Application Note, V 1.1, June 2004

AP32067

TriCore® 1 Guidelines for Cache Management

TC1M

Microcontrollers

Never stop thinking

Revision History: **2004-06** V 1.1

Previous Version:-

Page	Subjects (major changes since last revision)
	Minor changes

We Listen to Your Comments

Any information within this document that you feel is wrong, unclear or missing at all?
Your feedback will help us to continuously improve the quality of this document.
Please send your proposal (including a reference to this document) to:

ipdoc@infineon.com

1 Architecture

1.1 Introduction

This application note introduces the TriCore® 1 TC1M cache architecture and organization, and provides guidelines for managing caches in TC1M applications in terms of cache control, coherency and alignment.

1.2 Overview

TC1M implements on-chip Level-1 Harvard Architecture cache. This means that the instruction cache (I-cache) and data cache (D-cache) are separated.

I-cache is located in the on-chip Program Memory Unit (PMU) while D-cache is located in the on-chip Data Memory Unit (DMU). The off-chip main memory (external to CPU, PMU and DMU) are physically mapped into these L1 caches.

Figure1 below shows a simplified diagram of the cache architecture:

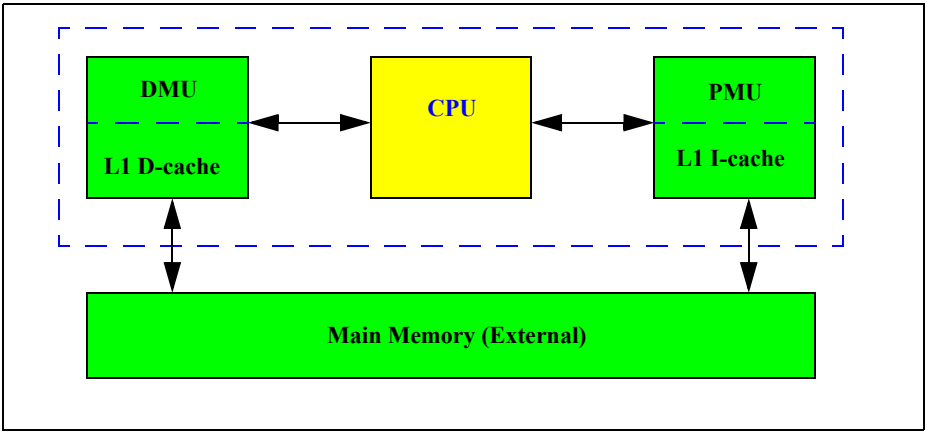

Figure 1 TC1M Cache Architecture Diagram (Simplified)

TC1M employs set-associative caches with LRU (Least Recently Used) replacement algorithms. TC1M also provides cache coherency support to both I-cache and D-cache through special cache control and instructions, ensuring a competitive design with high performance/price ratio.

1.3 I-Cache Organization

I-Cache is a two-way set-associative cache with 256-bits per line. A full cache line can hold a minimum of eight 32-bit instructions and a maximum of sixteen 16-bit instructions.

TC1M I-Cache size is fixed in silicon and can be 4K, 8K or 16K bytes in the PMU, dependent on the implementation in specific SOC (System-On-a-Chip) designs.

1.4 D-Cache Organization

D-cache is also a two-way set-associative cache with 128-bits per line.

D-cache size is fixed in silicon and can be 0k, 4K, 8K or 16K bytes in the DMU, dependent on the implementation in specific SOC (System-On-a-Chip) designs.

1.5 Memory Cacheability

TC1M can address up to 4Gbytes of space that is divided into 16 segments (256Mbytes each). Two segments, 8 and 9, are directly mapped into caches.

The MMU spaces, from segment 0 to 7, are also physically mapped into caches.

Memory cacheability is controllable in TC1M.

2 Controlling Caches

In TC1M the implementation limits cache control capability. Both I-cache and D-cache sizes are fixed in silicon but programmers are allowed to enable or disable (bypass) I-cache but not D-cache. I-cache is disabled (bypassed) by default after power reset.

To enable I-Cache, the bit **CCBYP** in **PMU_CON0** register should be set to 0. This should be set during boot up.

Note: It is strongly recommended to avoid toggling the bit CCBYP in PMU_CON0 after boot up (i.e. during runtime).

The pseudo code for cache control in TC1M would be:

```
ICacheEnable();                 // Clear CCBYP bit
ENDINIT_Protect();              // Set ENDINIT bit
```

2.1 Handling Cacheability With MMU

Programmers can take cache advantages by locating program into segments 8 and 9, as well as MMU (Memory Management Unit) space (segment 0 to 7), as long as TC1M works in **physical mode**. It should be noted however that a minor bug in TC1M restricts the program cacheability of MMU segments. Those segments are always cached when TC1M works in physical mode.

When working in **virtual mode** (by setting the **V** bit in **MMU_CON** register to 1), the upper 2GBytes of virtual memory (with virtual address A31 equals to 1) will be cacheable. However, the cacheability of the lower 2GBytes of virtual memory (with address A31 equals to 0) is controlled by the bit **C** in **MMU_TPA** register. To be cacheable, the bit C should be set to 1, otherwise it should be set to 0.

By following the guidelines for controlling cache that have been discussed above, the cacheable virtual memory can then be made cached.

2.2 Handling Cacheability With EBU

The TC1M EBU (External Bus Unit) mirroring feature allows segments to be mapped into other segments. The cacheable segment 8 or 9 can be mapped into non-cacheable segment 10 or 11 (or vice-versa), by setting the **ALTSEG** bits in address select register **ADDSELx** to an alternate segment value.

Note: Setting the ALTSEG bits in address select register ADDSELx to an alternate segment value should only be carried out during boot up and it is NOT recommended to modify this register "on-the-fly".

By mirroring program (code and data) located in cached segments (8 or 9) to non-cached segments (10 or 11), the code and data will not be cached with I-cache and D-cache. This "cache bypassing" mechanism is useful for applications where large shared data buffers (located

originally in cached segments) are used, to avoid handling complicated cache coherency between D-cache and shared memory.

On the contrary, the code and data located in non-cached segments but mirrored into cached segments will be cached with I-cache and D-cache. This mechanism is very useful for applications where slower flash memory (usually located in non-cached segments) is used for accommodating boot code, to speed up the boot up sequence by taking advantage of cache features.

2.3 Maintaining D-Cache Coherency

TC1M does not support hardware coherency for data cache but it does provide software support for data cache coherency. There are three special instructions for cache coherency implemented in TC1M:

- Cache write-through instruction CACHEA.W
  Forces modified data in a cache line to be written back to main memory. The data in the cache line is still valid and accessing this data will generate a cache hit.
- Cache invalid instruction CACHEA.I
  Invalidates an individual cache line. A subsequent load or store operation to an address mapping to that cache line will result in a cache miss, and the associated data will be loaded into that cache line.
- Cache write-through and invalid instruction CACHEA.WI
  This first forces the modified data in a cache line to be written back to memory and then invalidates that cache line.

TC1M uses a write-back data cache, which means that CPU data changes in a cache line will not be written into main memory until this cache line is to be replaced. This technique avoids frequent writing cycles to the slower main memory (compared to cache memory speed), therefore reducing bus traffic. A possible drawback however, is that it may raise cache coherency problems in some instances, in power down mode and in multiprocessor systems for example.

2.3.1 D-Cache Coherency In Power-down Mode

Many SOC (System-On-a-Chip) designs integrated with TC1M, feature a power-down mode that removes power supply for the TC1M CPU, PMU and DMU, including caches, for the sake of power saving. This feature is commonly seen in hand-held applications.

To maintain cache coherency between D-cache and main memory, the modified cache lines in D-cache should be written back to main memory before entering power down mode, as the cache content might be lost due to cache power missing. The cache write-through instruction CACHEA.W can be used to perform this task. However, this instruction is only able to process one cache line corresponding to a given address and is not able to handle all the modified cache

lines corresponding to various memory locations. Therefore a solution to write-through the whole D-cache is required.

The solution presented in this application note is based on the idea that the cache line that is least recently used will be evicted from the cache and written back into main memory. Loading a block of memory with size equal to the size of D-cache but with different content, will be enough to force out the whole D-cache data and be written back to main memory.

To effectively load a memory block, a context load instruction that loads a 64 byte memory block in a batch to the general purpose registers is used (A2~7, D0~7 for lower context, A10~15, D8~15 for upper context). The context load instruction can be either lower or upper context. To ensure the content of the memory block is different from the D-cache, a read-only space such as boot ROM that locates at any cached segment can be loaded. A zero-overhead loop instruction LOOP should be used together with a LDLCX or LDUCX instruction to speed up the process. The loop iteration number can be calculated as per the following formula:

```
Loop Iteration Number = (D-cache Size / 64) -1
```

The assembly code listed below illustrates how to flush (write-through) the whole D-cache effectively (using upper context load instruction):

```
LD.A A2, #BlockMemoryAddress ;Macro to load 32bit address to A2
LEA A3, #LoopIterationNumber
loop:
LDUCX [A2]
LOOP A3, loop
```

2.3.2 D-Cache Coherency In Multi-Processor Systems

TC1M architecture implements a proprietary Flexible Peripheral Interconnect (FPI) bus, which allows processors to be connected together to form a multi-processor system. **Figure 2** below shows a multi-processor system where the main memory is shared by two processors:

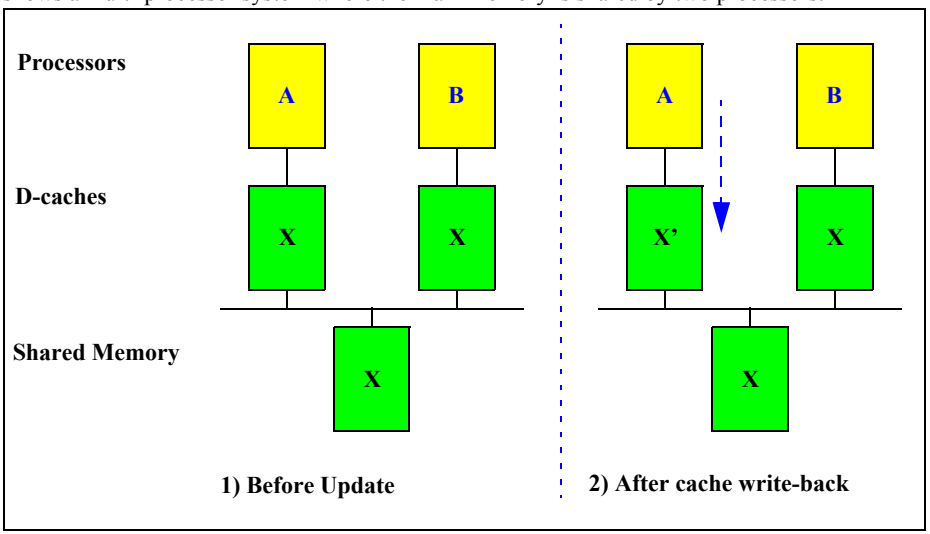

Figure 2 Data Cache Incoherency In Multiprocessor System

The cache content changed by processor A (from X to X') has not been written into the shared memory, while processor B is still using the old data in it's D-cache (X). This results in cache incoherency.

To avoid such a problem, a cache coherency protocol is needed to ensure that the content of the shared memory is consistent with the content in each processor's data cache. With the TC1M cache write-through and invalidate instructions it is possible to apply a software based cache coherency mechanism for a multiprocessor system.

Figure 3 (which follows) illustrates a very simple idea to implement the cache coherency between processor A and B. When cache A writes-back a cache line, a cache write-through instruction is used to force processor A to immediately update the shared memory content with the content of that line. Simultaneously, processor A informs processor B of the address information of changed data through a semaphore. Processor B then invalidates that cache line where the changed data sits by using a cache invalidate instruction. A cache miss will occur whenever processor B accesses that changed data, which forces processor B to load the cache line, including updated data content shared memory, into data cache B. The content in shared memory is then consistent with the content in both data cache A and B.

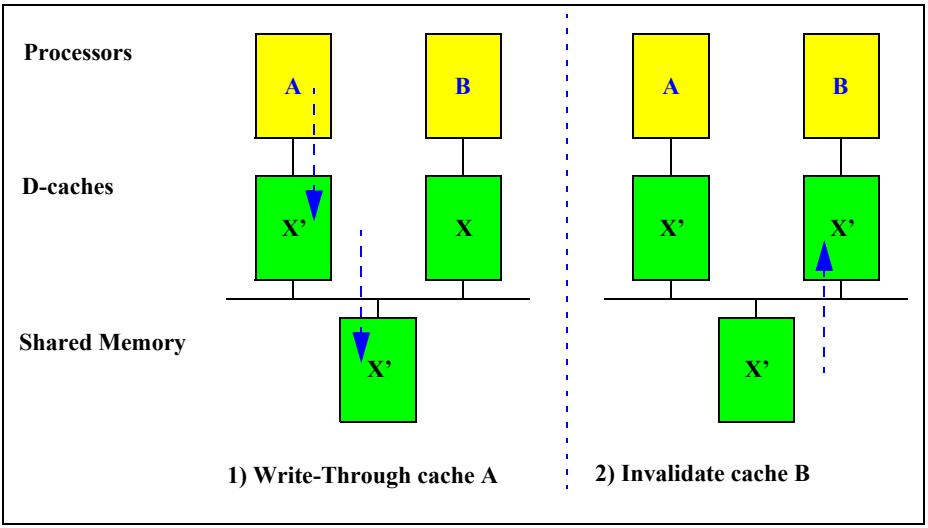

Figure 3 Maintaining Data Cache Coherency In A Multiprocessor System

The pseudo code for the implementation of Figure 3 is shown below. Please note that a **DSYNC** instruction is required after each cache write-through instruction or cache invalidate instruction to ensure the operation completed before proceeding to the next instruction.

```
Processor A:
If (Processor A modifies the data in a cache line) {
CACHE.W #AddressOfChangedData;
DSYNC;
SemaphoreToProcessorB();
}

Processor B:
If (SemaphoreFromProcessorA() ){
CACHE.I #AddressOfChangedData;
DSYNC;
}
```

In many TC1M applications, some master devices which do not have their own caches may also be connected around the FPI bus. These could be Co-Processors, PCI controllers or DMA devices for example. These devices could also share data memory with the TC1M processor. In this instance (in which only one data cache exists), it is also necessary to maintain data coherency.

To keep data consistency between data cache (on the process side) and shared memory, the processor should immediately write-through a cache line into the shared memory when it modifies any data in that cache line. Alternatively, when other FPI connected devices modify the

data in shared memory, they should inform the processor immediately with data address information via a semaphore, and the processor should invalidate the cache line upon getting the semaphore. A cache miss to that cache line will force the CPU (processor side) to load that data from the shared memory except for data cache, so cache coherency is maintained.

2.4 Maintaing I-Cache Coherency

TC1M provides code cache coherency support through an I-cache invalidation function. The I-cache content can be globally invalidated by setting the invalid control bit **CCINV** (bit 0) in **PMU_CON1** register. When this bit is set, the tag RAM in the cache will be reset and all the cache lines will be invalidated. After that, all cache accesses will be treated as cache miss and the I-cache will be updated with the content in main memory.

I-cache coherency is required when the code in main memory that has been mapped into the I-cache changes. In some applications new code could be loaded into main memory by DMA devices during runtime for example. To ensure code consistency between I-cache and main memory, the **CCINV** bit should be set to invalidate the whole I-cache so that the changed code will be loaded into I-cache (due to cache misses afterwards). In this instance it is necessary to clear the **CCINV** bit immediately after setting this bit, to ensure that I-cache still takes effect. The pseudo code for this method would be:

```
pmu = ReadRegister32(PMU_CON1);
pmu |= 0x1; // set bit CCINV to invalidate I-cache
WriteRegister32(PMU_CON1, pmu);
pmu &= ~0x1; //clear bit CCINV immediately (to validate I-cache)
WriteRegister32(PMU_CON1, pmu);
```

3 Memory Alignment Constraints

TC1M pipeline architecture requires memory alignment of both code and data. Failing to align code and data in memory may cause pipeline performance degradation, such as pipeline stalls for example. Memory alignment constraints also apply to caches.

3.1 I-Cache Alignment Constraint

The TC1M I-cache line is 256-bits wide and a minimum of 64-bit code will be loaded into a cache line. It is therefore preferred that the code in main memory mapped to I-cache lines is 256-bit alignment. There are four cases however that are not satisfied with this constraint:

- 128-bit aligned with address A0~3 equal to 0000 (excluding 256-bit alignment)
  Only 128-bits of code will be loaded into a cache line in this instance. Although this will waste a half of cache line space in the I-cache, there is no pipeline performance penalty.
- 64-bit aligned with address A0~2 equal to 000 (excluding 128-bit and 256-bit alignments)
  If the code is mapped into the last 64-bit of a cache line, only 64-bits of code will be loaded into a cache line. If the code is mapped into the 3rd 64-bit of the cache line then 192-bits of code will be loaded into a cache line. The wasted space will be ¾ or ¼ of a cache line respectively and there is no pipeline performance penalty.
- 32-bit aligned with address A0~1 equal to 00 (excluding 64-bit, 128-bit and 256-bit alignments).
  This is similar to the 64-bit aligned case; i.e. part of the cache line space will be wasted, but there is no pipeline performance penalty.
- 16-bit aligned with address A0 equals to 0 (excluding 32-bit, 64-bit, 128-bit and 256-bit alignments).
  This is the 'worst case'. A 32-bit wide instruction in main memory may be separately loaded into 2 neighboring cache lines, with each half (16-bit) located at one of two cache lines. This not only wastes cache line space but also causes performance penalties. When the CPU fetches a 32-bit instruction crossing two cache lines from I-cache, at least one cycle pipeline stall will be encountered.

To ensure code memory alignment to obtain the best cache performance, all subroutines should be 256-bit aligned. Unfortunately existing TC1M compilers do not support automatic subroutine code alignment. Programmers therefore have to do this manually by using #pragma option in their code. Below is an example to apply 256-bit alignment to a subroutine using the Tasking compiler:

```
#pragma align on
#pragma align 256
void function1(void)
{
```

```
…
}
#pragma align off
```

In practical applications it is only recommended to apply alignment to critical and/or frequently called routines, as this will avoid wasting memory space (lots of NOP instructions will probably be inserted to achieve alignment) but without sacrificing much cache performance.

3.2 D-Cache Alignment Constraint

TC1M's D-cache line is 128-bits wide. D-cache always loads 128-bits of data in main memory into a cache line upon a cache miss. Unlike I-cache, a 64-bit data alignment (with address A0~2 equal to 00) in main memory, is enough to ensure a smooth load/store operation on the pipeline.

- If the data is 16-bit aligned then the CPU may access a 32-bit or 64-bit data crossing 2 cache lines. This will cause a minimum one cycle pipeline stall.
- If the data is 32-bit aligned then the CPU may access a 64-bit data crossing 2 cache lines. This will cause a minimum one cycle pipeline stall.

Data alignment in main memory can be achieved by applying data allocation option in the linker/locator. It is recommended to allocate frequently used data and larger data sets into main memory where the address is at least 64-bit aligned.

4 References

- TriCore Architecture Manual V1.3
- TC1M Functional Description V1.0
- TC1M Architecture SFRs and Bits V1.0

Glossary

Reference	Description
Ax	Address Registers (x is a number)
CON	Control PMU_CON0 = Program Memory Unit_Control register 0
CPU	Central Processing Unit
DMA	Direct Memory Address
DMU	Data Memory Unit
Dx	Data Registers (x is a number)
EBU	External Bus Unit
FPI	Flexible Peripheral Interface
L2	Level 2 Cache
LRU	Least Recently Used
MMU	Memory Management Unit
PCI	Peripheral Component Interconnect
PMU	Program Memory Unit
SOC	System-On-a-Chip
TAG RAM	Random Access Memory Tag The area in a Level 2 cache that identifies which data from main memory is currently stored in each cache line.
TC	TriCore TC1M = TriCore 1 Modular
TPA	Translation Physical Address MMU_TPA = Memory Management Unit_Translation Physical Address

http://www.infineon.com

Published by Infineon Technologies AG