple cycles of access times, but cost less (e.g., *external SDRAM memory*).

Cache memory is a high-level memory mastered by the cache controller. It is guarded against direct data access. The cache controller allows larger instruction and data sections to exist in low-level memory, and code and data that are used most frequently are brought into cache (by the cache controller) and thus, are available for single-cycle access, just as though it is in L1 memory. The cache architecture is based on the fact that processor memory space has been sub-divided in to a number of fixed-size blocks (referred to as *cache-lines*). A cache-line is considered to be the smallest unit of memory to be transferred from external memory to the cache memory as a result of a cache miss.

A reference to the memory is identified as a reference to a particular block. Figure 2 depicts a memory configuration in which external memory space has been sub-divided into twenty-four memory blocks and the cache memory is divided into six blocks. (This is an example of a general memory configuration. The number of memory blocks is actually different for the Blackfin memory model.) The block size for the external memory and the cache memory is the same. Since the cache memory has a size of six blocks (in this particular example), at any time, a maximum of six data blocks of the main memory is available in the cache memory.

Cache Terminology

- *Way*: An array of line storage elements in an *m-way* cache.
- *Locked way*: If a way is locked, it does not participate in the *least-recently used* (*LRU*) replacement policy.
- *Set*: A group of *m-way* storage locations in a way of an *m-way* cache. A specific memory block is mapped to a specific *set*. The cache controller chooses a *way* within the *set*.
- *Cache-line*: 32-byte line of memory that is transferred to/from higher-level memory from/to cache.
- *Dirty/clean*: State of cache-line, indicating whether the data in the cache has changed since it was copied from source memory.
- *Cache hit*: The processor references a memory block (in the main memory) that is already buffered in the cache. The processor will access the data from the cache memory.
- *Cache miss*: The processor references a memory block that is not available in the cache. Upon a cache miss, the cache controller moves the referenced memory block from lower-level memory to the cache memory.
- *Victim*: A dirty cache line that must be written to memory before it can be replaced to free space for a cache line allocation (see *victim buffer* in Figure 1).

For more details on cache terminology, refer to the “Memory” chapter of the *Blackfin Processor Programming Reference* [3].

In the following discussions, the words *line*, *cache-line*, and *cache-block* are used interchangeably. Therefore, a block in external memory represents the same amount of memory size as a *cache-line*.

Block Placement

Three schemes are commonly used for deciding the location where the incoming block may be placed in the cache memory.

Direct Mapped Cache

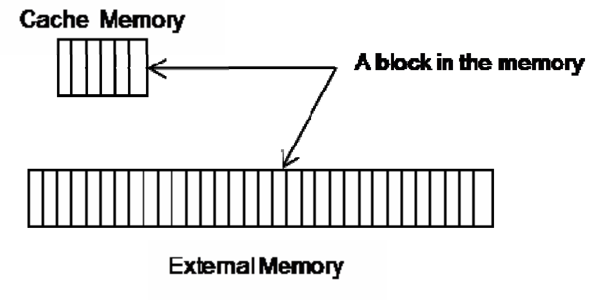


Figure 2. Memory arranges as fixed-size blocks

Every block in the lower-level memory has only one fixed destination block in the cache. There exists a one-to-one mapping from lower-level memory to the cache memory. This mapping is based on the address of the block in the lower-level memory. This scheme has the lowest management complexity and flexibility.

Fully Associative Cache

A block in the main memory can replace any block in the cache memory. The mapping is completely random. This strategy has the highest management complexity but also provides the highest flexibility.

Set Associative Cache

Cache memory is arranged in sets. A set consists of a number of blocks. Any block in the lower-level memory has a fixed destination set (in which it can be placed) in the cache. The incoming block may replace any of the blocks within this associated set. If there are *m* blocks in a set, the cache configuration is called an *m-way set associative*. This block placement method is a compromise between management complexity and flexibility. Figure 3 depicts a 2-way set associate memory.

The first two schemes are special cases of the set associative cache (direct mapped for *m = 1*, and fully associative for *m = number of cache-lines).*

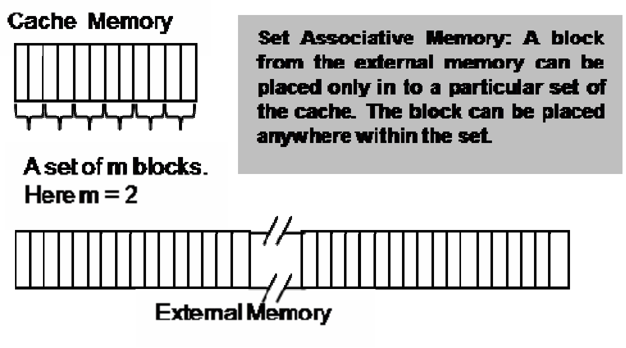

Cache Memory
Set Associative Memory: A block from the external memory can be placed only in to a particular set of the cache. The block can be placed anywhere within the set.
A set of m blocks. Here m = 2
External Memory

Figure 3. Configuration for a 2-way set associative cache memory

Block Replacement

Figure 4 shows a comparison of block placement for a 1-way and a 4-way set associative cache. Consider a linear flow through a large loop program in external memory. When cache locations are overwritten before they can be used again, cache is “trashed”. A 4-way cache is much more likely to have the required instruction or data word still present in the cache to re-use it again than a 1-way cache. Code must be re-used to make cache attractive.

In direct mapped cache, there is always a fixed location inside the cache memory, at which a specific part from cacheable (external) memory can be placed. However, with fully associative or set-associative placement, multiple locations may be chosen on a cache-miss. The blocks that can possibly be replaced are called *participating blocks*. Following are some strategies primarily employed for selecting the block to be replaced:

- *Random replacement*: The destination block is randomly selected out of participating blocks. A pseudo-random number generator may be used to select the destination block. This is the simplest, but least efficient implementation.
- *First in, first out (FIFO) replacement*: The incoming block replaces the oldest participating block.

- *Least-recently used (LRU) replacement*: Under this scheme, all accesses to blocks are recorded. The replaced block is the one that has been unused for the longest time. LRU replacement relies on a corollary of locality: If recently used blocks are likely to be used again, a good candidate for disposal is the least-recently used block.
- *Modified LRU replacement*: Modified in the case of a Blackfin processor means that bit 8 (`CPLB_LRUPRIO`) in the CPLB data registers is set or cleared (see Figures 17 and 18). This bit has higher priority than LRU policy. Under this scheme, a block is assigned a low priority or a high priority.

  If the incoming block is a low-priority block, only low-priority blocks can participate in the cache replacement.

  If the incoming block is of high priority, all low-priority blocks will participate in the cache replacement. If there are no low-priority blocks, the high-priority blocks can participate in the replacement policy.

  LRU policy is used to choose the victim block among participating blocks.

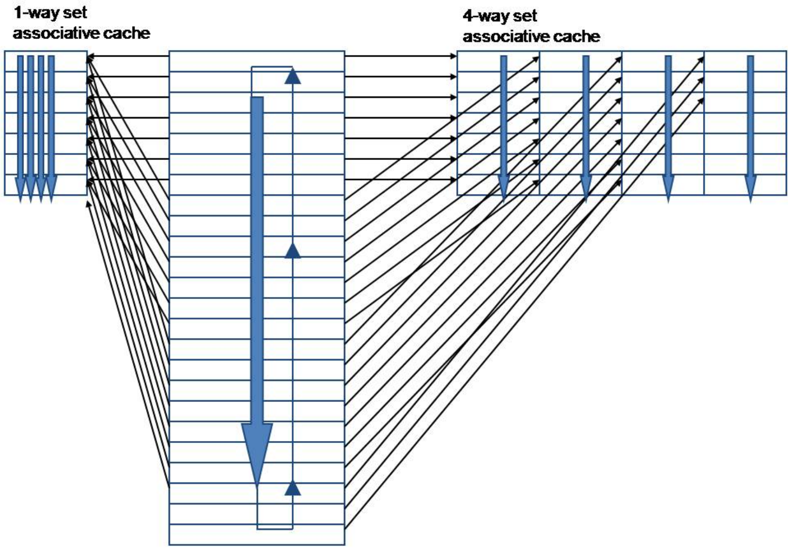


Figure 4. Block placement

Block Identification

Block Address		Block Offset
Tag Field	**Index Field**	
Used by the cache controller to determine a cache hit or miss	Used to map a given block to a particular set	Used to select a word within the given block

Table 1. Address partitioning

Not all the cache-blocks may have valid information contents. For example, at system startup, cache memory will not contain any valid data. The cache-blocks will be filled whenever a cache-miss is encountered. A mechanism must identify whether a cache-block has a valid data entry in it. To identify this, a so-called “valid” bit is associated with each cache-block. When a cache-block is filled with valid data, the corresponding valid bit is set

In addition to a valid bit, every block in the cache also has an address tag associated with it. This address tag provides the physical address of cached block in external memory. When the external memory block is referenced, the cache controller compares external address with cache address tags (with a valid bit) for the set that might contain this block. If the block address matches one of the cache address tags, it results in a cache hit.

Under a set-associative cache configuration, each memory address can be viewed as a combination of three fields (Table 1). The first division is between the block address and the block offset. The block (frame) address can be further divided into the tag field and the index field.

The block offset field selects the desired data from the block. The index field selects the set, and the tag field is compared against it for a hit.

Write Strategies

The following section discusses two different issues with the memory write operations associated with data cache.

Write Operations with a Cache Hit

There are two basic options when writing data back to external memory:

- *Write-through (WT):* The information is written to both the block in the cache and to the block in the source memory. The behavior/performance is similar to write accesses to external memory without having cache enabled.
- *Write-back (WB):* The information is written only to the block in the cache. When modified, the victim cache-line is written to the main memory only when it is replaced.

A store operation, especially in Write-Through operation mode, will only update the modified data. No complete cache-line is written to external memory if not necessary.

To reduce the frequency of writing back blocks on replacement, a feature called *dirty bit* is commonly used. This status bit indicates whether the block is dirty (modified while in the cache) or clean (not modified). If the block is clean, it is not written back during replacement. Although it is application dependent, write-back mode yields about a 10-15% improvement over write-through mode. Write-through mode is best when coherency must be maintained between more than one resource (e.g., DMA and core).

Write Operations with a Cache Miss

Since the data are not needed on a write, there are two options on a write miss:

- *Write allocate*: The block is allocated on a write miss, followed by the write-hit actions described above. In this scheme, a write miss act like a read miss. First, a cache-line is fetched from external memory before updating both internal cache and source memory.
- *No-write allocate*: Under this option, write-misses do not affect the cache. The block is modified only in lower-level memory and no caching takes place.

Blackfin Cache Model

Blackfin processors have (up to) three levels of memory, providing a trade-off between capacity and performance. Level 1 memory (also known as L1 memory) provides the highest performance with the lowest capacity. Level 2 (L2) memory and level 3 (L3) memory provide much larger memory sizes, but typically have multiple cycle access times.

Blackfin processors have on-chip (L1) data and instruction memory banks, which can be independently configured as SRAM or cache. When the memory is configured as cache, neither the DMA controller nor the core's load/store instructions can access its content. All Blackfin processors have a similar cache configuration (see Table 2), but as a reference, the following discussion is based on ADSP-BF533 devices.

The internal L1 memory of the Blackfin processor is connected via the External Access Bus (EAB) to the External Bus Interface Unit (EBIU), which manages accesses to external memory (e.g., SDRAM memory). Any ongoing activity on the EAB cannot be interrupted. An ongoing cache-line fill will always finish once started.

	Start Address:	Derivative:	ADSP-BF51x	ADSP-BF52x	ADSP-BF531	ADSP-BF532	ADSP-BF533	ADSP-BF536	ADSP-BF534/7	ADSP-BF538/9	ADSP-BF54x	ADSP-BF561 Core A	ADSP-BF561 Core B
Internal Memory L1	0xFFE0 0000	Core MMR	2MB	2MB	2MB	2MB	2MB	2MB	2MB	2MB	2MB	2MB	2MB
	0xFFC0 0000	System MMR	2MB	2MB	2MB	2MB	2MB	2MB	2MB	2MB	2MB	2MB	
	0xFFB0 1000	RESERVED											
	0xFFB0 0000	Scratchpad	4kB	4kB	4kB	4kB	4kB	4kB	4kB	4kB	4kB	4kB	
	0xFFA1 4000	Instr. ROM	RESERVED								64kB		
	0xFFA1 0000	Instr. SRAM/Cache	16kB	16kB	16kB	16kB	16kB	16kB	16kB	16kB	16kB	16kB	
	0xFFA0 C000	Instr. SRAM				16kB	16kB			16kB			
	0xFFA0 8000	Instr. SRAM		16kB	16kB	16kB	16kB	16kB	16kB	16kB	16kB		
	0xFFA0 4000	Instr. SRAM	16kB	16kB			16kB	16kB	16kB	16kB	16kB		
	0xFFA0 0000	Instr. SRAM	16kB	16kB			16kB	16kB	16kB	16kB	16kB	16kB	
	0xFF90 8000	RESERVED											
	0xFF90 4000	Data B SRAM/Cache	16kB	16kB		16kB	16kB	16kB	16kB	16kB	16kB	16kB	
	0xFF90 0000	Data SRAM	16kB	16kB			16kB		16kB	16kB	16kB	16kB	
	0xFF80 8000	RESERVED											
	0xFF80 4000	Data A SRAM/Cache	16kB	16kB	16kB	16kB	16kB	16kB	16kB	16kB	16kB	16kB	
	0xFF80 0000	Data SRAM	16kB	16kB			16kB		16kB	16kB	16kB	16kB	
	0xFF70 1000												
	0xFF70 0000	Scratchpad											4kB
	0xFF61 4000												
	0xFF61 0000	Instr. SRAM/Cache											16kB
	0xFF60 4000												
	0xFF60 0000	Instr. SRAM											16kB
	0xFF50 8000	RESERVED											
	0xFF50 4000	Data B SRAM/Cache											16kB
	0xFF50 0000	Data B SRAM											16kB
	0xFF40 8000												
	0xFF40 4000	Data A SRAM/Cache											16kB
	0xFF40 0000	Data A SRAM											16kB
	0xFEB2 0000												
L2	0xFEB0 0000	L2 SRAM	RESERVED								128kB	128kB	
External Memory L3		RESERVED											
	0xEF00 0000	Boot ROM	32kB	32kB	1kB	1kB	1kB	2kB	2kB	1kB	4kB	2kB	
		RESERVED											
	0x2030 0000	ASYNC Bank 3	1MB	1MB	1MB	1MB	1MB	1MB	1MB	1MB	64MB	1MB	
	0x2020 0000	ASYNC Bank 2	1MB	1MB	1MB	1MB	1MB	1MB	1MB	1MB	64MB	1MB	
	0x2010 0000	ASYNC Bank 1	1MB	1MB	1MB	1MB	1MB	1MB	1MB	1MB	64MB	1MB	
	0x2000 0000	ASYNC Bank 0	1MB	1MB	1MB	1MB	1MB	1MB	1MB	1MB	64MB	1MB	
		RESERVED											
	0x0000 0000	SDRAM Memory	128MB	128MB	128MB	128MB	128MB	512MB	512MB	128MB	512MB	128MB	

Table 2. Blackfin processors memory map

Blackfin Instruction Cache Configuration

Cache Organization

The ADSP-BF533 processor has 80K bytes of on-chip instruction memory. 64K bytes of instruction memory are available as instruction SRAM. The remaining 16K bytes of memory can be configured as instruction cache or can be used as instruction SRAM. Table 2 shows a combined memory map for all currently available Blackfin processor family members.

All Blackfin processor derivatives offer the same amount of instruction memory available that is configurable as either instruction cache or SRAM. The start address in the memory map is always identical.

When enabled as cache, the instruction memory works as a *4-way set associative* memory. Each of the four ways can be locked independently. The instruction cache-controller can be configured to use the modified LRU policy or the LRU policy for cache-line replacement.

- The 16K-byte cache is arranged as four 4K-byte subbanks. A subbank is selected by memory address bits `[13:12]`.
- Each 4K-byte subbank consists of 32 *sets*. A set is selected by memory address bits `[9:5]`.
- Each set consists of four *ways*. A way is selected by the cache controller according to the cache-line placement policy. The ways can be identified by the address bits `[11:10]` in SRAM.
- In other words, a *set-0* represents the four ways of *line-0*.
- The size of a *cache-line* (to be read on a cache miss) is 32 bytes.

The external read data port (for cache controller) is 64 bits (8 bytes) wide. Hence, the cache controller reads the complete cache-line as a burst of four 8-byte-wide chunks of data.

Each line has a tag portion associated with it. The tag portion consists of four parts:

- 20-bit *Address Tag*: Compared against memory address to determine cache-hit or cache-miss.
- *LRU Priority*: Priority for modified LRU policy.
- *LRU State*: To be used by cache controller for LRU policy.
- *Valid Bit*: Valid data in line.

The 32-bit address space is mapped to cache memory space as following:

- *Subbank Select*: Select a particular 4K-byte subbank.
- *Set Select*: Select a set out of 32 cache sets.
- *Byte Select*: Select a byte within the given line.

Figure 5 shows how a 4K-byte subbank in the instruction cache is arranged. Each 4K-byte subbank provides the same structure.

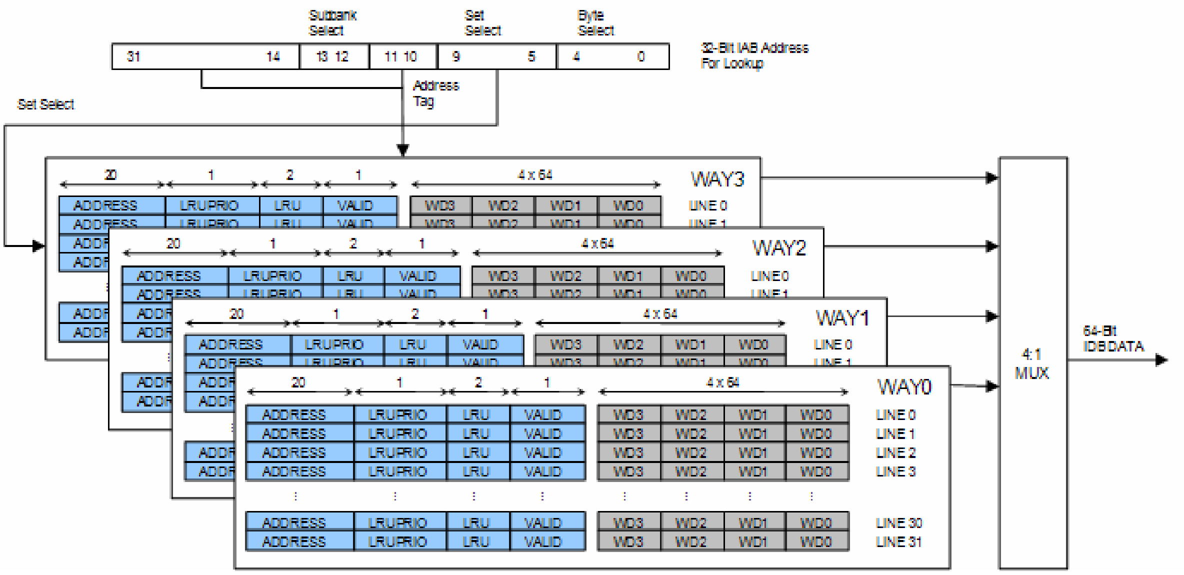


Figure 5. Instruction cache configuration for one subbank – 4-way set associative

Situation	Strategy
Only one invalid way in the set	Incoming block would replace this block
More than one invalid way in the cache	The invalid ways would be replaced in the following order: Way0 first, then way1, then way2, and finally way3
No invalid ways in the cache	Least-recently used (LRU) way would be replaced. For Modified LRU policy: Way with high priority would not be replaced if a low-priority way exists in the given set. A low-priority block cannot replace a high-priority block. If all ways are high priority, the low-priority way cannot be cached.

Table 3. Line replacement policy for the Blackfin instruction cache

Cache Hits/Misses and Cache-Line Replacement

A cache hit is determined by comparing the upper 18 bits and bits 11 and 10 of the instruction fetch address to the address tag of valid lines currently stored in a cache set. If the address-tag comparison operation results in a match, a cache hit occurs. If the address-tag compare operation does not result in a match, a cache miss occurs.

Consider an access to the address `0x10374956`. This address is mapped to set-10 of subbank 0. The upper 18 bits and bits 11-10 of this address (this forms the tag word) will hence be compared against all the valid tags of set-10.

On a cache miss, the instruction memory unit generates a cache-line fill access to retrieve the missing cache-line from the external memory. The core stalls until the target instruction word is returned from external memory.

The address for the external memory access is the address of the target instruction word. The cache-line replacement unit uses the `Valid` and `LRU` bits (of unlocked ways) to determine which block (in the given set) is to be used for the new cache-line. Table 3 shows how the cache-line replacement policy is selected.

The cache-line fill access consists of fetching 32 bytes (one block) of data from memory. The address for the read transfer is the address of the target instruction word. When responding to a line-read request from the instruction memory unit, the external memory returns the target instruction 64-bit word first. The next three words are fetched in sequential address order as depicted in Table 4.

Target word	Fetching order for next three words
WD0	WD0, WD1, WD2, WD3
WD1	WD1, WD2, WD3, WD0
WD2	WD2, WD3, WD0, WD1
WD3	WD3, WD0, WD1, WD2

Table 4. Instruction cache-line word fetching order

When the cache-block is retrieved from the external memory, each 64-bit word is buffered in a four-entry *line fill buffer* before it is written to a 4K-byte memory bank. The *line fill buffer* allows the core to access the data from the new cache-line as the line is being retrieved from external memory, rather than having to wait until the line has been written in to the cache.

Instruction Fetch Latency: Cache vs. no Cache

- *Cache off*: 64 bits / 8 bytes are fetched
- *Cache on*: always 256 bits / 32 bytes (burst fill) are fetched
- As the SDRAM interface is 16 bits wide on most of the Blackfin derivatives, each instruction fetch is a sequence of 16-bit accesses.

Figure 6 shows an instruction fetch from external SDRAM memory (*target address* `0x000Ah`) with instruction cache turned off. A 64-bit block is fetched (defined as `WDx`). *Start address* is `0x0008h` as the access (instruction fetch) is 64-bit aligned.

Figure 7 shows the same access but with instruction cache turned on. The start address is `0x0008h` as well. This is the beginning of `WD1`. `WD2` starts at address `0x0010h`, and `WD3` starts at address `0x0018h`. `WD0` at address `0x0000h` is the last part of the cache-block to be transferred.

Figure 8 and Figure 9 show the execution latency for instructions stored in external memory (start address `0x00000`). The signal "GPIO" is toggled from 1 to 0 by the function in external memory. The function looks like the following listing:

```
[--SP] = (R7:7,P5:5);
P5.H = hi(FIO_FLAG_C);
P5.L = hi(FIO_FLAG_C);
R7 = 0x20 (z);
w[P5] = R7;
ssync;
(R7:7,P5:5) = [SP++];
rts;
```

For an access to SDRAM memory, there are several operations beside the actual memory read (`RD: Read command`). For more details on SDRAM performance, see Performance Considerations on page 25.

Figure 8 and Figure 9 show the first access to a specific bank in SDRAM memory. The target row must be opened in the particular bank (`ACT: Active command`) first.

After the `ACTIVE-to-READ-or-WRITE-delay` (t_{RCD}, here 3 `SCLK` cycles), the read command can be sent. After the `CAS latency` (`CL`, here 3 `SCLK` cycles), the first piece of data is available.

The latency for both instruction cache on and instruction cache off is the same so far. For the first 64-bit block of instructions, it makes no differences. Execution time is identical. After about 10 `SCLK` cycles, the first instruction is executed.

The advantage of the instruction cache is that the next piece of code – and usually the program flow is linear – is fetched right after the first 64-bit block. Compared to a cache-line length of 256 bits, it takes about 4x11 (plus `ACT`) `SCLK` cycles until the last piece of data is available on the `EAB` when instruction cache is not enabled.

With instruction cache enabled, the same operation is done within 21 `SCLK` cycles (including `ACT`).

For more information on instruction fetch latency for internal L2 memory, refer to the "Memory" chapter in the *Blackfin Processor Programming Reference* [3].

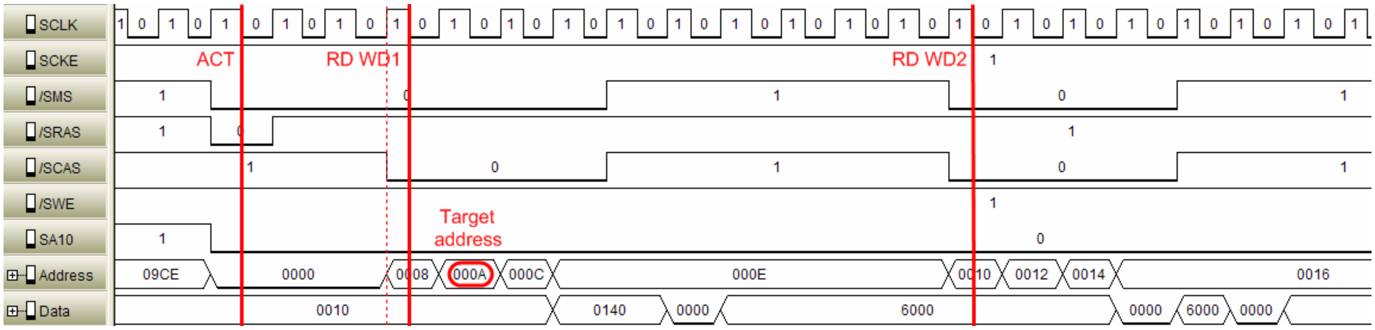


Figure 6. Instruction fetch with cache off

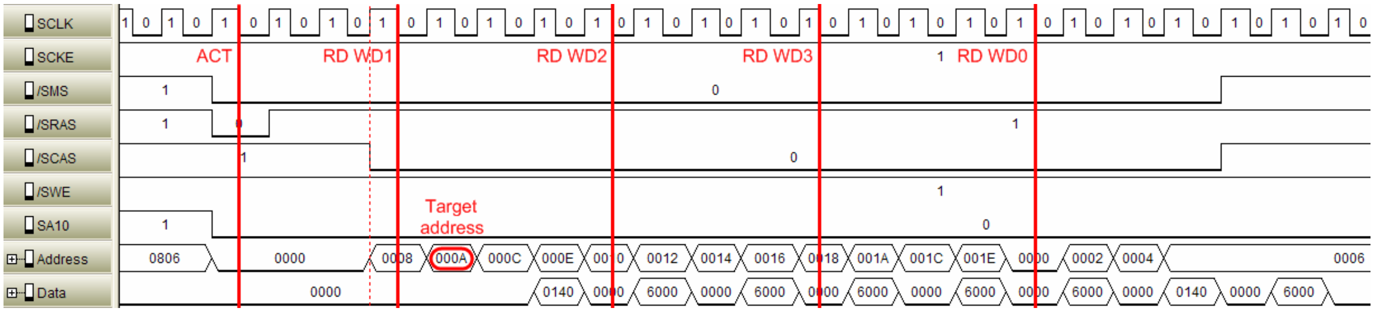


Figure 7. Instruction fetch with cache on

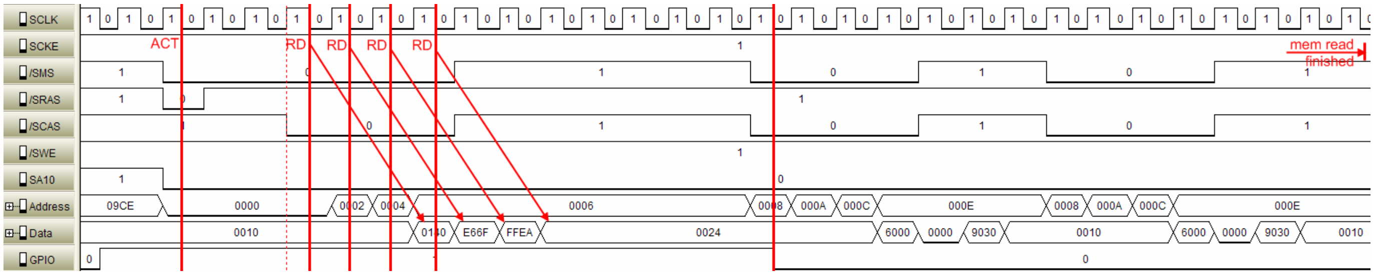


Figure 8. Latency with instruction cache off

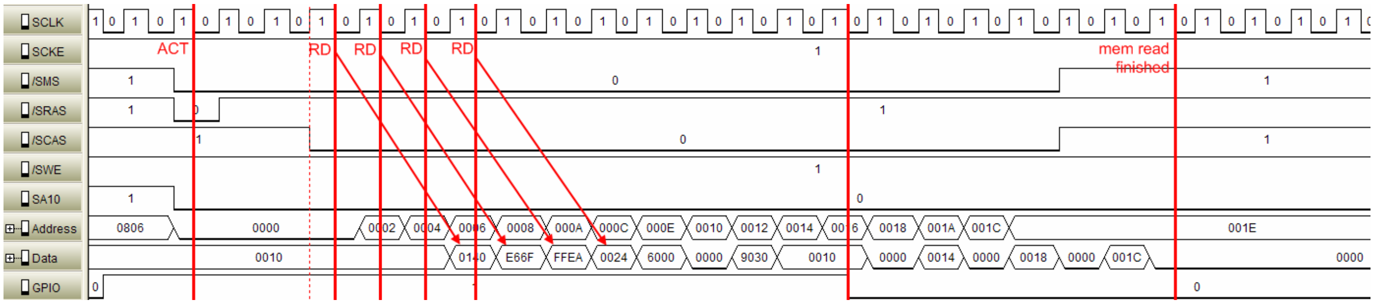


Figure 9. Latency with instruction cache on

Blackfin Data Cache Configuration

Cache Organization

The ADSP-BF533 has 64K bytes of on-chip data memory. 32K bytes of data memory are available as data SRAM. The remaining 32K bytes of memory are available as two independent 16K-byte memory banks, which can be configured as either data cache or data SRAM.

Some Blackfin processor derivatives offer only one bank of data memory that is configurable as either data cache or SRAM. See Table 2 for more details.

When enabled, the data cache works as *2-way set associative* memory. The data cache-controller uses LRU policy for cache-line replacement (it cannot use modified LRU policy as is the case with instruction cache).

- Depending on the number of memory banks configured as cache, the data cache is either 16K bytes (one bank) or 32K bytes (two banks).
- Each 16K-byte cache bank is arranged as four 4K-byte subbanks. A subbank is selected by memory address bits `[13:12]`.
- Each 4K-byte subbank consists of 64 *sets*. A set is selected by memory address bits `[10:5]`.
- Each set consists of two *ways*. A way is selected by the cache controller according to the cache-line placement policy. The ways can be identified by the address bit `[11]` in SRAM.
- In other words, a *set-0* represents the two ways of *line-0*.
- The size of a *cache-line* (to be read on a cache miss) is 32 bytes.

Similar to the instruction cache, each cache-line has a tag portion associated with it (see Figure 10):

- 19-bit *address tag*: Compared against memory address to determine cache-hit or cache-miss.
- *Dirty bit*: Cache-line has been modified.
- *LRU state*: To be used by cache controller for LRU policy.
- *Valid bit*: Valid data in line.

The 32-bit address space is mapped to cache memory space as following (see Figure 10):

- *Subbank select*: Select a particular 4K-byte subbank.
- *Set select*: Select a set out of 64 cache sets.
- *Byte select*: Select a byte within the given line.

Figure 10 shows how a 4K-byte subbank in the data cache is arranged. When both data banks are enabled as cache, depending on the state of `DCBS` bit, either bit 14 (every 16K bytes) or bit 23 (every 8M bytes) of the address space is used to select one of 16K-byte data banks (see Figure 11).

The cache-line fill access consists of fetching 32 bytes of data from memory. The address for the read transfer is the address of the target data word. Assuming a 16-bit read access on a 16-bit I/O memory, when responding to a line-read request from the data memory unit, the external memory returns the target data 16-bit word first. The next 15 words are fetched in sequential address order as depicted in Table 5.

Target word	**Fetching order for next 15 words**
WD0	WD0, … , WD15
WD1	WD1, … , WD15, WD0
WD2	WD2, …, WD15, WD0, WD1
WD15	WD15, WD0, … , WD14

Table 5. Data cache-line word fetching order

When the cache-line is retrieved from the external memory, each 32-bit word is buffered in an eight-entry *line fill buffer* before it is written to a 4K-byte memory bank. The *line fill buffer* allows the core to access the data from the new cache-line as the line is being retrieved from external memory, rather than having to wait until the line has been written in to the cache.

Unlike instructions, data are usually modified and written back to (external) memory. Any cacheable write-through single (non-burst) writes from the core to the external memory go through the *write buffer*. The depth of this buffer can vary according to the settings in the interrupt priority register (IPRIO).

A third buffer is used to read a dirty cache-line being flushed or being replaced by a cache-line fill and then to initialize a burst write operation on the bus to perform the line copy-back to the external memory. This buffer is called a *victim buffer*. It is implemented like the *line fill buffer* as an 8-entry-deep FIFO, 32-bit-wide each (see Figure 1).

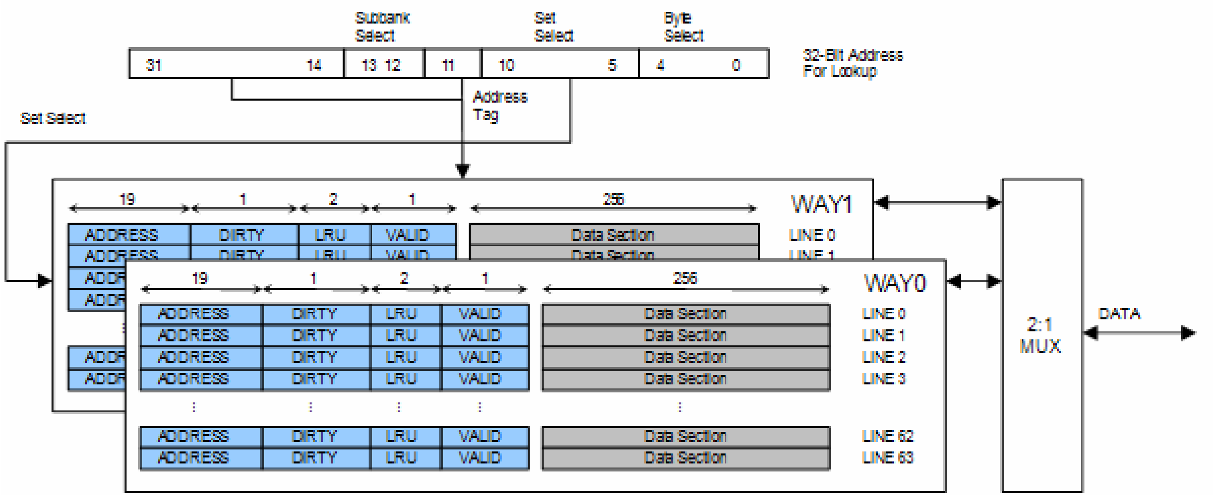


Figure 10. Data cache configuration for one subbank – 2-way set associative

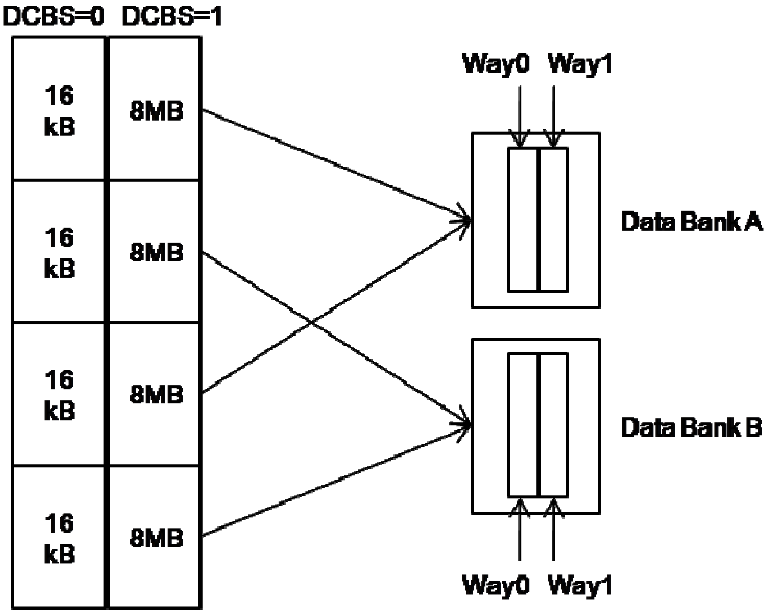


Figure 11. Data cache mapping according to DCBS bit

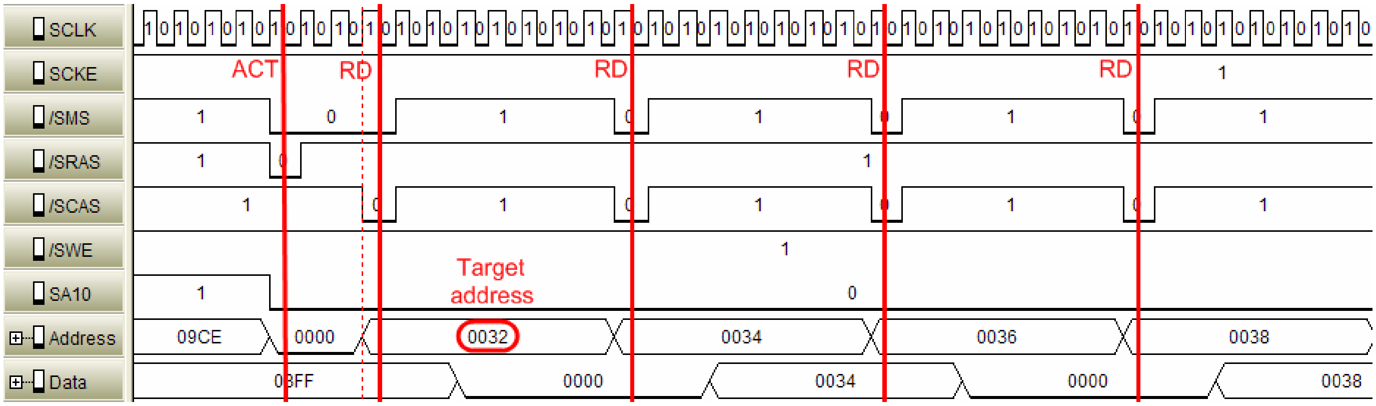


Figure 12. Data fetch with cache off

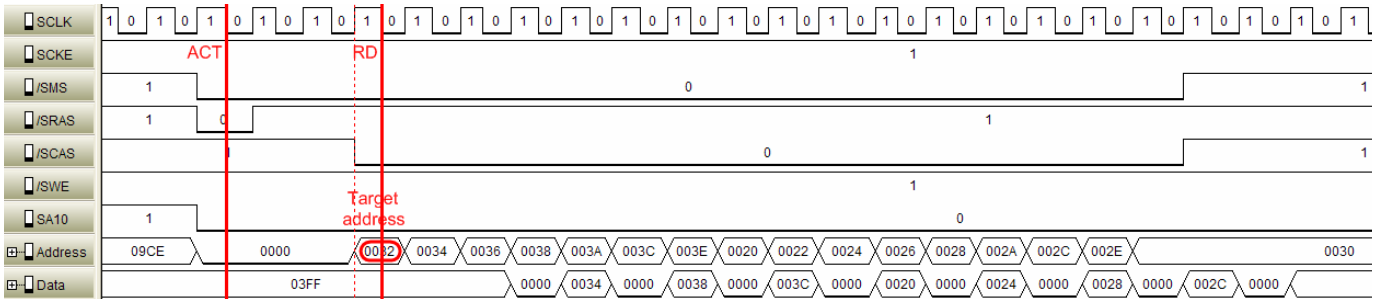


Figure 13. Data fetch with cache on

Data Fetch from External Memory

- *Cache off*: 1 byte, 2 bytes, or 4 bytes wide accesses
- *Cache on*: always 256 bits / 32 bytes (burst fill) are fetched
- As the SDRAM interface is 16 bits wide on most Blackfin derivatives, each access is 16 bits wide. For a byte access, the lower or upper byte is masked out by the `SDQM` signals.

Figure 12 shows a 16-bit data read access to the external memory (*target address* `0x0032h`). Data cache is turned off. For each access, one 16-bit word is fetched.

Figure 13 shows the same access, but with data cache turned on. The start address here is `0x0032h`. The other data belonging to this cache-line are fetched as depicted in Table 5.

Cache Write Methods

The external memory is divided into different pages (defined by data cache protection lookaside buffers — DCPLB registers). The attributes for each page can be configured independently. As discussed in the next section, the memory pages can be:

- Configured either in write-back mode or in write-through mode (`CPLB_WT` bit).
- Configured to allocate the cache-lines either on reads only or on reads and writes (`CPLB_L1_AOW` bit).

	Cache-line valid	**Cache-line invalid**
WT AOW = 0	Update cache-line & external memory	Update external memory only
WT AOW = 1	Update cache-line & external memory	Fetch cache-line Update cache-line Update ext. mem.
WB	Update cache-line only	Fetch cache-line Update cache-line

Table 6. Behavior in specific write situation

Table 6 describes the behavior with the different cache write methods. Figure 14 illustrates when data cache is enabled in write-through mode and with `CPLB_L1_AOW` bit set. A write access to address 0x0040h is initiated. First, a cache-line fill is done, followed by the write to the external memory.

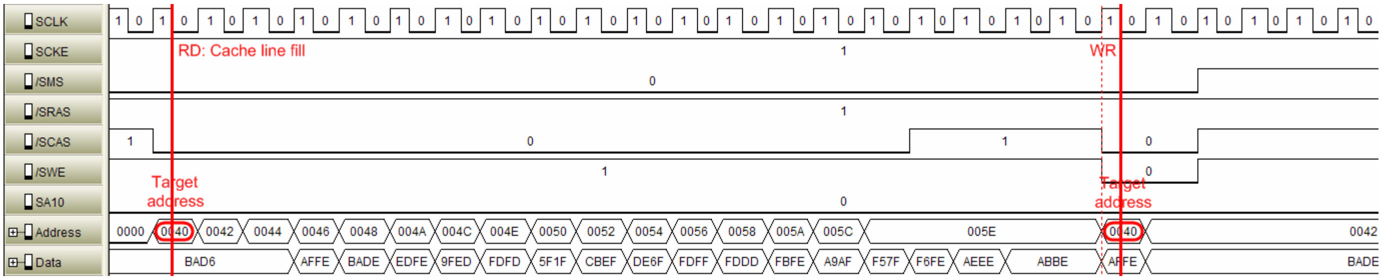


Figure 14. Data fetch with cache on in write-through mode and AOW enabled

Memory Protection and Cache Unit

The Blackfin processor contains a page-based *Memory Protection and Cache Unit* (*MPCU*), which provides control over cacheability of memory ranges and management of protection attributes at a page level. The MPCU is implemented as two 16-entry content addressable memory (CAM) blocks. Each entry is referred to as a cacheability protection lookaside buffer (CPLB) descriptor. The MPCU functionality includes:

- Caching and protection lookaside buffers (CPLBs)
- Cache/protection properties determined on a per memory page basis (1KB, 4KB, 1MB, and 4MB sizes)
- User/supervisor, and task/task protection

Cacheability Protection Lookaside Buffers (CPLBs)

Each entry to the CPLB descriptors defines cacheability and protection attributes for the given memory page. The CPLB entries are divided between instruction and data CPLBs. 16 CPLB entries (called ICPLBs) are used for instruction fetch requests. Another 16 CPLB entries (called DCPLBs) are used for data transactions. Setting the appropriate bits in the instruction memory control (`IMEM_CONTROL`) and data memory control (`DMEM_CONTROL`) registers enable the ICPLBs and DCPLBs. Each CPLB entry consists of a pair of 32-bit values. Before loading descriptor data into any CPLBs, the corresponding group of 16 CPLBs must be disabled using the `ENICPLB` or `ENDCPLB` bits in the instruction memory control register (`IMEM_CONTROL`) and data memory control register (`DMEM_CONTROL`), respectively.

For Instruction Fetches

`ICPLB_ADDR[n]` defines the start address of the page described by the CPLB descriptor.

`ICPLB_DATA[n]` defines the properties of the page described by the CPLB descriptor. Figure 15 depicts various bit-fields and their functionality in the `ICPLB_DATA` register.

For Data Operations

`DCPLB_ADDR[m]` defines the start address of the page described by the CPLB descriptor.

`DCPLB_DATA[m]` defines the properties of the page described by the CPLB descriptor. Figure 16 depicts various bit-fields and their functionality in the `DCPLB_DATA` register.

Using CPLBs

- Cache enabled: CPLB must be used to define cacheability properties.

- Cache disabled: CPLBs <u>can</u> be used to protect pages of memory.
- If CPLBs are used, a valid CPLB entry must exist before an access to a specific memory location is attempted. Otherwise, an exception will be generated.
- There are two default CPLB descriptors for data accesses to the scratchpad data memory and to the system and core MMR space. These default descriptors define the above space as non-cacheable, so that additional CPLB's do not need to be set up for these regions of memory.

If valid CPLBs are set up for this space, the default CPLBs are ignored.

On newer Blackfin derivatives such as ADSP-BF51x, ADSP-BF52x, and ADSP-BF54x, the on-chip boot ROM (read-only memory) provides functions (`SysControl()`) for accessing the PLL and voltage regulator registers. Additionally, internal L1 instruction ROM and/or L2 SRAM is available as well. For all these cases, a valid CPLB descriptor (instruction and data) is required. Refer to the processor's data sheet or Table 2 for more details on the amount of memory available on you Blackfin derivative.

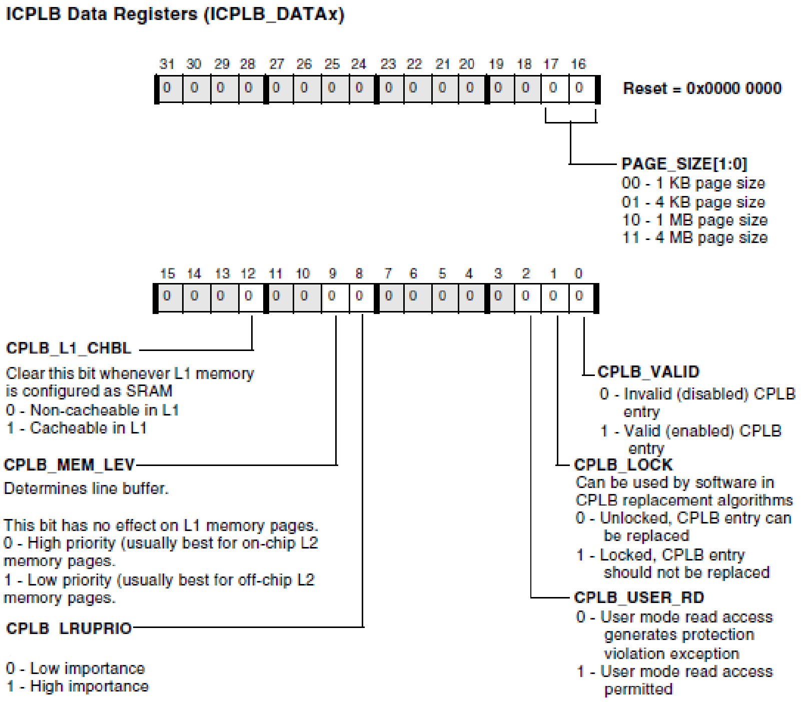

ICPLB Data Registers (ICPLB_DATAx)
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000 0000
PAGE_SIZE[1:0]
00 - 1 KB page size
01 - 4 KB page size
10 - 1 MB page size
11 - 4 MB page size
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
CPLB_L1_CHBL
Clear this bit whenever L1 memory is configured as SRAM
0 - Non-cacheable in L1
1 - Cacheable in L1
CPLB_MEM_LEV
Determines line buffer.
This bit has no effect on L1 memory pages.
0 - High priority (usually best for on-chip L2 memory pages.
1 - Low priority (usually best for off-chip L2 memory pages.
CPLB_LRUPRIO
0 - Low importance
1 - High importance
CPLB_VALID
0 - Invalid (disabled) CPLB entry
1 - Valid (enabled) CPLB entry
CPLB_LOCK
Can be used by software in CPLB replacement algorithms
0 - Unlocked, CPLB entry can be replaced
1 - Locked, CPLB entry should not be replaced
CPLB_USER_RD
0 - User mode read access generates protection violation exception
1 - User mode read access permitted

Figure 15. Bit fields and their functionalities for the ICPLB_DATA registers

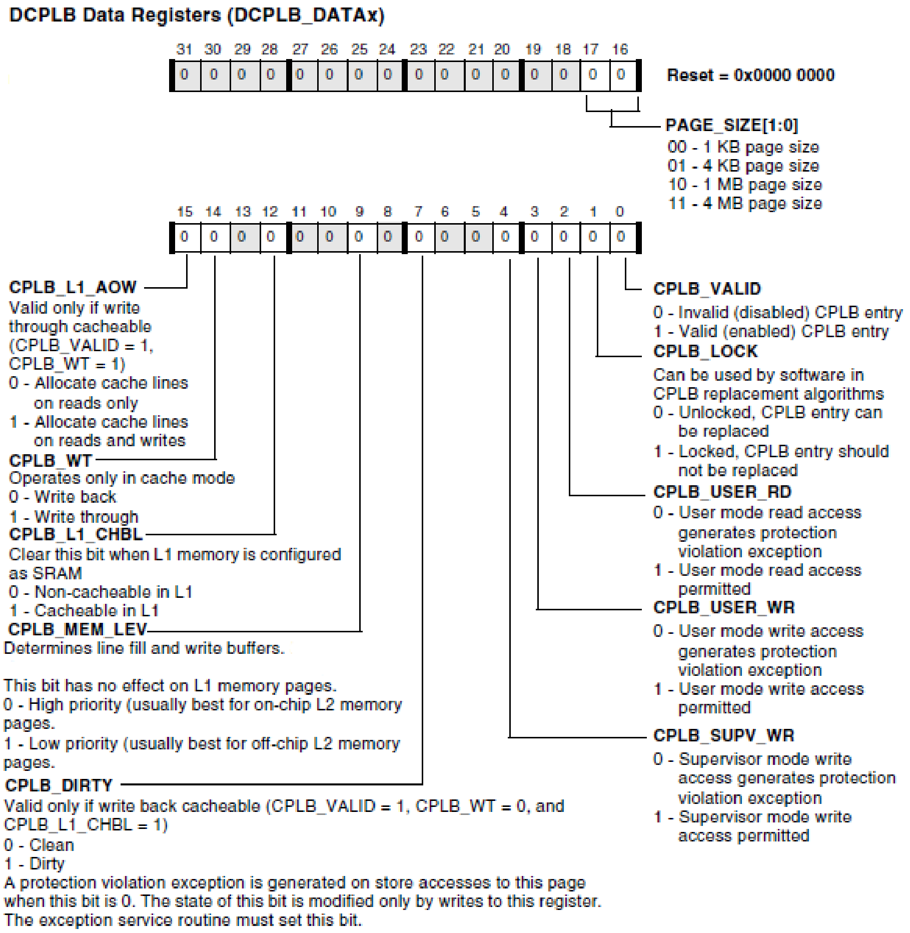

DCPLB Data Registers (DCPLB_DATAx)
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000 0000
PAGE_SIZE[1:0]
00 - 1 KB page size
01 - 4 KB page size
10 - 1 MB page size
11 - 4 MB page size
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
CPLB_L1_AOW
Valid only if write through cacheable (CPLB_VALID = 1, CPLB_WT = 1)
0 - Allocate cache lines on reads only
1 - Allocate cache lines on reads and writes
CPLB_WT
Operates only in cache mode
0 - Write back
1 - Write through
CPLB_L1_CHBL
Clear this bit when L1 memory is configured as SRAM
0 - Non-cacheable in L1
1 - Cacheable in L1
CPLB_MEM_LEV
Determines line fill and write buffers.
This bit has no effect on L1 memory pages.
0 - High priority (usually best for on-chip L2 memory pages.
1 - Low priority (usually best for off-chip L2 memory pages.
CPLB_DIRTY
Valid only if write back cacheable (CPLB_VALID = 1, CPLB_WT = 0, and CPLB_L1_CHBL = 1)
0 - Clean
1 - Dirty
A protection violation exception is generated on store accesses to this page when this bit is 0. The state of this bit is modified only by writes to this register. The exception service routine must set this bit.
CPLB_VALID
0 - Invalid (disabled) CPLB entry
1 - Valid (enabled) CPLB entry
CPLB_LOCK
Can be used by software in CPLB replacement algorithms
0 - Unlocked, CPLB entry can be replaced
1 - Locked, CPLB entry should not be replaced
CPLB_USER_RD
0 - User mode read access generates protection violation exception
1 - User mode read access permitted
CPLB_USER_WR
0 - User mode write access generates protection violation exception
1 - User mode write access permitted
CPLB_SUPV_WR
0 - Supervisor mode write access generates protection violation exception
1 - Supervisor mode write access permitted

Figure 16. Bit fields and their functionalities for the DCPLB_DATA registers

Memory Pages and Page Attributes

Each CPLB entry corresponds to a valid memory page. Every address within a page shares the attributes defined for that page. The address descriptor `xCPLB_ADDR[n]` provides the base address of the page in memory. The property descriptor word `xCPLB_DATA[n]` specifies size and attributes for the page. Figure 15 and Figure 16 depict various bit-fields and their functionality in the `ICPLB_DATAx` registers and `DCPLB_DATAx` registers), respectively. A short description of the bits follows.

Page Size

The Blackfin memory architecture supports four different page sizes – 1KB, 4KB, 1MB, or 4MB. Pages must be aligned on page boundaries that are an integer multiple of their size.

Cache-Line Allocation

Present on data memory only, with write-through cache enabled. The `CPLB_L1_AOW` bit controls whether a cache-line fill is triggered by a read only or on a write access as well. For writes with the `CPLB_L1_AOW` bit not set, cache behaves as if it is not present. If this bit is set, a cache-line fill is triggered first followed by updating internal and external memory.

Write-Through/Write-Back Flag

Present on data memory only, this attribute (`CPLB_WT` bit) enables the write-through mode for the data cache when set. By default (`CPLB_WT` = 0), write-back mode is active.

Cacheable/Non-cacheable Flag

If a page is defined as non-cacheable (`CPLB_L1_CHBL` = 0), access to this page bypasses the cache. The memory pages may not be defined as cacheable, when:

- An I/O device is mapped to the memory address.
- The code residing in the page is in-frequently called (the user may not want it to be cached).
- The code is extremely non-linear.

LRU Priority

This attribute (CPLB_LRUPRIO) is available on instruction memory CPLBs only. It defines LRU priority (low/high) for the given page. This is used for the modified LRU policy (see Block Placement on page 4 for further details).

Dirty/Modified Flag

The CPLB_DIRTY bit gives the programmer the choice for signaling that a (first) write access to external memory has occurred.

Present on data memory only, this attribute is valid only when the page is defined as cacheable in write-back mode. This bit should be set by software prior to store accesses to this page.

When this bit is cleared, an access to the page causes an exception (EXCAUSE 0x23). The exception routine must be set the bit to mark the page as dirty.

Write Access Permission Flags

Data memory CPLBs feature two flags that enable/disable write accesses to the corresponding page for supervisor mode (CPLB_SUPV_WR) and user mode (CPLB_USER_WR), individually.

User Read Access Permission Flag

This attribute (CPLB_USER_RD) enables/disables reads from this page in user mode.

Lock Flag

When the CPLB_LOCK bit is set, the corresponding CPLB entry is locked. This attribute is useful for dynamic memory management. When a CPLB entry is locked, the exception handler for CPLB miss will not consider it for replacement.

The page attributes related to “read/write permission” deal with the memory protection. It may be required in a real-time application in which the entire application is partitioned between OS code and user code. The user code may have different threads, with each thread having its own memory resources, which are not accessible to the other threads. However, the OS kernel can access all the memory resources. This task can be achieved by having different CPLB configurations in different threads.

When the CPLBs are enabled, a valid CPLB entry should exist in the CPLB table for every address to which an access is made. If there is no valid CPLB entry for the referenced address, a CPLB exception is generated.

CPLB Status Registers

The MPCU features two independent status registers: one for the ICPLB status (Figure 17), and one for the DCPLB status (Figure 18).

FAULT_ILLADDR

An access to memory that does not exist was attempted.

FAULT_DAG

This bit indicates that DAG0 or DAG1 caused the fault data access (DCPLB only).

FAULT_USERSUPV

Access was done in either user mode or supervisor mode.

FAULT_RW

Data access was either a read or a write access (DCPLB only).

FAULT

Hit/miss status of the associated CPLB entry.

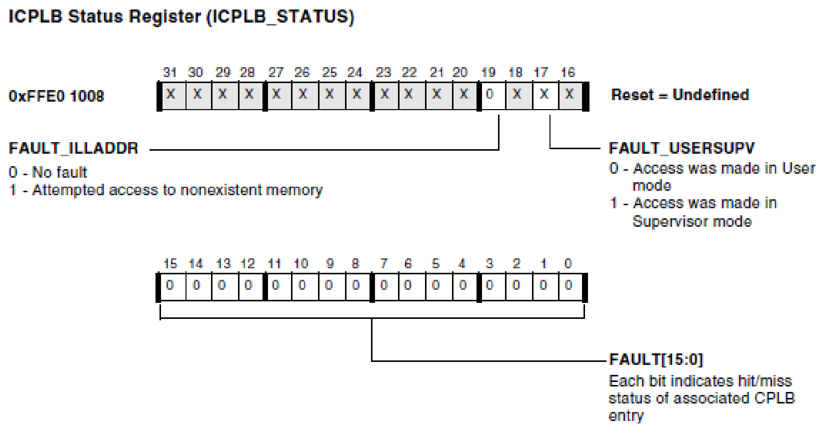

ICPLB Status Register (ICPLB_STATUS)
0xFFE0 1008
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
X X X X X X X X X X X X 0 X X X
Reset = Undefined
FAULT_ILLADDR
0 - No fault
1 - Attempted access to nonexistent memory
FAULT_USERSUPV
0 - Access was made in User mode
1 - Access was made in Supervisor mode
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
FAULT[15:0]
Each bit indicates hit/miss status of associated CPLB entry

Figure 17. Bit fields and their functionalities for the ICPLB status register

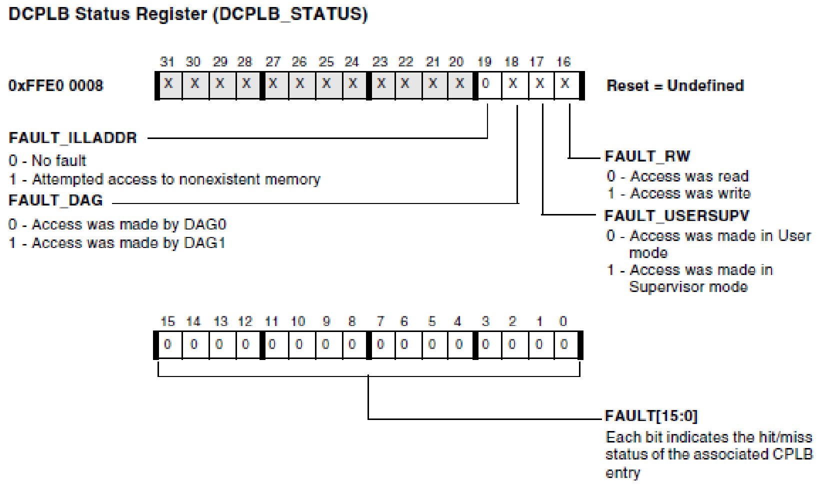

DCPLB Status Register (DCPLB_STATUS)
0xFFE0 0008
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
X X X X X X X X X X X X 0 X X X
Reset = Undefined
FAULT_ILLADDR
0 - No fault
1 - Attempted access to nonexistent memory
FAULT_DAG
0 - Access was made by DAG0
1 - Access was made by DAG1
FAULT_RW
0 - Access was read
1 - Access was write
FAULT_USERSUPV
0 - Access was made in User mode
1 - Access was made in Supervisor mode
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
FAULT[15:0]
Each bit indicates the hit/miss status of the associated CPLB entry

Figure 18. Bit fields and their functionalities for the DCPLB status register

Exception	**EXCAUSE [5:0]**	**Notes/Examples**
Data access CPLB protection violation	0x23	Attempted read/write to Supervisor resource (see [3]), or illegal data memory access. This entry is used to signal a protection violation caused by disallowed memory access, and it is defined by the Memory Protection and Cache Unit (MPCU) cacheability protection lookaside buffer (CPLB).
Data access misaligned address violation	0x24	Attempted misaligned data memory or data cache access.
Data access CPLB miss	0x26	Used by the MPCU to signal a CPLB miss on a data access.
Data access multiple CPLB hits	0x27	More than one CPLB entry matches data fetch address.
Instruction fetch CPLB protection violation	0x2B	Illegal instruction fetch access (memory protection violation).
Instruction fetch CPLB miss	0x2C	CPLB miss on an instruction fetch.
Instruction fetch multiple CPLB hits	0x2D	More than one CPLB entry matches instruction fetch address.

Table 7. CPLB events that cause exceptions [3]

CPLB-Related Sequencer Exceptions

In the `ICPLB_DATAx` and `DCPLB_DATAx` registers, some access policies (read/write in user mode and supervisor mode) can be defined for specific memory areas. Violating the permission rules cause sequencer exceptions.

The reason for an exception can be read from the `EXCAUSE` bit field of the sequencer status register (`SEQSTAT`). Table 7 shows an extract of all Exceptions that can be originated by the CPLBs.

Page Descriptor Table

Generally, a memory-based data structure called a *page descriptor table* is used for CPLB management. All the potentially required CPLB entries can be stored in the page descriptor table (which is generally located in the internal SRAM). When the CPLBs need to be configured, the application code can pick up the CPLB entries from page descriptor table and fill them into the CPLB descriptor registers.

For small/simple memory models it may be possible to define a set of CPLB descriptors that fit into 32 CPLB entries (16 ICPLBs and 16 DCPLBs). This type of definition is referred to as a *static memory management model*. “Example1” [(1)] uses a static memory management model.

For complex memory models, the page descriptor table may have CPLB entries that do not fit into the available 16 CPLB registers. Under such conditions, the CPLBs can be configured first with any 16 entries. When the processor references a memory location, which is not defined by the CPLBs, an exception is generated and the address of faulting memory location is stored in the Fault Address register (`xCPLB_FAULT_ADDR`). The exception handler can use the faulting address to search for the required entry in the CPLB table. One of the existing CPLB entries can be replaced by this new CPLB entry.

The CPLB replacement policy can be simple or complex, depending upon the system requirement. It is possible that more than one memory reference are made to the addresses for which there are no valid entry in the CPLB descriptors. Under such a condition, the exceptions are prioritized and serviced in the following order:

1. Instruction page misses
2. Page misses on DAG0
3. Page misses on DAG1

The code in “Example2” [(2)] provides an exception handler for DCPLB miss (see Table 7: `EXCAUSE 0x26`). It uses a round-robin scheduling method for the DCPLB replacement.

Coherency Considerations

If an outside source (e.g., DMA controller) is accessing external memory that is defined as cacheable, the programmer must ensure memory coherency. The cache controller is not aware of any changes that are not done by the MPCU. Simple memory polling will not work.

An example for such a situation might be a circular data buffer that is stored in external memory. Data are transferred between buffer and a peripheral interface (e.g. audio stream). The core has to do some calculations and must write the data back to the buffer where the data are transferred back to the peripheral interface. In this case, a write-through strategy is preferable.

Cache and Instruction Pipeline

Instr Fetch 1	Instr Fetch 2	Instr Fetch 3	Instr Dec	Addr Calc	Data Fetch 1	Data Fetch 2	Ex 1	Ex 2	WB

Table 8. Stages of the instruction pipeline

Table 8 shows the stages of the Blackfin processor’s instruction pipeline.

In stage 1 (IF1), an instruction address is issued to the Instruction Access Bus (IAB). In this phase, the comparison of the instruction cache tags is started.

In stage 6 (DF1), a data address is issued to the Data Access Buses (DA0 and DA1). In this phase, the comparison of the data cache tags is started.

For more details on the instruction pipeline, refer to the "Program Sequencer" chapter in the *Blackfin Processor Programming Reference* [3].

Handling the Instruction and Data Cache

Enabling the Cache

The instruction and data caches can be enabled or disabled independently by configuring the `IMEM_CONTROL` and `DMEM_CONTROL` registers appropriately. The example demonstrates how the data/instruction caches can be enabled.

- Before enabling the cache, valid CPLB descriptors must be configured and enabled.
- When the memory is configured as cache, it cannot be accessed directly (neither through core, nor through DMA).

Instruction Memory Control Register (IMEM_CONTROL)

Figure 19 depicts various bit-fields and their functionality in the `IMEM_CONTROL` register.

LRU Priority Reset

The `LRUPRIORST` bit can be used to reset all cached LRU priority bits.

Instruction Cache Locking by Way

The instruction cache has four independent lock bits (these bits are available in the Instruction Memory Control register), which can be used to lock any of the four ways independently.

When a particular way is locked (the corresponding `ILOC` bit in the instruction memory control register is set), it does not participate in the line replacement. The cached instructions from a locked way can be removed only with an `IFLUSH` instruction.

Cache locking only prevents valid cache-lines from being selected for replacement. Invalid cache-lines stored in a locked way can still be selected for replacement. This means a cache miss to an invalid entry will cause that entry to be replaced with the new cache-line.

"Example3" [(3)] (see associated `ZIP` file) demonstrates how the more frequently used functions (in the external memory) can be cached and locked such that they would not be replaced. The scheme consists of locking `Way1`, `Way2`, `Way3` (`Way0` unlocked), and making a dummy call to the functions of interest. The functions will be cached to `Way0` (as all other ways are locked). Now, `Way0` can be locked (and `Way1`, `Way2`, `Way3` can be unlocked). Any subsequent cache miss can replace lines in `Ways 1-3` (`Way0` is locked) only.

Locking all four ways at the same time is not recommended.

L1 Instruction Memory Configuration

The `IMC` bit controls if the upper 16K-byte bank of the instruction memory is configured as cache. The `ENICPLB` bit must be 1 as well if cache support is enabled.

Instruction CPLB Enable

With the `ENICPLB` bit, Instruction CPLBs can be enabled/disabled. Before loading new descriptor data into any CPLBs, the corresponding group of 16 CPLBs must be disabled using the `ENICPLB` bit.

Data Memory Control Register (DMEM_CONTROL)

Figure 20 depicts various bit-fields and their functionality in the `DMEM_CONTROL` register.

DAG Port Preference

With the two `PORT_PREFx` bits non-cacheable data fetches – originated by the *Data Address*

Generators (DAG0 and DAG1) – can be mapped to one specific DAG Port (*Port A* or *Port B*, see Figure 1).

L1 Data Cache Bank Select

The DCBS bit controls if address bit 14 or address bit 23 is used to switch between Bank A and Bank B for cache access (see Figure 20).

L1 Data Memory Configuration

The DMC bits enable/disable cache support for the L1 data memory banks. The DMC[1] bit controls Bank A, and DMC[0] controls Bank B. Configuring Bank A or Bank A + Bank B as cache is supported. Bank B cannot be configured as the only available L1 data cache memory. The ENDCPLB bit must be 1 as well if cache support is enabled.

Some Blackfin derivatives do not have two data banks and hence do not support the DCBS and the DMC[0] bit (see Table 2).

Data CPLB Enable

With the ENDCPLB bit, data CPLBs can be enabled/disabled. Before loading new descriptor data into any CPLBs, the corresponding group of 16 CPLBs must be disabled using the ENDCPLB bit.

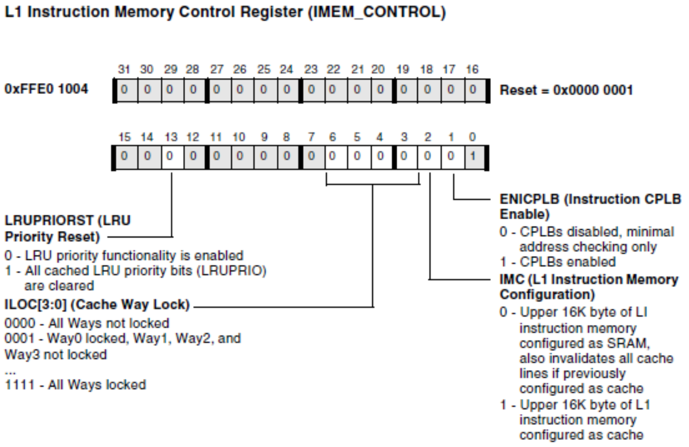

L1 Instruction Memory Control Register (IMEM_CONTROL)
0xFFE0 1004
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000 0001
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
LRUPRIORST (LRU Priority Reset)
0 - LRU priority functionality is enabled
1 - All cached LRU priority bits (LRUPRIO) are cleared
ILOC[3:0] (Cache Way Lock)
0000 - All Ways not locked
0001 - Way0 locked, Way1, Way2, and Way3 not locked
...
1111 - All Ways locked
ENICPLB (Instruction CPLB Enable)
0 - CPLBs disabled, minimal address checking only
1 - CPLBs enabled
IMC (L1 Instruction Memory Configuration)
0 - Upper 16K byte of LI instruction memory configured as SRAM, also invalidates all cache lines if previously configured as cache
1 - Upper 16K byte of L1 instruction memory configured as cache

Figure 19. Bit fields and their functionalities for the IMEM_CONTROL register

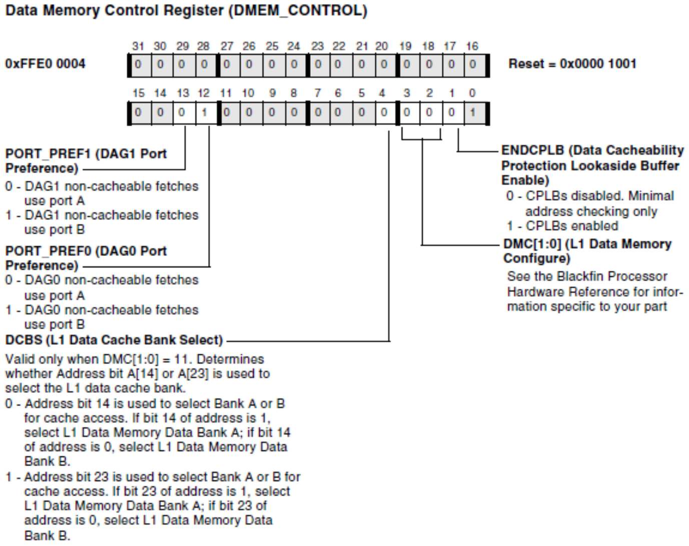

Data Memory Control Register (DMEM_CONTROL)
0xFFE0 0004
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Reset = 0x0000 1001
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 1
PORT_PREF1 (DAG1 Port Preference)
0 - DAG1 non-cacheable fetches use port A
1 - DAG1 non-cacheable fetches use port B
PORT_PREF0 (DAG0 Port Preference)
0 - DAG0 non-cacheable fetches use port A
1 - DAG0 non-cacheable fetches use port B
DCBS (L1 Data Cache Bank Select)
Valid only when DMC[1:0] = 11. Determines whether Address bit A[14] or A[23] is used to select the L1 data cache bank.
0 - Address bit 14 is used to select Bank A or B for cache access. If bit 14 of address is 1, select L1 Data Memory Data Bank A; if bit 14 of address is 0, select L1 Data Memory Data Bank B.
1 - Address bit 23 is used to select Bank A or B for cache access. If bit 23 of address is 1, select L1 Data Memory Data Bank A; if bit 23 of address is 0, select L1 Data Memory Data Bank B.
ENDCPLB (Data Cacheability Protection Lookaside Buffer Enable)
0 - CPLBs disabled. Minimal address checking only
1 - CPLBs enabled
DMC[1:0] (L1 Data Memory Configure)
See the Blackfin Processor Hardware Reference for information specific to your part

Figure 20. Bit fields and their functionalities for the DMEM_CONTROL register

Cache Control Instructions

Instruction Cache Invalidation

By invalidating the cache-lines associated with the buffer, “coherency” is maintained between the contents stored in cache and the actual values in source memory. There are three schemes for invalidating the instruction cache:

- The `IFLUSH` instruction can be used to invalidate a specific address in the memory map. When the instruction `IFLUSH [P2];` is executed, if the memory address pointed by `P2` has been brought in to the cache, the corresponding cache-line will be invalidated after execution of the above instruction. When the instruction is used like `IFLUSH [P2++];` the pointer increments by the size of a cache-line.
- The `VALID` bit of the tag section for any line in the cache can be cleared explicitly by writing a one to the tag section. The value to the tag section can be written using the `ITEST_COMMAND` register. This is discussed in detail in the next section.
- In order to invalidate the entire cache, the `IMC` bit in the `IMEM_CONTROL` register can be cleared. This clears the `VALID` bit for all tag sections in the cache. The `IMC` bit can be set to enable the cache again.

Data Cache Control Instructions

- The `PREFETCH` instruction can be used to allocate a line into the L1 cache.
- The `FLUSH` instruction causes the data cache to synchronize the specified cache-line with the external memory. If the cached data line is dirty, the instruction writes the line back to external memory and marks the line clean in the data cache.
- The `FLUSHINV` instruction causes the data cache to perform the same function as the `FLUSH` instruction and then invalidates the specified line in the cache. If the location is not dirty, no flush will occur. In this case, only the invalidate step takes place.

Accessing the Cache Memory

When configured as cache, the L1 memory bank cannot be accessed directly by the core or the DMA. Read/write operations can be performed onto the cache space using the `ITEST_COMMAND` and `DTEST_COMMAND` registers. The `DTEST_COMMAND` register can also be used to access the instruction SRAM banks. Figure 21 shows the bit fields for the `ITEST_COMMAND` register. Figure 22 shows bit fields for the `DTEST_COMMAND` register.

Accessing the Instruction Cache

The `ITEST_COMMAND` register can be used to access the data or tag sections of the instruction cache-lines.

A cache-line is divided in to four 64-bit words. Any of the four words can be selected for access. While reading the cache, the data value is read into the `ITEST_DATA[1:0]` register set. While writing to the cache, the value from the `ITEST_DATA[1:0]` register set are written to the cache.

When a tag section is being accessed, the 32-bit tag value is transferred to/from the `ITEST_DATA0` register.

Consider an example where value `0x0C010360` is written in to the `ITEST_COMMAND` register. This instruction will read the tag section from the way-3, set-27, subbank-1 and transfer it to the `ITEST_DATA0` register. Similarly, writing `0x0C010362` transfers the contents of `ITEST_DATA0` register to the tag section of the line located in way-3, set-27, and subbank-1. While accessing the tag section (read or write), bits 3 and 4 of the `ITEST_COMMAND` register are reserved.

Writing a value of `0x0C010374` will read the second word of the cache-line located at way-3, set-27, subbank-1 and transfer it to the `ITEST_DATA[1:0]` register set. Writing a value of 0x0C010366 to the ITEST_COMMAND register transfers the value of the ITEST_DATA[1:0] register set to word-0 of the cache-line located at way-3, set-27, subbank-1 of the instruction cache. While writing to the cache, the ITEST_DATA[1:0] register test must be loaded before the ITEST_COMMAND register is written.

Accessing the Instruction SRAM

When bit 24 of the DTEST_COMMAND register is set, the DTEST_COMMAND register can be used to access the instruction SRAM. A 64-bit word can be transferred to/from the DTEST_DATA[1:0] register set to/from the instruction SRAM. Thus, memory can be accessed eight bytes at a time.

Bit 2 of the DTEST_COMMAND register must be set while working in this mode. Bits 3-10 must be assigned with bits 3-10 of the address being accessed. Consider a case where the byte from address 0xFFA07935 has to be read from the instruction memory. This address lies in bank-1. While accessing the above byte an entire line addressed by (0xFFA07930 – 0xFFA07937) will be accessed. The control value that must be loaded to the DTEST_COMMAND register will be 0x05034134.

Accessing the Data Cache

When bit 24 of the DTEST_COMMAND register is cleared, the DTEST_COMMAND register can be used to access the data or tag sections of data cache-lines. A word from the data section of cache-line can be transferred to/from the DTEST_DATA[1:0] register set.

While accessing the tag section, the 32-bit tag value is transferred to/from the DTEST_DATA0 register.

Consider a case where the values of the DTEST_DATA[1:0] register set needs to be written to word-0 of the data line in way-1, set-39, subbank-0, data cache bank-A. Then the DTEST_COMMAND register can be written with a value of 0x040004E5. The DTEST_DATA[1:0] register set must be loaded before writing to the DTEST_COMMAND register. Bit 14 of the DTEST_COMMAND register is reserved while accessing the data cache space (bit 24 = 0).

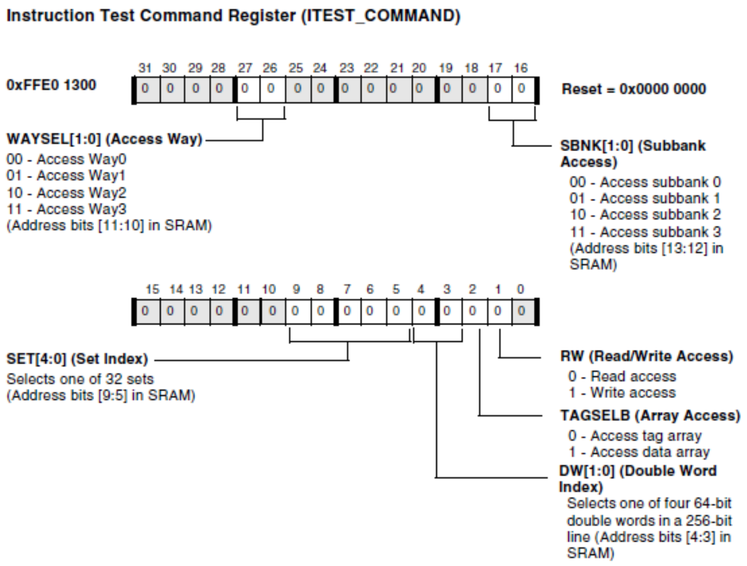


Figure 21. Bit fields and their functionalities for the ITEST_COMMAND register

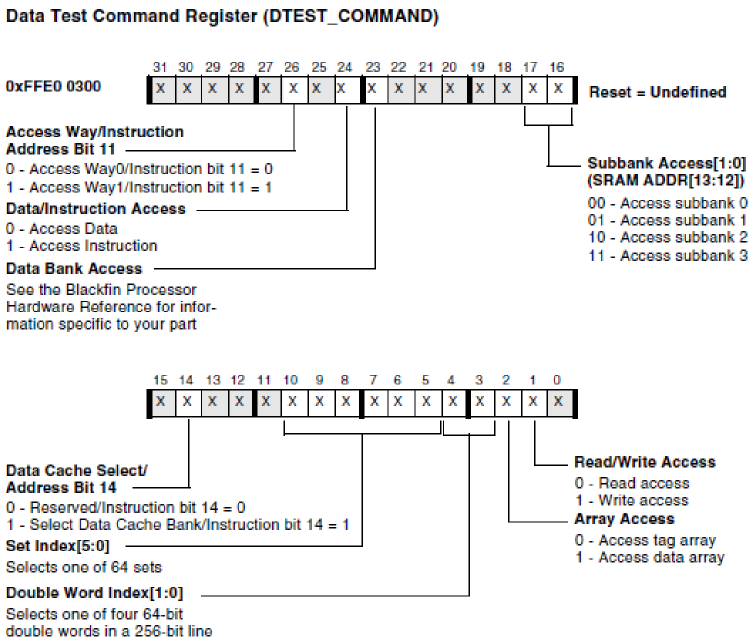

Data Test Command Register (DTEST_COMMAND)
0xFFE0 0300
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
X X X X X X X X X X X X X X X X
Reset = Undefined
Access Way/Instruction Address Bit 11
0 - Access Way0/Instruction bit 11 = 0
1 - Access Way1/Instruction bit 11 = 1
Data/Instruction Access
0 - Access Data
1 - Access Instruction
Data Bank Access
See the Blackfin Processor Hardware Reference for information specific to your part
Subbank Access[1:0] (SRAM ADDR[13:12])
00 - Access subbank 0
01 - Access subbank 1
10 - Access subbank 2
11 - Access subbank 3
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
X X X X X X X X X X X X X X X X
Data Cache Select/ Address Bit 14
0 - Reserved/Instruction bit 14 = 0
1 - Select Data Cache Bank/Instruction bit 14 = 1
Set Index[5:0]
Selects one of 64 sets
Double Word Index[1:0]
Selects one of four 64-bit double words in a 256-bit line
Read/Write Access
0 - Read access
1 - Write access
Array Access
0 - Access tag array
1 - Access data array

Figure 22. Bit fields and their functionalities for the DTEST_COMMAND register

Performance Considerations

The following section provides comparison figures with and without using the cache. These examples apply to special scenarios and provide ideas of how to obtain the best performance for a specific framework.

For these particular examples, the PLL is configured for a 270 MHz core clock speed (`CCLK`) and a 54 MHz system clock speed (`SCLK`), respectively (`CLKIN` = 27 MHz) on the ADSP-BF533 EZ-KIT Lite® evaluation board.

Instruction Cache: Optimized Conditions

For a linear code execution from external memory, the performance can be increased by up to 13% (compare to Instruction Fetch Latency: Cache vs. no Cache on page 10) when code is not stored in cache memory. If the instructions are already copied to cache and executed at least two times, greater than 50% performance enhancement is possible. Table 9 shows a comparison (in core clock cycles), when functions are executed 1x to 1000x (e.g., filter algorithm) and instruction cache is turned off and on, respectively. The numbers – counted in core clock cycles – are derived from "Example6" [6]. The "Gain" column shows how many cycles are saved (in percent). The numbers can vary up to ± 100 cycles when performing several test runs (e.g., refreshing SDRAM).

Exec #	ICache OFF	ICache ON	Gain
1x	1430	1245	12.9%
2x	2480	1329	46.4%
4x	4580	1469	67.9%
8x	8740	1973	77.4%
10x	10814	2173	79.9%
100x	104620	11850	88.7%
1000x	1042950	109050	89.5%

Table 9. Instruction cache performance with good conditions

Instruction Cache: Bad Concept

Table 9 shows similar numbers similar to the previous example. But now the code is no longer linear (each function contains just a jump instruction) and many cache-misses / cache-line replacements must be done. Each cache miss causes a cache-line fill operation. If the function just contains a jump instruction, the advantage of having code already available in cache when needed (linear code flow) is gone, and the additional fetched code was a waste of time.

Exec #	ICache OFF	ICache ON	Loss
1x	1031	1164	-12.9%
2x	1924	2394	-24.4%
4x	3747	4654	-24.2%
8x	7462	8918	-19.5%
10x	9314	11100	-19.2%
100x	91643	109201	-19.2%
1000x	915488	1090127	-19.1%

Table 10. Instruction cache performance with wrong concept

Data Cache

Data Cache Settings	Mem Write	Mem Read
No Data Cache	23039	55248
WB	10572	3866
WT without AOW bit set	23039	10583
WT with AOW bit set	32154	3866

Table 11. Data cache performance

The page size is set to 1KB. The core is first writing 15K bytes to external memory and then reading the data back; 32-bit accesses are performed to the same internal bank. With this setup, no CPLB replacement will be triggered. The DCBS bit is not relevant in this case, only L1 Data Bank A will be used for cache and all data can be stored there without a need for cache data replacement.

A CPLB replacement can cost several hundred cycles (exception handling, replacement algorithm, etc.). A good strategy for the CPLB table settings is important.

Table 11 shows the results represented by core clock cycles:

- A memory write without cache enabled and write-through cache (allocate cache-line on reads only) makes no difference.
- A memory write operation with write-back strategy, additional cache-line fills require more cycles.
- A memory write with the AOW bit set, write-through cache has worst write performance when accessing external memory for the first time (all cache-lines invalid).
- A memory read in write-back mode and write-through mode with the AOW bit set gives best performance as data are already stored in cache memory.
- A memory read with write-through cache with the AOW bit not set benefits from the cache-line burst fill.
- The memory read performance without data cache enabled is poor compared to the rest.

Core Clock vs. System Clock

An inappropriate ratio between CCLK and SCLK can cause a penalty. Figure 23 shows an instruction fetch starting from address 0x00h with instruction cache turned on.

CCLK:SCLK = 3:1 or higher should be the preferred settings when cache is in use. This requires 22 SCLK cycles.

The ratio between core clock and system clock is 2:1. For every second word, an additional SCLK cycle is inserted. About 28 SCLK cycles are required to fetch the complete cache-line (including activation command).

A ratio of 1:1 lowers the throughput. In this case, two additional SCLK cycles are inserted with each second word (see Figure 24). About 36 SCLK cycles are required.

This differences show the time for the core required to compare cache tags, arrange instruction in the alignment unit, and finally execute them.

Refreshing SDRAM Memory

Figure 25 shows a situation where a cache-line fill is interrupted by a memory refresh. The transfer stops and the memory controller (*auto-refresh* mode) issues a `precharge` (close row/page) command and a `refresh` command. When finished, the transfer continues with the next increment of the last address.

When the SDRAM is in self-refresh mode, the memory can still be accessed by the processor. In this case, the SDC releases the *self-refresh mode* to either *temporary auto-refresh* or *auto-refresh mode*, depending on the state of the `SRFS` bit in the `EBIU_SDGCTL` register.

SDRAM Performance

More information on SDRAM performance with the Blackfin SDC can be found in the "External Bus Interface Unit" chapter of the *ADSP-BF537 Blackfin Processor Hardware Reference* [5]. A DAG read/write access is 8/1 `SCLK` cycles per 16-bit word, respectively. Instruction fetches and cache-line fills require about 1.1 `SCLK` cycles per 16-bit word.

Conclusions

ⓘ Code must be used in a linear way to utilize the advantage of a cache-line burst fill (pre-fetch).

ⓘ Code must be re-used to make cache attractive.

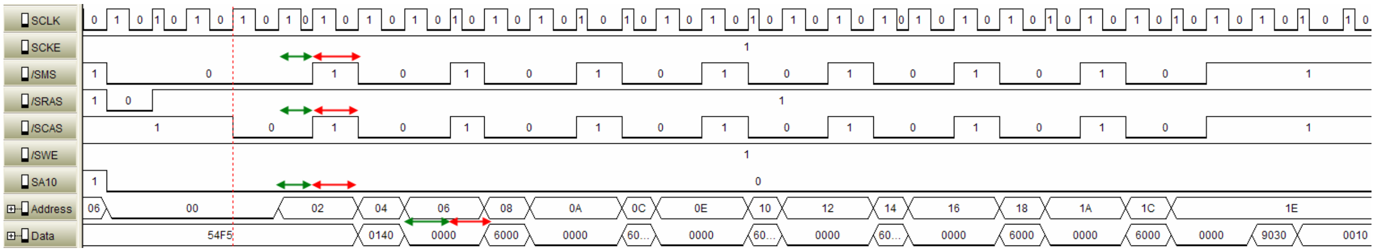


Figure 23. Instruction fetch with cache on (CCLK:SCLK = 2:1)

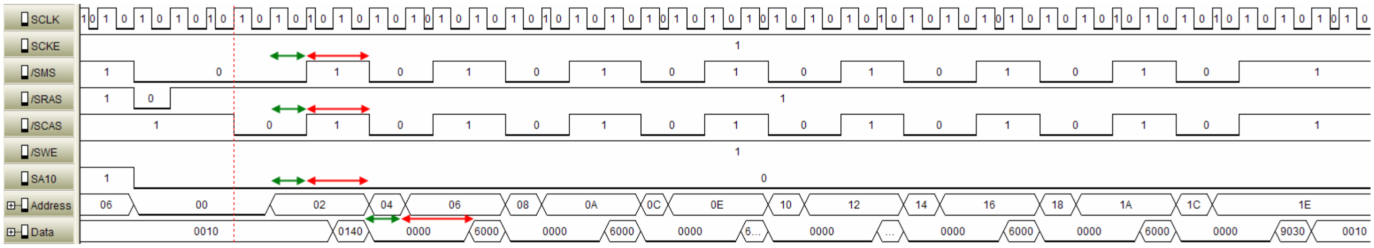


Figure 24. Instruction fetch with cache on (CCLK:SCLK = 1:1)

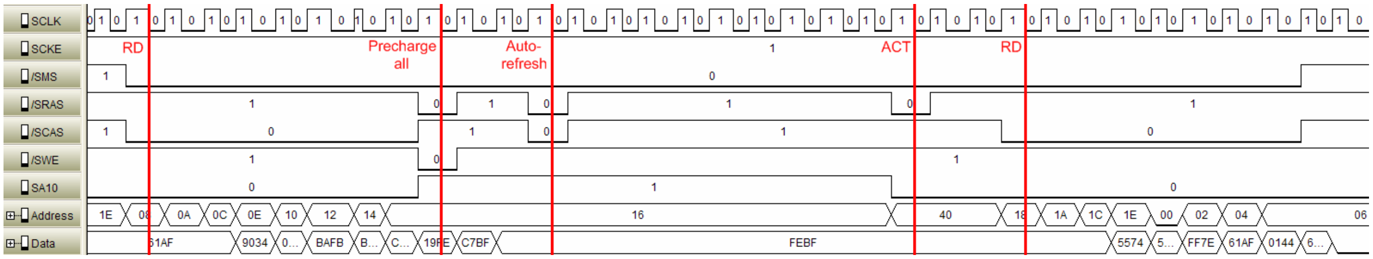


Figure 25. Instruction fetch with cache on (interrupted by auto-refresh)

VisualDSP++® Compiler Support

The VisualDSP++® tools support cache memory management. Some features are discussed below. Detailed information can be found in the *VisualDSP++ C/C++ Compiler and Library Manual for Blackfin Processors* [4]. Instruction and data caches can be enabled together or separately, and the memory spaces they cache are configured individually.

Header Files

The VisualDSP++ header files `def_LPBlackfin.h` and `cplb.h` offer user-friendly pre-defined macros and masks for `CPLB` configuration and more. The value `0x0C010366` from one of the previous examples would look like:

```
TEST_WAY3|TEST_MB1|TEST_SET(27)|TEST_D
W0|TEST_DATA|TEST_WRITE
```

CPLB Control

CPLB support is controlled through a global integer variable, `___cplb_ctrl`. Its C name has two leading underscore characters, and its assembler name has three underscore characters. The value of this variable determines whether the startup code enables the CPLB system. By default, the variable has the value `zero`, indicating that CPLBs should not be enabled. The pragma `retain_name` should be used with `__cplb_ctrl`, such that this variable is not eliminated by the compiler when optimization is enabled.

The value of `___cplb_ctrl` may be changed in the following ways:

- The variable may be defined as a new global with an initialization value. This definition supersedes the definition in the library.
- The linked-in version of the variable may be altered in a debugger, after loading the application but before running it, so that the startup code sees a different value.

When enabling caches using `___cplb_ctrl`, it is imperative that `USE_CACHE` also be specified.

CPLB Installation

When `___cplb_ctrl` indicates that CPLBs are to be enabled, the startup code calls the routine `_cplb_init`. This routine sets up instruction and data CPLBs from a table, and enables the memory-protection hardware. The default configuration tables are defined in files called `cplbtab`*n*`.s` in `VisualDSP\Blackfin\lib\src\libc\crt`, where *n* is the part number of the Blackfin processor.

When the cache is enabled, the default CPLB configuration defined in the above file is installed. However, you can modify the given files to define your own CPLB configuration. The given file must be included in the project file in order for the changes to be effective. The project "Example5"(5) demonstrates how the CPBL configuration table can be modified.

Exception Handling

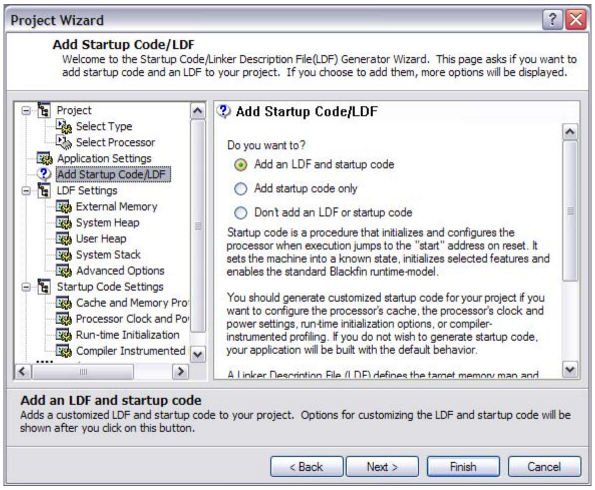
As discussed earlier, in a complex memory model there may need to be more CPLBs than can be active at once. In such systems, there will eventually come a time when the application attempts to access memory that is not covered by one of the active CPLBs. This will raise a CPLB miss exception.

The VisualDSP++ library includes a CPLB management routine for these occasions, called `_cplb_mgr`. This routine should be called from an exception handler that has determined that a CPLB miss has occurred, regardless whether it is a data miss or an instruction miss. `_cplb_mgr` identifies the inactive CPLB that must be installed to resolve the access, and replaces one of the active CPLBs with this one. If CPLBs are to be enabled, the default startup code installs a default exception handler, called `_cplb_hdr`; this does nothing other than to test for CPLB miss exceptions, which it delegates to `_cplb_mgr`. It is expected that users will have their own exception handlers to deal with additional events.

Using the Project Wizard

The Project Wizard offers the possibility to create a project that includes support for memory protection and cache for both instruction and data memory. Figure 26 through Figure 30 show the additional actions required:

1. `.LDF` file and startup code must be added (Figure 26).



Project Wizard

Add Startup Code/LDF
Welcome to the Startup Code/Linker Description File(LDF) Generator Wizard. This page asks if you want to add startup code and an LDF to your project. If you choose to add them, more options will be displayed.

Add Startup Code/LDF

Do you want to?

● Add an LDF and startup code
○ Add startup code only
○ Don't add an LDF or startup code

Startup code is a procedure that initializes and configures the processor when execution jumps to the "start" address on reset. It sets the machine into a known state, initializes selected features and enables the standard Blackfin runtime-model.

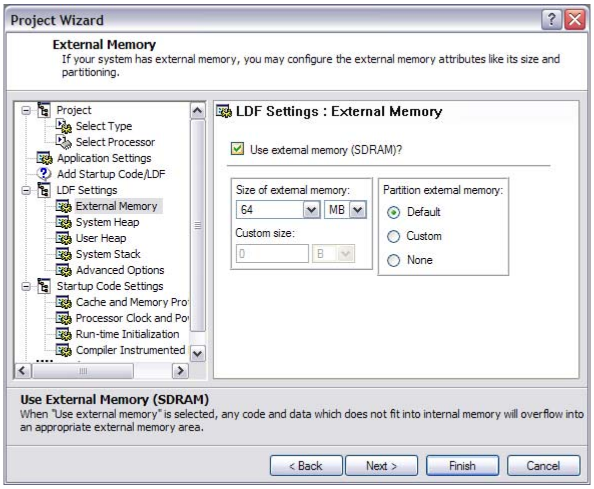
You should generate customized startup code for your project if you want to configure the processor's cache, the processor's clock and power settings, run-time initialization options, or compiler-instrumented profiling. If you do not wish to generate startup code, your application will be built with the default behavior.

A Linker Description File (LDF) defines the target memory map and

Add an LDF and startup code
Adds a customized LDF and startup code to your project. Options for customizing the LDF and startup code will be shown after you click on this button.

Figure 26. Project Wizard: Add Startup Code/LDF

2. Support for external memory (SDRAM) and the memory size must be added (Figure 27).



Project Wizard

External Memory
If your system has external memory, you may configure the external memory attributes like its size and partitioning.

LDF Settings : External Memory

☑ Use external memory (SDRAM)?

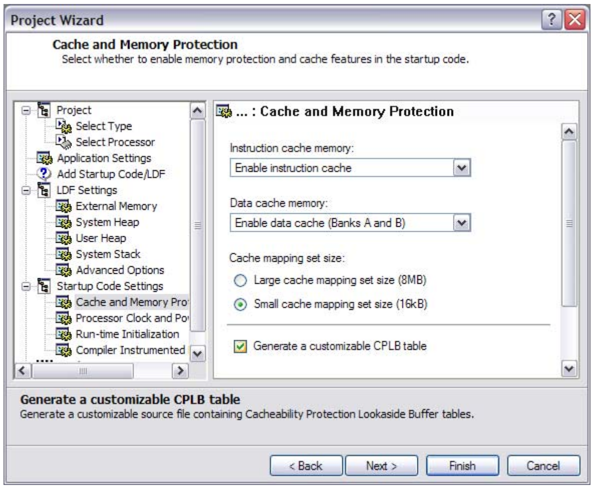
Size of external memory: 64 MB
Custom size: 0 B

Partition external memory:
● Default
○ Custom
○ None

Use External Memory (SDRAM)
When "Use external memory" is selected, any code and data which does not fit into internal memory will overflow into an appropriate external memory area.

Figure 27. Project Wizard: LDF Settings: External Memory

3. In the "Cache and Memory Protection" settings, the support for memory protection and cache can be added for both instruction and data memory (Figure 28). Customizable CPLB tables can be generated, and you can choose between write-through/write-back data cache. The "Cache mapping set size" option controls the `DCBS` bit in the `DMEM_CONTROL` register.



Project Wizard

Cache and Memory Protection
Select whether to enable memory protection and cache features in the startup code.

... : Cache and Memory Protection

Instruction cache memory: Enable instruction cache

Data cache memory: Enable data cache (Banks A and B)

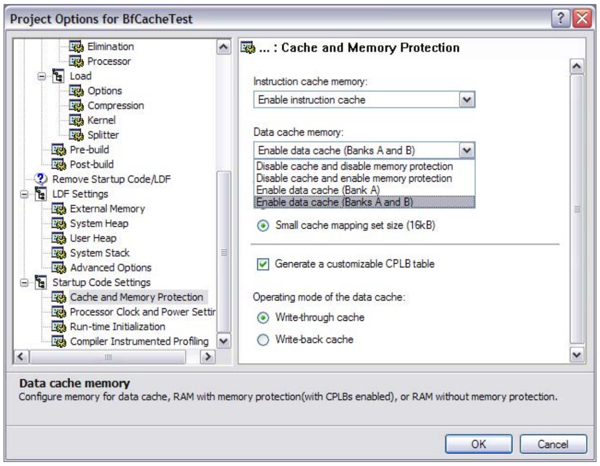
Cache mapping set size:
○ Large cache mapping set size (8MB)
● Small cache mapping set size (16kB)

☑ Generate a customizable CPLB table

Generate a customizable CPLB table
Generate a customizable source file containing Cacheability Protection Lookaside Buffer tables.

Figure 28. Project Wizard: Startup Code Settings: Cache and Memory Protection

4. All settings can be modified in the Project Options dialog box. The files will then be re-created (Figure 29).



Project Options for BfCacheTest

... : Cache and Memory Protection

Instruction cache memory: Enable instruction cache

Data cache memory: Enable data cache (Banks A and B)
Disable cache and disable memory protection
Disable cache and enable memory protection
Enable data cache (Bank A)
Enable data cache (Banks A and B)

● Small cache mapping set size (16kB)

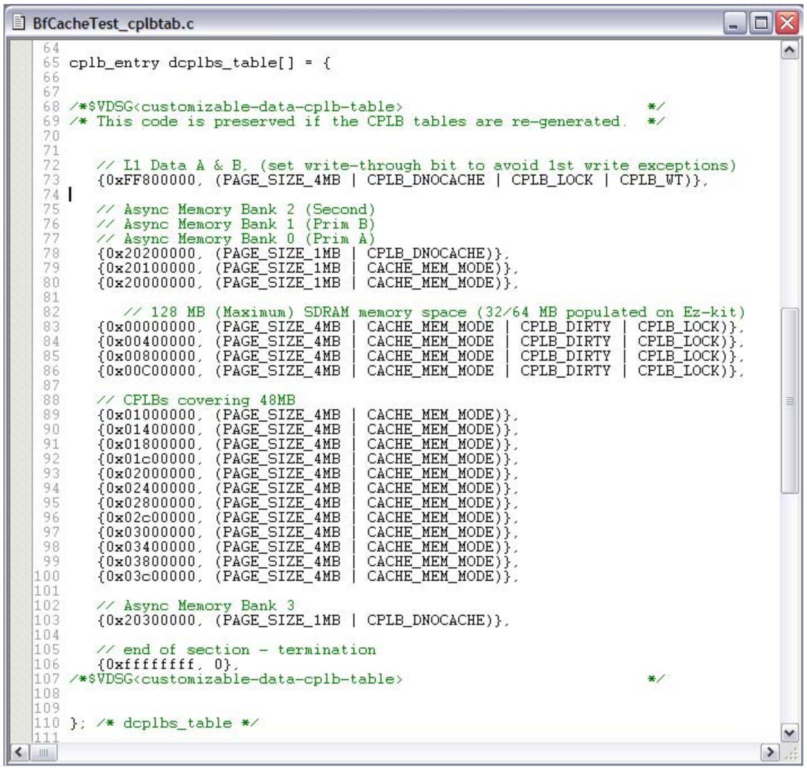
☑ Generate a customizable CPLB table

Operating mode of the data cache:
● Write-through cache
○ Write-back cache

Data cache memory
Configure memory for data cache, RAM with memory protection(with CPLBs enabled), or RAM without memory protection.

Figure 29. Project Options: Startup Code Settings: Cache and Memory Protection

When finished, a project will be generated including a file called *<Project_Name>*`_cplbtab.c` (Generated Files->Startup). This file contains the configurable CPLB table (Figure 30).



```
cplb_entry dcplbs_table[] = {


/*$VDSG<customizable-data-cplb-table>                           */
/* This code is preserved if the CPLB tables are re-generated.  */


    // L1 Data A & B, (set write-through bit to avoid 1st write exceptions)
    {0xFF800000, (PAGE_SIZE_4MB | CPLB_DNOCACHE | CPLB_LOCK | CPLB_WT)},

    // Async Memory Bank 2 (Second)
    // Async Memory Bank 1 (Prim B)
    // Async Memory Bank 0 (Prim A)
    {0x20200000, (PAGE_SIZE_1MB | CPLB_DNOCACHE)},
    {0x20100000, (PAGE_SIZE_1MB | CACHE_MEM_MODE)},
    {0x20000000, (PAGE_SIZE_1MB | CACHE_MEM_MODE)},

        // 128 MB (Maximum) SDRAM memory space (32/64 MB populated on Ez-kit)
    {0x00000000, (PAGE_SIZE_4MB | CACHE_MEM_MODE | CPLB_DIRTY | CPLB_LOCK)},
    {0x00400000, (PAGE_SIZE_4MB | CACHE_MEM_MODE | CPLB_DIRTY | CPLB_LOCK)},
    {0x00800000, (PAGE_SIZE_4MB | CACHE_MEM_MODE | CPLB_DIRTY | CPLB_LOCK)},
    {0x00C00000, (PAGE_SIZE_4MB | CACHE_MEM_MODE | CPLB_DIRTY | CPLB_LOCK)},

    // CPLBs covering 48MB
    {0x01000000, (PAGE_SIZE_4MB | CACHE_MEM_MODE)},
    {0x01400000, (PAGE_SIZE_4MB | CACHE_MEM_MODE)},
    {0x01800000, (PAGE_SIZE_4MB | CACHE_MEM_MODE)},
    {0x01c00000, (PAGE_SIZE_4MB | CACHE_MEM_MODE)},
    {0x02000000, (PAGE_SIZE_4MB | CACHE_MEM_MODE)},
    {0x02400000, (PAGE_SIZE_4MB | CACHE_MEM_MODE)},
    {0x02800000, (PAGE_SIZE_4MB | CACHE_MEM_MODE)},
    {0x02c00000, (PAGE_SIZE_4MB | CACHE_MEM_MODE)},
    {0x03000000, (PAGE_SIZE_4MB | CACHE_MEM_MODE)},
    {0x03400000, (PAGE_SIZE_4MB | CACHE_MEM_MODE)},
    {0x03800000, (PAGE_SIZE_4MB | CACHE_MEM_MODE)},
    {0x03c00000, (PAGE_SIZE_4MB | CACHE_MEM_MODE)},

    // Async Memory Bank 3
    {0x20300000, (PAGE_SIZE_1MB | CPLB_DNOCACHE)},

    // end of section - termination
    {0xffffffff, 0},
/*$VDSG<customizable-data-cplb-table>                           */


}; /* dcplbs_table */
```

Figure 30. VisualDSP++: Generated CPLB table

Summary of MMRs

The following memory-map registers are used for the memory management on Blackfin processors:

IMEM_CONTROL	DMEM_CONTROL
ITEST_COMMAND	DTEST_COMMAND
ITEST_DATA [1:0]	DTEST_DATA [1:0]
ICPLB_DATA [15:0]	DCPLB_DATA [15:0]
ICPLB_ADDR [15:0]	DCPLB_ADDR [15:0]
ICPLB_STATUS	DCPLB_STATUS
ICPLB_FAULT_ADDR	DCPLB_FAULT_ADDR

Table 12. CPLB memory-mapped registers

Cache Configuration for Blackfin Derivatives

The preceding section discussed cache configuration and cache control on ADSP-BF533 processors. The same discussion also applies to ADSP-BF531 and ADSP-BF532 processors. The example code (1) through (5) can be used for ADSP-BF531 and ADSP-BF532 processors too. Example code (6) and (7) supplied with this application note can be used with all single-core Blackfin processors.

The amount of instruction memory configurable as cache is the same (16K bytes) on all currently available single-core Blackfin processors.

Most single-core Blackfin processors support two L1 data banks configurable as cache, 16K bytes each. The ADSP-BF531 processor has only one data bank available.

The amount of memory available on the ADSP-BF561 Blackfin dual-core processor as cache is double of that of ADSP-BF533 processor.

Conclusion

This document discusses the instruction and data cache configuration on Blackfin processors. The address mapping to the cache-lines is also discussed. The example code provided with this application note demonstrates how to set up

CPLB descriptors for instruction memory and data memory, how to enable/disable the instruction/data cache, and how to handle the CPLB exceptions and locking the instruction cache by way. Discussion on accessing the instruction/data cache by core through ITEST_COMMAND and DTEST_COMMAND is also included.

Appendix

A .ZIP file is associated with this document. It contains the following code examples:

(1) Example code for configuring the CPLB descriptors and instruction/data cache (C)

(2) Example code for CPLB exception handling (C)

(3) Example code for locking the instruction cache by way (C)

(4) Example codes demonstrating the data cache control instructions (C)

(5) Example code demonstrating VisualDSP++ compiler support for Blackfin cache (C)

(6) Example code for configuring the ICPLB descriptors and instruction cache (Blackfin assembly)

(7) Example code for configuring the DCPLB descriptors and data cache (Blackfin assembly)

(8) Example code for a bad instruction cache concept (Blackfin assembly)

References

[1] *ADSP-BF533 Blackfin Processor Hardware Reference.* Rev 3.4, April 2009. Analog Devices, Inc.

[2] *Computes Architecture A Quantitative Approach.* Second Edition, 2000 David A. Patterson and John L. Hennessy

[3] *Blackfin Processor Programming Reference.* Rev. 1.3, September 2008. Analog Devices, Inc.

[4] *VisualDSP++ C/C++ Compiler and Library Manual for Blackfin Processors*. Rev. 5.1, August 2008. Analog Devices, Inc.

[5] *ADSP-BF537 Blackfin Processor Hardware Reference.* Rev 3.1, March 2009. Analog Devices, Inc.

Table of Figures

Figure 1. ADSP-BF533 block diagram 1

Figure 2. Memory arranges as fixed-size blocks 4

Figure 3. Configuration for a 2-way set associative cache memory 4

Figure 4. Block placement 5

Figure 5. Instruction cache configuration for one subbank – 4-way set associative 9

Figure 6. Instruction fetch with cache off 11

Figure 7. Instruction fetch with cache on 11

Figure 8. Latency with instruction cache off 11

Figure 9. Latency with instruction cache on 11

Figure 10. Data cache configuration for one subbank – 2-way set associative 13

Figure 11. Data cache mapping according to DCBS bit 13

Figure 12. Data fetch with cache off 14

Figure 13. Data fetch with cache on 14

Figure 14. Data fetch with cache on in write-through mode and AOW enabled 15
Figure 15. Bit fields and their functionalities for the ICPLB_DATA registers 16
Figure 16. Bit fields and their functionalities for the DCPLB_DATA registers 17
Figure 17. Bit fields and their functionalities for the ICPLB status register 19
Figure 18. Bit fields and their functionalities for the DCPLB status register 19
Figure 19. Bit fields and their functionalities for the IMEM_CONTROL register 22
Figure 20. Bit fields and their functionalities for the DMEM_CONTROL register 22
Figure 21. Bit fields and their functionalities for the ITEST_COMMAND register 24
Figure 22. Bit fields and their functionalities for the DTEST_COMMAND register 25
Figure 23. Instruction fetch with cache on (CCLK:SCLK = 2:1) 27
Figure 24. Instruction fetch with cache on (CCLK:SCLK = 1:1) 27
Figure 25. Instruction fetch with cache on (interrupted by auto-refresh) 27
Figure 26. Project Wizard: Add Startup Code/LDF 29
Figure 27. Project Wizard: LDF Settings: External Memory 29
Figure 28. Project Wizard: Startup Code Settings: Cache and Memory Protection 29
Figure 29. Project Options: Startup Code Settings: Cache and Memory Protection 29
Figure 30. VisualDSP++: Generated CPLB table 30

List of Tables

Table 1. Address partitioning 5
Table 2. Blackfin processors memory map 7
Table 3. Line replacement policy for the Blackfin instruction cache 9
Table 4. Instruction cache-line word fetching order 10
Table 5. Data cache-line word fetching order 12
Table 6. Behavior in specific write situation 14
Table 7. CPLB events that cause exceptions [3] 19
Table 8. Stages of the instruction pipeline 20
Table 9. Instruction cache performance with good conditions 25
Table 10. Instruction cache performance with wrong concept 26
Table 11. Data cache performance 26
Table 12. CPLB memory-mapped registers 30

Document History

Revision	**Description**
Rev 2 – May 13, 2009 *by Andreas Pellkofer*	Added new figures along with their descriptions. Example codes rebuild for VisualDSP++ Release 5.0 (tested with Update 5). Also, two new assembly example codes added to associated .ZIP file.
Rev 1 – June 13, 2005 *by Kunal Singh*	Initial release.