9. Cache and Tightly-Coupled Memory

NII52007-10.1.0

Introduction

Nios® II processor cores can contain instruction and data caches. This chapter discusses cache-related issues that you need to consider to guarantee that your program executes correctly on the Nios II processor. Fortunately, most software based on the Nios II hardware abstraction layer (HAL) works correctly without any special accommodations for caches. However, some software must manage the cache directly. For code that needs direct control over the cache, the Nios II architecture provides facilities to perform the following actions:

- Initialize lines in the instruction and data caches
- Flush lines in the instruction and data caches
- Bypass the data cache during load and store instructions

This chapter discusses the following common cases in which you must manage the cache:

- Initializing cache after reset
- Writing device drivers
- Writing program loaders or self-modifying code
- Managing cache in multi-master or multi-processor systems

This chapter contains the following sections:

Nios II Cache Implementation

Depending on the Nios II core implementation, a Nios II processor system might or might not have data or instruction caches. You can write programs generically so that they function correctly on any Nios II processor, regardless of whether it has cache memory. For a Nios II core without one or both caches, cache management operations are benign and have no effect.

The current Nios II cores have no hardware cache coherency mechanism. Therefore, if multiple masters can access shared memory, software must explicitly maintain coherency across all masters.

For complete details about the features of each Nios II core implementation, refer to the *Nios II Core Implementation Details* chapter of the *Nios II Processor Reference Handbook*.

The details for a particular Nios II processor system are defined in the **system.h** file. Example 9–1 shows an excerpt from the **system.h** file, defining the cache properties, such as cache size and the size of a single cache line.

Example 9–1. An Excerpt from system.h that Defines the Cache Structure

```
#define NIOS2_ICACHE_SIZE 4096
#define NIOS2_DCACHE_SIZE 0
#define NIOS2_ICACHE_LINE_SIZE 32
#define NIOS2_DCACHE_LINE_SIZE 0
```

This system has a 4-kilobyte (KB) instruction cache with 32 byte lines, and no data cache.

HAL API Functions for Managing Cache

The HAL application program interface (API) provides the following functions for managing cache memory:

- `alt_dcache_flush()`
- `alt_dcache_flush_all()`
- `alt_icache_flush()`
- `alt_icache_flush_all()`
- `alt_uncached_malloc()`
- `alt_uncached_free()`
- `alt_remap_uncached()`
- `alt_remap_cached()`

For details about API functions, refer to the *HAL API Reference* chapter of the *Nios II Software Developer's Handbook*.

Further Information

This chapter covers only cache management issues that affect Nios II programmers. It does not discuss the fundamental operation of caches. Refer to *The Cache Memory Book* by Jim Handy for a discussion of general cache management issues.

Initializing Cache after Reset

After reset, the contents of the instruction cache and data cache are unknown. They must be initialized at the start of the software reset handler for correct operation.

The Nios II caches cannot be disabled by software; they are always enabled. To allow proper operation, a processor reset causes the instruction cache to invalidate the one instruction cache line that corresponds to the reset handler address. This forces the instruction cache to fetch instructions corresponding to this cache line from memory. The reset handler address must be aligned to the size of the instruction cache line.

It is the responsibility of the first eight instructions of the reset handler to initialize the remainder of the instruction cache. The Nios II `initi` instruction initializes a single instruction cache line. Do not use the `flushi` instruction because it might cause undesired effects when used to initialize the instruction cache in future Nios II implementations.

Place the `initi` instruction in a loop that executes `initi` for each instruction cache line address. Example 9–2 shows an example of assembly code to initialize the instruction cache.

Example 9–2. Assembly Code to Initialize the Instruction Cache

```
    mov     r4, r0
    movhi   r5, %hi(NIOS2_ICACHE_SIZE)
    ori     r5, r5, %lo(NIOS2_ICACHE_SIZE)
icache_init_loop:
    initi   r4
    addi    r4, r4, NIOS2_ICACHE_LINE_SIZE
    bltu    r4, r5, icache_init_loop
```

After the instruction cache is initialized, the data cache must also be initialized. The Nios II `initd` instruction initializes a single data cache line. Do not use the `flushd` instruction for this purpose, because it writes dirty lines back to memory. The data cache is undefined after reset, including the cache line tags. Using `flushd` can cause unexpected writes of random data to random addresses. The `initd` instruction does not write back dirty data.

Place the `initd` instruction in a loop that executes `initd` for each data cache line address. Example 9–3 shows an example of assembly code to initialize the data cache:

Example 9–3. Assembly Code to Initialize the Data Cache

```
    mov     r4, r0
    movhi   r5, %hi(NIOS2_DCACHE_SIZE)
    ori     r5, r5, %lo(NIOS2_DCACHE_SIZE)
dcache_init_loop:
    initd   0(r4)
    addi    r4, r4, NIOS2_DCACHE_LINE_SIZE
    bltu    r4, r5, dcache_init_loop
```

It is legal to execute instruction and data cache initialization code on Nios II cores that do not implement one or both of the caches. The `initi` and `initd` instructions are simply treated as `nop` instructions if there is no cache of the corresponding type present.

For HAL Users

Programs based on the HAL need not manage the initialization of cache memory. The HAL C run-time code (`crt0.S`) provides a default reset handler that performs cache initialization before `alt_main()` or `main()` is called.

Writing Device Drivers

Device drivers typically access control registers associated with their device. These registers are mapped into the Nios II address space. When accessing device registers, the data cache must be bypassed to ensure that accesses are not lost or deferred due to the data cache.

When writing a device driver, bypass the data cache with the `ldio`/`stio` family of instructions. On Nios II cores without a data cache, these instructions behave just like their corresponding `ld`/`st` instructions, and therefore are benign.

☞ Declaring a C pointer `volatile` does not make pointer accesses bypass the data cache. The `volatile` keyword merely prevents the compiler from optimizing out accesses using the pointer. This `volatile` behavior is different from the methodology for the first-generation Nios processor.

For HAL Users

The HAL provides the C-language macros `IORD` and `IOWR` that expand to the appropriate assembly instructions to bypass the data cache. The `IORD` macro expands to the `ldwio` instruction, and the `IOWR` macro expands to the `stwio` instruction. These macros are provided to enable HAL device drivers to access device registers.

Table 9–1 shows the available macros. All of these macros bypass the data cache when they perform their operation. In general, your program passes values defined in **system.h** as the `BASE` and `REGNUM` parameters. These macros are defined in the file *<Nios II EDS install path>***/components/altera_nios2/HAL/inc/io.h**.

Table 9–1. HAL I/O Macros to Bypass the Data Cache

Macro	Use
`IORD(BASE, REGNUM)`	Read the value of the register at offset `REGNUM` in a device with base address `BASE`. Registers are assumed to be offset by the address width of the bus.
`IOWR(BASE, REGNUM, DATA)`	Write the value `DATA` to the register at offset `REGNUM` in a device with base address `BASE`. Registers are assumed to be offset by the address width of the bus.
`IORD_32DIRECT(BASE, OFFSET)`	Make a 32-bit read access at the location with address `BASE+OFFSET`.
`IORD_16DIRECT(BASE, OFFSET)`	Make a 16-bit read access at the location with address `BASE+OFFSET`.
`IORD_8DIRECT(BASE, OFFSET)`	Make an 8-bit read access at the location with address `BASE+OFFSET`.
`IOWR_32DIRECT(BASE, OFFSET, DATA)`	Make a 32-bit write access to write the value `DATA` at the location with address `BASE+OFFSET`.

Table 9–1. HAL I/O Macros to Bypass the Data Cache

Macro	Use
`IOWR_16DIRECT(BASE, OFFSET, DATA)`	Make a 16-bit write access to write the value `DATA` at the location with address `BASE+OFFSET`.
`IOWR_8DIRECT(BASE, OFFSET, DATA)`	Make an 8-bit write access to write the value `DATA` at the location with address `BASE+OFFSET`.

Writing Program Loaders or Self-Modifying Code

Software that writes instructions to memory, such as program loaders or self-modifying code, needs to ensure that old instructions are flushed from the instruction cache and processor pipeline. This flushing is accomplished with the `flushi` and `flushp` instructions, respectively. Additionally, if new instruction(s) are written to memory using store instructions that do not bypass the data cache, you must use the `flushd` instruction to flush the new instruction(s) from the data cache to memory.

Example 9–4 shows assembly code that writes a new instruction to memory.

Example 9–4. Assembly Code That Writes a New Instruction to Memory

```
/*
 * Assume new instruction in r4 and
 * instruction address already in r5.
 */
stw    r4, 0(r5)
flushd    0(r5)
flushi    r5
flushp
```

The `stw` instruction writes the new instruction in `r4` to the instruction address specified by `r5`. If a data cache is present, the instruction is written just to the data cache and the associated line is marked dirty. The `flushd` instruction writes the data cache line associated with the address in `r5` to memory and invalidates the corresponding data cache line. The `flushi` instruction invalidates the instruction cache line associated with the address in `r5`. Finally, the `flushp` instruction ensures that the processor pipeline has not prefetched the old instruction at the address specified by `r5`.

Notice that Example 9–4 uses the `stw/flushd` pair instead of the `stwio` instruction. The `stwio` instruction does not flush the data cache, and therefore might leave stale data in the data cache.

This code sequence is correct for all Nios II implementations. If a Nios II core does not have a particular kind of cache, the corresponding flush instruction (`flushd` or `flushi`) is executed as a `nop`.

For Users of the HAL

The HAL API does not provide functions for this cache management case.

Managing Cache in Multi-Master/ Multi-Processor Systems

The Nios II architecture does not provide hardware cache coherency. Instead, software cache coherency must be provided when communicating through shared memory. The data cache contents of all processors accessing the shared memory must be managed by software to ensure that all masters read the most recent values and do not overwrite new data with stale data. This management is done by using the data cache flushing and bypassing facilities to move data between the shared memory and the data cache(s) as needed.

The `flushd` instruction ensures that the data cache and memory contain the same value for one line. If the line contains dirty data, it is written to memory. The line is then invalidated in the data cache.

Consistently bypassing the data cache is very important. The processor does not check if an address is in the data cache when bypassing the data cache. If software cannot guarantee that a particular address is in the data cache, it must flush the address from the data cache before bypassing it for a load or store. This action guarantees that the processor does not bypass new (dirty) data in the cache, and mistakenly access old data in memory.

Bit-31 Cache Bypass

The `ldio`/`stio` family of instructions explicitly bypass the data cache. Bit-31 provides an alternate method to bypass the data cache. Using the bit-31 cache bypass, the normal `ld`/`st` family of instructions can be used to bypass the data cache if the most significant bit of the address (bit 31) is set to one. The value of bit 31 is only used internally to the processor; bit 31 is forced to zero in the actual address accessed. This limits the maximum byte address space to 31 bits.

Using bit 31 to bypass the data cache is a convenient mechanism for software because the cacheability of the associated address is contained in the address. This usage allows the address to be passed to code that uses the normal `ld`/`st` family of instructions, while still guaranteeing that all accesses to that address consistently bypass the data cache.

Bit-31 cache bypass is only provided in the Nios II/f core, and must not be used with other Nios II cores. The other Nios II cores limit their maximum byte address space to 31 bits to ease migration of code from one implementation to another. They effectively ignore the value of bit 31, which allows code written for a Nios II/f core using bit 31 cache bypass to run correctly on other current Nios II implementations. In general, this feature depends on the Nios II core implementation.

For details, refer to the *Nios II Core Implementation Details* chapter of the *Nios II Processor Reference Handbook*.

For HAL Users

The HAL provides the C-language `IORD_*DIRECT` macros that expand to the `ldio` family of instructions and the `IOWR_*DIRECT` macros that expand to the `stio` family of instructions. Refer to Table 9–1 on page 9–4. These macros are provided to access noncacheable memory regions.

The HAL provides the `alt_uncached_malloc()`, `alt_uncached_free()`, `alt_remap_uncached()`, and `alt_remap_cached()` routines to allocate and manipulate regions of uncached memory. These routines are available on Nios II cores with or without a data cache—code written for a Nios II core with a data cache is completely compatible with a Nios II core without a data cache.

The `alt_uncached_malloc()` and `alt_remap_uncached()` routines guarantee that the allocated memory region is not in the data cache and that all subsequent accesses to the allocated memory regions bypass the data cache.

Tightly-Coupled Memory

If you want the performance of cache all the time, place your code or data in a tightly-coupled memory. Tightly-coupled memory is fast on-chip memory that bypasses the cache and has guaranteed low latency. Tightly-coupled memory gives the best memory access performance. You assign code and data to tightly-coupled memory partitions in the same way as other memory sections.

Cache instructions do not affect tightly-coupled memory. However, cache-management instructions become NOPs, which might result in unnecessary overhead.

For more information, refer to “Memory Usage” in the *Developing Programs Using the Hardware Abstraction Layer* chapter of the *Nios II Software Developer’s Handbook*.

Document Revision History

Table 9–2 shows the revision history for this document.

Table 9–2. Document Revision History

Date	Version	Changes
February 2011	10.1.0	Removed “Referenced Documents” section.
July 2010	10.0.0	Maintenance release.
November 2009	9.1.0	Maintenance release.
March 2009	9.0.0	■ Reorganized and updated information and terminology to clarify role of Nios II Software Build Tools. ■ Corrected minor typographical errors.
May 2008	8.0.0	Maintenance release.
October 2007	7.2.0	Maintenance release.
May 2007	7.1.0	■ Added table of contents to “Introduction” section. ■ Added Referenced Documents section.
March 2007	7.0.0	Maintenance release.
November 2006	6.1.0	Maintenance release.
May 2006	6.0.0	Maintenance release.
October 2005	5.1.0	Added detail to section “Tightly-Coupled Memory”.
May 2005	5.0.0	Added tightly-coupled memory section.
May 2004	1.0	Initial release.