

## Introduction

Nios® II processor cores may 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 HAL system library 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 when you need to 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:

- [“Initializing Cache after Reset” on page 9-3](#)
- [“Writing Device Drivers” on page 9-4](#)
- [“Writing Program Loaders or Self-Modifying Code” on page 9-5](#)
- [“Managing Cache in Multi-Master /Multi-CPU Systems” on page 9-6](#)
- [“Tightly-Coupled Memory” on page 9-8](#)

## Nios II Cache Implementation

Depending on the Nios II core implementation, a Nios II processor system may or may 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.

In all current Nios II cores, there is no hardware cache coherency mechanism. Therefore, if there are multiple masters accessing shared memory, software must explicitly maintain coherency across all masters.



For complete details on the features of each Nios II core implementation, see 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. The following code 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: 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 Kbyte instruction cache with 32 byte lines, and no data cache.

## HAL API Functions for Managing Cache

The HAL 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 on API functions, see 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. *The Cache Memory Book* by Jim Handy is a good text that covers 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 is required to 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 `init_i` instruction is used to initialize one instruction cache line. Do not use the `flush_i` instruction because it may cause undesired effects when used to initialize the instruction cache in future Nios II implementations.

Place the `init_i` instruction in a loop that executes `init_i` for each instruction cache line address. The following code shows an example of assembly code to initialize the instruction cache.

### Example: 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:
init_i 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 `init_d` instruction is used to initialize one data cache line. Do not use the `flush_d` 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 `flush_d` can cause unexpected writes of random data to random addresses. The `init_d` instruction does not write back dirty data.

Place the `init_d` instruction in a loop that executes `init_d` for each data cache line address. The following code shows an example of assembly code to initialize the data cache:

### Example: 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:
init_d 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 don't implement one or both of the caches. The `init_i` and `init_d` instructions are simply treated as nop instructions if there is no cache of the corresponding type present.

### For HAL System Library Users

Programs based on the HAL do not have to 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()` are 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.

For device drivers, the data cache should be bypassed by using 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.

For C programmers, note that declaring a pointer as `volatile` does not cause accesses using that volatile pointer to bypass the data cache. The `volatile` keyword only 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 System Library 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 should be used by 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
<code>IORD(BASE, REGNUM)</code>	Read the value of the register at offset <code>REGNUM</code> within a device with base address <code>BASE</code> . Registers are assumed to be offset by the address width of the bus.
<code>IOWR(BASE, REGNUM, DATA)</code>	Write the value <code>DATA</code> to the register at offset <code>REGNUM</code> within a device with base address <code>BASE</code> . Registers are assumed to be offset by the address width of the bus.
<code>IORD_32DIRECT(BASE, OFFSET)</code>	Make a 32-bit read access at the location with address <code>BASE+OFFSET</code> .
<code>IORD_16DIRECT(BASE, OFFSET)</code>	Make a 16-bit read access at the location with address <code>BASE+OFFSET</code> .
<code>IORD_8DIRECT(BASE, OFFSET)</code>	Make an 8-bit read access at the location with address <code>BASE+OFFSET</code> .
<code>IOWR_32DIRECT(BASE, OFFSET, DATA)</code>	Make a 32-bit write access to write the value <code>DATA</code> at the location with address <code>BASE+OFFSET</code> .
<code>IOWR_16DIRECT(BASE, OFFSET, DATA)</code>	Make a 16-bit write access to write the value <code>DATA</code> at the location with address <code>BASE+OFFSET</code> .
<code>IOWR_8DIRECT(BASE, OFFSET, DATA)</code>	Make an 8-bit write access to write the value <code>DATA</code> at the location with address <code>BASE+OFFSET</code> .

## Writing Program Loaders or Self-Modifying Code

Software that writes instructions into memory, such as program loaders or self-modifying code, needs to ensure that old instructions are flushed from the instruction cache and CPU 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 into memory.

The following code shows assembly code that writes a new instruction to memory.

### Example: 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 CPU pipeline has not prefetched the old instruction at the address specified by `r5`.

Notice that the above code sequence used the `stw/flushd` pair instead of the `stwio` instruction. Using a `stwio` instruction doesn't flush the data cache so could leave stale data in the data cache.

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

### For Users of the HAL System Library

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

## Managing Cache in Multi-Master/Multi-CPU 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 is used to ensure 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 of utmost importance. 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 may 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 CPU; 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 within 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 explicitly provided in the Nios II/f core, and should not be used for other Nios II cores. The other Nios II cores that do not support bit-31 cache bypass 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 is dependent 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 System Library 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. See [Table 9-1](#). These macros are provided to access non-cacheable 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 isn't 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, put 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 the “Assigning Code and Data to Memory Partitions” section of the *Developing Programs using the HAL* chapter of the *Nios II Software Developer's Handbook*.

## Referenced Documents

This chapter references the following documents:

- *Nios II Core Implementation Details* chapter of the *Nios II Processor Reference Handbook*
- *HAL API Reference* chapter in the *Nios II Software Developer's Handbook*

## Document Revision History

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

<b>Date &amp; Document Version</b>	<b>Changes Made</b>	<b>Summary of Changes</b>
May 2008 v8.0.0	No change from previous release.	
October 2007 v7.2.0	No change from previous release.	
May 2007 v7.1.0	<ul style="list-style-type: none"> <li>● Chapter 8 was formerly chapter 7.</li> <li>● Added table of contents to Introduction section.</li> <li>● Added Referenced Documents section.</li> </ul>	
March 2007 v7.0.0	No change from previous release.	
November 2006 v6.1.0	No change from previous release.	
May 2006 v6.0.0	No change from previous release.	
October 2005 v5.1.0	Added detail to section "Tightly-Coupled Memory".	
May 2005 v5.0.0	Added tightly-coupled memory section.	
May 2004 v1.0	Initial Release.	

