3. Programming Model

NII51003-9.0.0

Introduction

This chapter describes the Nios® II programming model, covering processor features at the assembly language level. Fully understanding the contents of this chapter requires prior knowledge of computer architecture, operating systems, virtual memory and memory management, software processes and process management, exception handling, and instruction sets. This chapter assumes you have a detailed understanding of the aforementioned concepts and focuses on how these concepts are specifically implemented in the Nios II processor. Where possible, this chapter uses industry-standard terminology.

This chapter discusses the following topics from the system programmer's perspective:

- Operating modes, page 3–2—Defines the relationships between executable code and memory.
- Memory management unit (MMU), page 3–3—Describes virtual memory support for full-featured operating systems.
- Memory protection unit (MPU), page 3–8—Describes memory protection without virtual memory management.
- General-purpose registers, page 3–10—Describes the Nios II general register set.
- Control registers, page 3–11—Describes the Nios II control register set.
- Exception processing, page 3–25—Describes how the Nios II processor responds to exceptions.
- Processor reset state, page 3–27—Describes how the Nios II processor responds to a processor reset signal.
- Hardware-assisted debug processing, page 3–28—Describes how the Nios II processor responds to break exceptions.
- Hardware interrupts, page 3–29—Describes how the Nios II processor responds to hardware interrupts.
- Memory, cache memory, and peripheral organization, page 3–38—Describes how the Nios II processor interfaces with memory and peripherals.
- Instruction set categories, page 3–40—Introduces the Nios II instruction set.
- Custom instructions, page 3–44—Describes the scope of Nios II custom instructions.

Because of the flexibility and capability range of the Nios II processor, this chapter covers topics that support a variety of operating systems and runtime environments. While reading, be aware that all sections might not apply to you. For example, if you are using a minimal system runtime environment, you can skip the sections covering operating modes, the MMU, the MPU, or the control registers exclusively used by the MMU and MPU.

High-level software development tools are not discussed here. Refer to the *Nios II Software Developer's Handbook* for information about developing software.

Operating Modes

Operating modes control how the processor operates, manages system memory, and accesses peripherals. The Nios II architecture supports these operating modes:

- Supervisor mode
- User mode

The following sections define the modes, their relationship to your system software and application code, and their relationship to the Nios II MMU and Nios II MPU. Refer to "Memory Management Unit" on page 3–3 for more information about the Nios II MMU. Refer to "Memory Protection Unit" on page 3–8 for more information about the Nios II MPU.

Supervisor Mode

Supervisor mode allows unrestricted operation of the processor. All code has access to all processor instructions and resources. The processor may perform any operation the Nios II architecture provides. Any instruction may be executed, any I/O operation may be initiated, and any area of memory may be accessed.

Operating systems and other system software run in supervisor mode. In systems with an MMU, application code runs in user mode, and the operating system, running in supervisor mode, controls the application's access to memory and peripherals. In systems with an MPU, your system software controls the mode in which your application code runs. In Nios II systems without an MMU or MPU, all application and system code runs in supervisor mode.

Code that needs direct access to and control of the processor runs in supervisor mode. For example, the processor enters supervisor mode whenever a processor exception (including processor reset or break) occurs. Software debugging tools also use supervisor mode to implement features such as breakpoints and watchpoints.

For systems without an MMU or MPU, all code runs in supervisor mode.

User Mode

User mode is available only when the Nios II processor in your hardware design includes an MMU or MPU. User mode exists solely to support operating systems. Operating systems (that make use of the processor's user mode) run your application code in user mode. The user mode capabilities of the processor are a subset of the supervisor mode capabilities. Only a subset of the instruction set is available in user mode.

The operating system determines which memory addresses are accessible to user mode applications. Attempts by user mode applications to access memory locations without user access enabled are not permitted and cause an exception. Code running in user mode uses system calls to make requests to the operating system to perform I/O operations, manage memory, and access other system functionality in the supervisor memory.

The Nios II MMU statically divides the 32-bit virtual address space into user and supervisor partitions. Refer to "Address Space and Memory Partitions" on page 3–4 for more information about the MMU memory partitions. The MMU provides operating systems access permissions on a per-page basis. Refer to "Virtual Addressing" on page 3–3 for more information about MMU pages.

The Nios II MPU supervisor and user memory divisions are determined by the operating system or runtime environment. The MPU provides user access permissions on a region basis. Refer to "Memory Regions" on page 3–8 for more information about MPU regions.

Memory Management Unit

The Nios II processor provides an MMU to support full-featured operating systems. Operating systems that require virtual memory rely on an MMU to manage the virtual memory. When present, the MMU manages memory accesses including translation of virtual addresses to physical addresses, memory protection, cache control, and software process memory allocation.

Recommended Usage

Including the Nios II MMU in your Nios II hardware system is optional. The MMU is only useful with an operating system that takes advantage of it.

Many Nios II systems have simpler requirements where minimal system software or a small-footprint operating system (such as Altera® HAL or a third party real-time operating system) is sufficient. Such software is unlikely to function correctly in a hardware system with an MMU-based Nios II processor. Do not include an MMU in your Nios II system unless your operating system requires it.

☞ The Altera HAL and HAL-based real-time operating systems do not support the MMU.

If your system needs memory protection, but not virtual memory management, refer to "Memory Protection Unit" on page 3–8.

Memory Management

Memory management comprises two key functions:

- Virtual addressing—Mapping a virtual memory space into a physical memory space
- Memory protection—Allowing access only to certain memory under certain conditions

Virtual Addressing

A virtual address is the address that software uses. A physical address is the address which the hardware outputs on the address lines of the Avalon® bus. The Nios II MMU divides virtual memory into 4 KByte pages and physical memory into 4 KByte frames.

The MMU contains a hardware translation lookaside buffer (TLB). The operating system is responsible for creating and maintaining a page table (or equivalent data structures) in memory. The hardware TLB acts as a software managed cache for the page table. The MMU does not perform any operations on the page table, such as hardware table walks. Therefore the operating system is free to implement its page table in any appropriate manner.

Table 3–1 shows how the Nios II MMU divides up the virtual address. There is a 20 bit virtual page number (VPN) and a 12 bit page offset.

Table 3–1. MMU Virtual Address Fields

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Virtual Page Number																				Page Offset											

As input, the TLB takes a VPN plus a process identifier (to guarantee uniqueness). As output, the TLB provides the corresponding physical frame number (PFN).

Distinct processes can use the same virtual address space. The process identifier, concatenated with the virtual address, distinguishes identical virtual addresses in separate processes. To determine the physical address, the Nios II MMU translates a VPN to a PFN and then concatenates the PFN with the page offset. The bits in the page offset are not translated.

Memory Protection

The Nios II MMU maintains read, write, and execute permissions for each page. The TLB provides the permission information when translating a VPN. The operating system can control whether or not each process is allowed to read data from, write data to, or execute instructions on each particular page. The MMU also controls whether accesses to each data page are cacheable or uncacheable by default.

Whenever an instruction attempts to access a page that either has no TLB mapping, or lacks the appropriate permissions, the MMU generates a precise exception. Precise exceptions enable the system software to update the TLB, and then re-execute the instruction if desired.

Address Space and Memory Partitions

The MMU provides a 4 GByte virtual address space, and is capable of addressing up to 4 GBytes of physical memory.

☞ The amount of actual physical memory, determined by the configuration of your hardware system, might be less than the available 4 GBytes of physical address space.

Virtual Memory Address Space

The 4 GByte virtual memory space is divided into partitions. The upper 2 GBytes of memory is reserved for the operating system and the lower 2 GBytes is reserved for user processes. Table 3–2 names and describes the partitions.

Table 3–2. Virtual Memory Partitions

Partition	Virtual Address Range	Used By	Memory Access	User Mode Access	Default Data Cacheability
I/O *(1)*	`0xE0000000–0xFFFFFFFF`	Operating system	Bypasses TLB	No	Disabled
Kernel *(1)*	`0xC0000000–0xDFFFFFFF`	Operating system	Bypasses TLB	No	Enabled
Kernel MMU *(1)*	`0x80000000–0xBFFFFFFF`	Operating system	Uses TLB	No	Set by TLB
User	`0x00000000–0x7FFFFFFF`	User processes	Uses TLB	Set by TLB	Set by TLB

Note to Table 3–2:

(1) Supervisor-only partition

Each partition has a specific size, purpose, and relationship to the TLB:

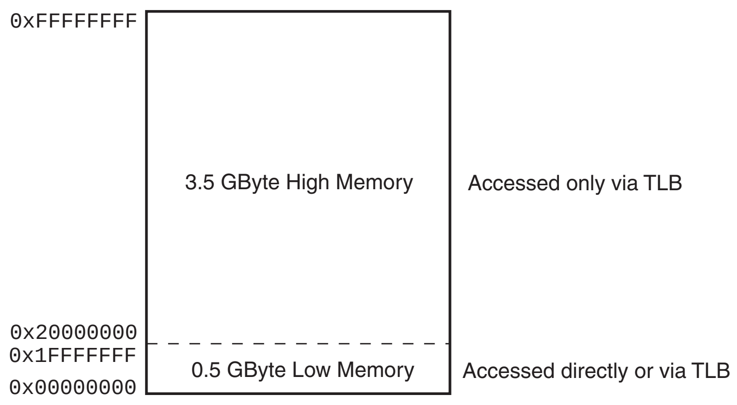
- The 512 MByte I/O partition provides access to peripherals.
- The 512 MByte kernel partition provides space for the operating system kernel.
- The 1 GByte kernel MMU partition is used by the TLB miss handler and kernel processes.
- The 2 GByte user partition is used by application processes.

I/O and kernel partitions bypass the TLB. The kernel MMU and user partitions use the TLB. If all software runs in the kernel partition, the MMU is effectively disabled.

Physical Memory Address Space

The 4 GByte physical memory is divided into low memory and high memory. The lowest 0.5 GBytes of physical address space is low memory. The upper 3.5 GBytes of physical address space is high memory. Figure 3–1 shows how physical memory is divided.

Figure 3–1. Division of Physical Memory



High physical memory can only be accessed through the TLB. Any physical address in low memory (29-bits or less) can be accessed through the TLB or by bypassing the TLB. When bypassing the TLB, a 29-bit physical address is computed by clearing the top three bits of the 32-bit virtual address.

☞ To function correctly, the base physical address of all exception vectors (reset, general exception, break, and fast TLB miss) must point to low physical memory so that hardware can correctly map their virtual addresses into the kernel partition. This restriction is enforced by the Nios II Processor MegaWizard interface in SOPC Builder.

Data Cacheability

Each partition has a rule that determines the default data cacheability property of each memory access. When data cacheability is enabled on a partition of the address space, a data access to that partition can be cached, if a data cache is present in the system. When data cacheability is disabled, all access to that partition goes directly to the Avalon switch fabric. Bit 31 is not used to specify data cacheability, as it is in Nios II cores without MMUs. Virtual memory partitions that bypass the TLB have a default data cacheability property, as shown in Table 3–2. For partitions that are mapped through the TLB, data cacheability is controlled by the TLB on a per-page basis.

Non-I/O load and store instructions use the default data cacheability property. I/O load and store instructions are always non-cacheable, so they ignore the default data cacheability property.

TLB Organization

A TLB functions as a cache for the operating system's page table. In Nios II processors with an MMU, one main TLB is shared by instruction and data accesses. The TLB is stored in on-chip RAM and handles translations for instruction fetches and instructions that perform data accesses.

The TLB is organized as an *n*-way set-associative cache. The software specifies the way (set) when loading a new entry.

☞ You can configure the number of TLB entries and the number of ways (set associativity) of the TLB in SOPC Builder at system generation time. By default, the TLB is a 16-way cache. The default number of entries depends on the target device, as follows:

- Cyclone®, Cyclone II, Stratix®, Stratix II, Stratix II GX—128 entries, requiring one M4K RAM
- Cyclone III, Stratix III, Stratix IV—256 entries, requiring one M9K RAM

  For further detail, refer to the *Instantiating the Nios II Processor in SOPC Builder* chapter of the *Nios II Processor Reference Handbook*.

The operating system software is responsible for guaranteeing that multiple TLB entries do not map the same virtual address. The hardware behavior is undefined when multiple entries map the same virtual address.

Each TLB entry consists of a tag and data portion. This is analogous to the tag and data portion of instruction and data caches.

Refer to the *Nios II Core Implementation Details* chapter of the *Nios II Processor Reference Handbook* for details on instruction and data caches.

The tag portion of a TLB entry contains information used when matching a virtual address to a TLB entry. Table 3–3 describes the tag portion of a TLB entry.

Table 3–3. TLB Tag Portion Contents

Field Name	Description
VPN	VPN is the virtual page number field. This field is compared with the top 20 bits of the virtual address.
PID	PID is the process identifier field. This field is compared with the value of the current process identifier stored in the tlbmisc control register, effectively extending the virtual address. The field size is configurable at system generation time, and can be between 8 and 14 bits.
G	G is the global flag. When G = 1, the PID is ignored in the TLB lookup.

The TLB data portion determines how to translate a matching virtual address to a physical address. Table 3–4 describes the data portion of a TLB entry.

Table 3–4. TLB Data Portion Contents

Field Name	Description
PFN	PFN is the physical frame number field. This field specifies the upper bits of the physical address. The size of this field depends on the range of physical addresses present in the system. The maximum size is 20 bits.
C	C is the cacheable flag. Determines the default data cacheability of a page. Can be overridden for data accesses using I/O load and store family of Nios II instructions.
R	R is the readable flag. Allows load instructions to read a page.
W	W is the writeable flag. Allows store instructions to write a page.
X	X is the executable flag. Allows instruction fetches from a page.

Because there is no "valid bit" in the TLB entry, the operating system software invalidates the TLB by writing unique VPN values from the I/O partition of virtual addresses into each TLB entry.

TLB Lookups

A TLB lookup attempts to convert a virtual address (VADDR) to a physical address (PADDR).

The TLB lookup algorithm for instruction fetches is as follows:

```
if (VPN match and (G = 1 or PID match))
    if (X = 1)
        PADDR = concat(PFN, VADDR[11:0])
    else
        take TLB permission violation exception
else
    if (EH bit of status register = 1)
        take double TLB miss exception
    else
        take fast TLB miss exception
```

Refer to "Instruction-Related Exceptions" on page 3–30 for details on TLB exceptions.

The TLB lookup algorithm for data accesses is as follows:

```
if (VPN match and (G = 1 or PID match))
    if ((load and R = 1) or (store and W = 1) or flushda)
        PADDR = concatenate(PFN, VADDR[11:0])
    else
        take TLB permission violation exception
else
    if (EH bit of status register = 1)
        take double TLB miss exception
    else
        take fast TLB miss exception
```

Memory Protection Unit

The Nios II processor provides an MPU for operating systems and runtime environments that desire memory protection but do not require virtual memory management. For information about memory protection with virtual memory management, refer to “Memory Management Unit” on page 3–3.

When present and enabled, the MPU monitors all Nios II instruction fetches and data memory accesses to protect against errant software execution. The MPU is a hardware facility that system software uses to define memory regions and their associated access permissions. The MPU triggers a precise exception if software attempts to access a memory region in violation of its permissions, allowing you to intervene and handle the exception as appropriate. The precise exception effectively prevents the illegal access to memory.

The MPU extends the Nios II processor to support user mode and supervisor mode. Typically, system software runs in supervisor mode and end-user applications run in user mode, although all software can run in supervisor mode if desired. System software defines which MPU regions belong to supervisor mode and which belong to user mode.

Memory Regions

The MPU contains up to 32 instruction regions and 32 data regions. Each region is defined by the following attributes:

- Base address
- Region type
- Region index
- Region size or upper address limit
- Access permissions
- Default cacheability (data regions only)

Base Address

The base address specifies the lowest address of the region. The base address is aligned on a region-sized boundary. For example, a 4 Kbyte region must have a base address that is a multiple of 4 Kbytes. If the base address is not properly aligned, the behavior is undefined.

Region Type

Each region is identified as either an instruction region or a data region.

Region Index

Each region has an index ranging from zero to the number of regions of its region type minus one. Index zero has the highest priority.

Region Size or Upper Address Limit

An SOPC Builder generation-time option controls whether the amount of memory in the region is defined by size or upper address limit. The size is an integer power of two bytes. The limit is the highest address of the region plus one. The minimum supported region size is 64 bytes but can be configured at system generation time for larger minimum sizes to save logic resources. The maximum supported region size equals the Nios II address space (a function of the address ranges of slaves connected to the Nios II masters). Any access outside of the Nios II address space is considered not to match any region and triggers an MPU region violation exception.

When regions are defined by size, the size is encoded as a binary mask to facilitate the following MPU region address range matching:

```
(address & region_mask) == region_base_address
```

When regions are defined by limit, the limit is encoded as an unsigned integer to facilitate the following MPU region address range matching:

```
(address >= region_base) && (address < region_limit)
```

The region limit uses a less-than instead of a less-than-or-equal-to comparison because less-than provides a more efficient implementation. The limit is one bit larger than the address so that full address range may be included in a range. Defining the region by limit results in slower and larger address range match logic than defining by size but allows finer granularity in region sizes.

Access Permissions

The access permissions consist of execute permissions for instruction regions and read/write permissions for data regions. Any instruction that performs a memory access that violates the access permissions triggers an exception. Additionally, any instruction that performs a memory access that does not match any region triggers an exception.

Default Cacheability

The default cacheability specifies whether normal load and store instructions access the data cache or bypass the data cache. The default cacheability is only present for data regions. You can override the default cacheability by using the `ldio` or `stio` instructions. The bit 31 cache bypass feature is available when the MPU is present. Refer to "Cache Memory" on page 3–38 for more information on cache bypass.

Overlapping Regions

The memory addresses of regions can overlap. Overlapping regions have several uses including placing markers or small holes inside of a larger region. For example, the stack and heap may be located in the same region. To detect stack/heap overflows, you can define a small region between the stack and heap with no access permissions and assign it a higher priority than the larger region. Any access attempts to the hole region trigger an exception informing system software about the stack/heap overflow.

If regions overlap so that a particular access matches more than one region, the region with the highest priority (lowest index) determines the access permissions and default cacheability.

Enabling the MPU

The MPU is disabled on system reset. System software enables and disables the MPU by writing to a control register. Before enabling the MPU, you must create at least one instruction and one data region, otherwise unexpected results can occur. Refer to "Working with the MPU" on page 3–24 for more information.

General-Purpose Registers

The Nios II architecture provides thirty-two 32-bit general-purpose registers, `r0` through `r31`, as shown in Table 3–5. Some registers have names recognized by the assembler. For example, the `zero` register (`r0`) always returns the value zero, and writing to `zero` has no effect. The `ra` register (`r31`) holds the return address used by procedure calls and is implicitly accessed by `call` and `ret` instructions. C and C++ compilers use a common procedure-call convention, assigning specific meaning to registers `r1` through `r23` and `r26` through `r28`.

Table 3–5. The Nios II General Purpose Registers (Part 1 of 2)

Register	Name	Function	Register	Name	Function
r0	zero	0x00000000	r16		
r1	at	Assembler temporary	r17		
r2		Return value	r18		
r3		Return value	r19		
r4		Register arguments	r20		
r5		Register arguments	r21		
r6		Register arguments	r22		
r7		Register arguments	r23		
r8		Caller-saved register	r24	et	Exception temporary
r9		Caller-saved register	r25	bt	Breakpoint temporary *(1)*
r10		Caller-saved register	r26	gp	Global pointer
r11		Caller-saved register	r27	sp	Stack pointer
r12		Caller-saved register	r28	fp	Frame pointer
r13		Caller-saved register	r29	ea	Exception return address
r14		Caller-saved register	r30	ba	Breakpoint return address *(1)*

Table 3–5. The Nios II General Purpose Registers (Part 2 of 2)

Register	Name	Function	Register	Name	Function
`r15`		Caller-saved register	`r31`	`ra`	Return address

Notes to Table 3–5:

(1) This register is used exclusively by the JTAG debug module.

For more information, refer to the *Application Binary Interface* chapter of the *Nios II Processor Reference Handbook*.

Control Registers

Control registers report the status and change the behavior of the processor. Control registers are accessed differently than the general-purpose registers. The special instructions `rdctl` and `wrctl` provide the only means to read and write to the control registers and are only available in supervisor mode.

When writing to control registers, all undefined bits must be written as zero.

The Nios II architecture supports up to 32 control registers. Table 3–6 shows details of the currently-defined control registers. All non-reserved control registers have names recognized by the assembler.

Table 3–6. Control Register Names and Bits

Register	Name	Register Contents
0	`status`	Refer to Table 3–7 on page 3–12
1	`estatus`	Refer to Table 3–9 on page 3–12
2	`bstatus`	Refer to Table 3–10 on page 3–13
3	`ienable`	Interrupt-enable bits
4	`ipending`	Pending-interrupt bits
5	`cpuid`	Unique processor identifier
6	Reserved	Reserved
7	`exception`	Refer to Table 3–11 on page 3–14
8	`pteaddr` *(1)*	Refer to Table 3–13 on page 3–14
9	`tlbacc` *(1)*	Refer to Table 3–15 on page 3–15
10	`tlbmisc` *(1)*	Refer to Table 3–17 on page 3–16
11	Reserved	Reserved
12	`badaddr`	Refer to Table 3–19 on page 3–19
13	`config` *(2)*	Refer to Table 3–21 on page 3–19
14	`mpubase` *(2)*	Refer to Table 3–23 on page 3–20
15	`mpuacc` *(2)*	Refer to Table 3–25 on page 3–21
16-31	Reserved	Reserved

Notes to Table 3–6:

(1) Available only when the MMU is present. Otherwise reserved.
(2) Available only when the MPU is present. Otherwise reserved.

The following sections describe the non-reserved control registers.

The status Register

The value in the `status` register controls the state of the Nios II processor. All status bits are cleared at processor reset. Some bits are exclusively used by and available only to certain features of the processor. Table 3–7 shows the layout of the `status` register.

Table 3–7. status Control Register Fields

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved																													`EH`	`U`	`PIE`

Table 3–8 gives details of the fields defined in the `status` register.

Table 3–8. status Control Register Field Descriptions

Bit	Description	Access	Reset	Available
`EH` *(1)*	`EH` is the exception handler bit. The processor sets `EH` to one when an exception occurs (including breaks). Software clears `EH` to zero when ready to handle exceptions again. `EH` is used by the MMU to determine whether a TLB miss exception is a fast TLB miss or a double TLB miss. In systems without an MMU, `EH` is always zero.	Read/Write	0	MMU only
`U` *(1)*	`U` is the user mode bit. When `U` = 1, the processor operates in user mode. When `U` = 0, the processor operates in supervisor mode. In systems without an MMU, `U` is always zero.	Read/Write	0	MMU or MPU only
`PIE`	`PIE` is the processor interrupt-enable bit. When `PIE` = 0, interrupts are ignored. When `PIE` = 1, interrupts can be taken, depending on the value of the `ienable` register.	Read/Write	0	Always

Notes to Table 3–8:

(1) The state where both `EH` and `U` are one is illegal and causes undefined results.

The estatus Register

The `estatus` register holds a saved copy of the `status` register during non-break exception processing. Table 3–9 shows the layout of the `status` register.

Table 3–9. estatus Control Register Fields

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved																													`EEH`	`EU`	`EPIE`

The names of the defined bits are the same as the status register bits prepended with the letter E. Table 3–8 also describes the details of the fields defined in the `estatus` register.

The exception handler can examine estatus to determine the pre-exception status of the processor. When returning from an exception, the `eret` instruction causes the processor to copy estatus back to status, restoring the pre-exception value of status. Refer to "Exception Processing" on page 3–25 for more information.

The bstatus Register

The bstatus register holds a saved copy of the status register during break exception processing. Table 3–10 shows the layout of the status register.

Table 3–10. bstatus Control Register Fields

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved																													BEH	BU	BPIE

The names of the defined bits are the same as the status register bits prepended with the letter B. Table 3–8 also describes the details of the fields defined in the estatus register.

When a break occurs, the value of the status register is copied into bstatus. Using bstatus, the debugger can restore the status register to the value prior to the break. The bret instruction causes the processor to copy bstatus back to status. Refer to "Processing a Break" on page 3–28 for more information.

The ienable Register

The ienable register controls the handling of external hardware interrupts. Each bit of the ienable register corresponds to one of the interrupt inputs, irq0 through irq31. A value of one in bit *n* means that the corresponding irq*n* interrupt is enabled; a bit value of zero means that the corresponding interrupt is disabled. Refer to "Exception Processing" on page 3–25 for more information.

The ipending Register

The value of the ipending register indicates the value of the interrupt signals driven into the processor. A value of one in bit *n* means that the corresponding irq*n* input is asserted. Writing a value to the ipending register has no effect.

The cpuid Register

The cpuid register holds a constant value that uniquely identifies each processor in a multi-processor system. The cpuid value is determined at system generation time and is guaranteed to be unique for each processor in the system. Writing to the cpuid register has no effect.

The exception Register

When the extra exception information option is enabled, the Nios II processor provides information useful to system software for exception processing in the exception and badaddr registers when an exception occurs. When your system contains an MMU or MPU, the extra exception information is always enabled. When no MMU or MPU is present, the Nios II Megawizard interface gives you the option to have the processor provide the extra exception information.

To see how to control the extra exception information option, refer to the *Instantiating the Nios II Processor in SOPC Builder* chapter of the *Nios II Processor Reference Handbook.*

Table 3–11 shows the layout of the exception register.

Table 3–11. exception Control Register Field Descriptions

Field	Description	Access	Reset	Available
`CAUSE`	`CAUSE` is written by the Nios II processor when any non-break exception occurs. `CAUSE` contains a code for the highest-priority exception occurring at the time. The Cause column in Table 3–31 on page 3–26 shows the `CAUSE` field value for each exception.	Read	0	Only with extra exception information

Table 3–12 gives details of the fields defined in the `exception` register.

Table 3–12. exception Control Register Fields

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved																									`CAUSE`					Rsvd	

The pteaddr Register

The `pteaddr` register contains the virtual address of the operating system's page table and is only available in systems with an MMU. The `pteaddr` register layout accelerates fast TLB miss exception handling. Table 3–13 shows the layout of the `pteaddr` register.

Table 3–13. pteaddr Control Register Fields

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
`PTBASE`										`VPN`																				Rsvd	

Table 3–14 gives details of the fields defined in the `pteaddr` register.

Table 3–14. pteaddr Control Register Field Descriptions

Field	Description	Access	Reset	Available
`PTBASE`	`PTBASE` is the base virtual address of the page table.	Read/Write	0	Only with MMU
`VPN`	`VPN` is the virtual page number. `VPN` can be set by both hardware and software.	Read/Write	0	Only with MMU

Software writes to the `PTBASE` field when switching processes. Hardware never writes to the `PTBASE` field.

Software writes to the `VPN` field when writing a TLB entry. Hardware writes to the `VPN` field on a fast TLB miss exception, a TLB permission violation exception, or on a TLB read operation. The `VPN` field is not written on any exceptions taken when an exception is already active, that is, when `status.EH` is already one.

The tlbacc Register

The `tlbacc` register is used to access TLB entries and is only available in systems with an MMU. The `tlbacc` register holds values that software will write into a TLB entry or has previously read from a TLB entry. The `tlbacc` register provides access to only a portion of a complete TLB entry. `pteaddr.VPN` and `tlbmisc.PID` hold the remaining TLB entry fields.

Table 3–15 shows the layout of the `tlbacc` register.

Table 3–15. tlbacc Control Register Fields

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IG							C	R	W	X	G	PFN																			

Table 3–16 gives details of the fields defined in the `tlbacc` register.

Table 3–16. tlbacc Control Register Field Descriptions

Field	Description	Access	Reset	Available
`IG`	`IG` is ignored by hardware and available to hold operating system specific information. Read as zero but can be written as non-zero.	Read/Write	0	Only with MMU
`C`	`C` is the data cacheable flag. When `C` = 0, data accesses are uncacheable. When `C` = 1, data accesses are cacheable.	Read/Write	0	Only with MMU
`R`	`R` is the readable flag. When `R` = 0, load instructions are not allowed to access memory. When `R` = 1, load instructions are allowed to access memory.	Read/Write	0	Only with MMU
`W`	`W` is the writable flag. When `W` = 0, store instructions are not allowed to access memory. When `W` = 1, store instructions are allowed to access memory.	Read/Write	0	Only with MMU
`X`	`X` is the executable flag. When `X` = 0, instructions are not allowed to execute. When `X` = 1, instructions are allowed to execute.	Read/Write	0	Only with MMU
`G`	`G` is the global flag. When `G` = 0, `tlbmisc.PID` is included in the TLB lookup. When `G` = 1, `tlbmisc.PID` is ignored and only the virtual page number is used in the TLB lookup.	Read/Write	0	Only with MMU
`PFN`	`PFN` is the physical frame number field. All unused upper bits must be zero.	Read/Write	0	Only with MMU

Issuing a `wrctl` instruction to the `tlbacc` register writes the `tlbacc` register with the specified value. If `tlbmisc.WE` = 1, the `wrctl` instruction also initiates a TLB write operation, which writes a TLB entry. The TLB entry written is specified by the line portion of `pteaddr.VPN` and the `tlbmisc.WAY` field. The value written is specified by the value written into `tlbacc` along with the values of `pteaddr.VPN` and `tlbmisc.PID`. A TLB write operation also increments `tlbmisc.WAY`, allowing software to quickly modify TLB entries.

Issuing a `rdctl` instruction to the `tlbacc` register returns the value of the `tlbacc` register. The `tlbacc` register is written by hardware when software triggers a TLB read operation (that is, when `wrctl` sets `tlbmisc.RD` to one).

The `tlbacc` register format is the recommended format for entries in the operating system page table. The `IG` bits are ignored by the hardware on `wrctl` to `tlbacc` and read back as zero on `rdctl` from `tlbacc`. The operating system can use the `IG` bits to hold operating system specific information without having to clear these bits to zero on a TLB write operation.

The tlbmisc Register

The `tlbmisc` register contains the remaining TLB-related fields and is only available in systems with an MMU. Table 3–17 shows the layout of the `tlbmisc` register.

Table 3–17. tlbmisc Control Register Fields

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved								`WAY` *(1)*				`RD`	`WE`	`PID` *(1)*														`DBL`	`BAD`	`PERM`	`D`

Notes to Table 3–17:

(1) This field size is variable. Unused upper bits must written as zero.

Table 3–18 gives details of the fields defined in the `tlbmisc` register.

Table 3–18. tlbmisc Control Register Field Descriptions

Field	Description	Access	Reset	Available
`WAY`	The `WAY` field controls the mapping from the VPN to a particular TLB entry.	Read/Write	0	Only with MMU
`RD`	`RD` is the read flag. Setting `RD` to one triggers a TLB read operation.	Write	0	Only with MMU
`WE`	`WE` is the TLB write enable flag. When `WE` = 1, a write to `tlbacc` writes through to a TLB entry.	Read/Write	0	Only with MMU
`PID`	`PID` is the process identifier field.	Read/Write	0	Only with MMU
`DBL` *(1)*	`DBL` is the double TLB miss exception flag.	Read	0	Only with MMU
`BAD` *(1)*	`BAD` is the bad virtual address exception flag.	Read	0	Only with MMU
`PERM` *(1)*	`PERM` is the TLB permission violation exception flag.	Read	0	Only with MMU
`D`	`D` is the data access exception flag. When `D` = 1, the exception is a data access exception. When `D` = 0, the exception is an instruction access exception.	Read	0	Only with MMU

Notes to Table 3–18:

(1) You can also use `exception.CAUSE` to determine these exceptions.

The sections below provide further details of the `tlbmisc` fields.

The RD Flag

System software triggers a TLB read operation by setting `tlbmisc.RD` (with a `wrctl` instruction). A TLB read operation loads the following register fields with the contents of a TLB entry:

- The tag portion of `pteaddr.VPN`
- `tlbmisc.PID`
- The `tlbacc` register

The TLB entry to be read is specified by the following values:

- the line portion of `pteaddr.VPN`
- `tlbmisc.WAY`

When system software changes the fields that specify the TLB entry, there is no immediate effect on `pteaddr.VPN`, `tlbmisc.PID`, or the `tlbacc` register. The registers retain their previous values until the next TLB read operation is initiated. For example, when the operating system sets `pteaddr.VPN` to a new value, the contents of `tlbacc` continues to reflect the previous TLB entry. `tlbacc` does not contain the new TLB entry until after an explicit TLB read.

The WE Flag

When `WE` = 1, a write to `tlbacc` writes the `tlbacc` register and a TLB entry. When `WE` = 0, a write to `tlbacc` only writes the `tlbacc` register.

Hardware sets the `WE flag` to one on a TLB permission violation exception, and on a TLB miss exception when `status.EH` = 0. When a TLB write operation writes the `tlbacc` register, the write operation also writes to a TLB entry when `WE` = 1.

The WAY Field

The `WAY` field controls the mapping from the VPN to a particular TLB entry. `WAY` specifies the set to be written to in the TLB. The MMU increments `WAY` when system software performs a TLB write operation. Unused upper bits in `WAY` must be written as zero.

☞ The number of ways (sets) is configurable in SOPC Builder at generation time, up to a maximum of 16.

The PID Field

`PID` is a unique identifier for the current process that effectively extends the virtual address. The process identifier can be less than 14 bits. Any unused upper bits must be zero.

`tlbmisc.PID` contains the `PID` field from a TLB tag. The operating system must set the `PID` field when switching processes, and before each TLB write operation.

☞ Use of the process identifier is optional. To implement memory management without process identifiers, clear `tlbmisc.PID` to zero. Without a process identifier, all processes share the same virtual address space.

The MMU sets `tlbmisc.PID` on a TLB read operation. When the software triggers a TLB read, by setting `tlbmisc.RD` to one with the `wrctl` instruction, the `PID` value read from the TLB has priority over the value written by the `wrctl` instruction.

The size of the `PID` field is configured in SOPC Builder at system generation, and can be from 8 to 14 bits. If system software defines a process identifier smaller than the `PID` field, unused upper bits must be written as zero.

The DBL Flag

During a general exception, the processor sets `DBL` to one when a double TLB miss condition exists. Otherwise, the processor clears `DBL` to zero.

The `DBL` flag indicates whether the most recent exception is a double TLB miss condition. When a general exception occurs, the MMU sets `DBL` to one if a double TLB miss is detected, and clears `DBL` to zero otherwise.

The BAD Flag

During a general exception, the processor sets `BAD` to one when a bad virtual address condition exists, and clears `BAD` to zero otherwise. The following exceptions set the `BAD` flag to one:

- Supervisor-only instruction address
- Supervisor-only data address
- Misaligned data address
- Misaligned destination address

Refer to Table 3–31 on page 3–26 for more information on these exceptions.

The PERM Flag

During a general exception, the processor sets `PERM` to one for a TLB permission violation exception, and clears `PERM` to zero otherwise.

The D Flag

The `D` flag indicates whether the exception is an instruction access exception or a data access exception. During a general exception, the processor sets `D` to one when the exception is related to a data access, and clears `D` to zero for all other non-break exceptions.

The following exceptions set the `D` flag to one:

- Fast TLB miss (data)
- Double TLB miss (data)
- TLB permission violation (read or write)
- Misaligned data address
- Supervisor-only data address

The badaddr Register

When the extra exception information option is enabled, the Nios II processor provides information useful to system software for exception processing in the `exception` and `badaddr` registers when an exception occurs. When your system contains an MMU or MPU, the extra exception information is always enabled. When no MMU or MPU is present, the Nios II Megawizard interface gives you the option to have the processor provide the extra exception information.

To see how to control the extra exception information option, refer to the *Instantiating the Nios II Processor in SOPC Builder* chapter of the *Nios II Processor Reference Handbook*.

When the option for extra exception information is enabled and a processor exception occurs, the `badaddr` register contains the byte instruction or data address associated with certain exceptions at the time the exception occurred. Table 3–31 on page 3–26 shows which exceptions write the `badaddr` register along with the value written. Table 3–19 shows the layout of the `badaddr` register.

Table 3–19. badaddr Control Register Fields

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BADDR																															

Table 3–20 gives details of the fields defined in the badaddr register.

Table 3–20. badaddr Control Register Field Descriptions

Field	Description	Access	Reset	Available
BADDR	BADDR contains the byte instruction address or data address associated with an exception when certain exceptions occur. The Address column of Table 3–31 on page 3–26 shows which exceptions write the BADDR field.	Read	0	Only with extra exception information

The BADDR field allows up to a 32-bit instruction address or data address. If an MMU or MPU is present, the BADDR field is 32 bits because MMU and MPU instruction and data addresses are always full 32-bit values. When an MMU is present, the BADDR field contains the virtual address.

If there is no MMU or MPU and the Nios II address space is less than 32 bits, unused high-order bits are written and read as zero. If there is no MMU, bit 31 of a data address (used to bypass the data cache) is always zero in the BADDR field.

The config Register

The config register configures Nios II runtime behaviors that do not need to be preserved during exception processing (in contrast to the information in the status register). Table 3–21 shows the layout of the config register.

Table 3–21. config Control Register Fields

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved																															PE

Table 3–22 gives details of the fields defined in the config register

Table 3–22. config Control Register Field Descriptions

Field	Description	Access	Reset	Available
PE	PE is the memory protection enable bit. When PE =1, the MPU is enabled. When PE = 0, the MPU is disabled. In systems without an MPU, PE is always zero.	Read/Write	0	Only with MPU

The mpubase Register

The mpubase register works in conjunction with the mpuacc register to set and retrieve MPU region information and is only available in systems with an MPU. Table 3–23 shows the layout of the mpubase register.

Table 3–23. mpubase Control Register Fields

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	BASE *(2)*																									INDEX *(1)*					D

Notes to Table 3–23:

(1) This field size is variable. Unused upper bits must written as zero.

(2) This field size is variable. Unused upper bits and unused lower bits must written as zero.

Table 3–24 gives details of the fields defined in the mpubase register

Table 3–24. mpubase Control Register Field Descriptions

Field	Description	Access	Reset	Available
BASE	BASE is the base memory address of the region identified by the INDEX and D fields.	Read/Write	0	Only with MPU
INDEX	INDEX is the region index number.	Read/Write	0	Only with MPU
D	D is the region access bit. When D =1, INDEX refers to a data region. When D = 0, INDEX refers to an instruction region.	Read/Write	0	Only with MPU

The BASE field specifies the base address of an MPU region. The 25-bit BASE field corresponds to bits 6 through 30 of the base address, making the base address always a multiple of 64 bytes. If the minimum region size set in SOPC Builder at generation time is larger than 64 bytes, unused low-order bits of the BASE field must be written as zero and are read as zero. For example, if the minimum region size is 1024 bytes, the four least-significant bits of the BASE field (bits 6 though 9 of the mpubase register) must be zero. Similarly, if the Nios II address space is less than 31 bits, unused high-order bits must also be written as zero and are read as zero.

The INDEX and D fields specify the region information to access when an MPU region read or write operation is performed. The D field specifies whether the region is a data region or an instruction region. The INDEX field specifies which of the 32 data or instruction regions to access. If there are fewer than 32 instruction or 32 data regions, unused high-order bits must be written as zero and are read as zero.

Refer to "MPU Region Read and Write Operations" on page 3–24 for more information on MPU region read and write operations.

The mpuacc Register

The mpuacc register works in conjunction with the mpubase register to set and retrieve MPU region information and is only available in systems with an MPU. The mpuacc register consists of attributes that will be set or have been retrieved which define the MPU region. The mpuacc register only holds a portion of the attributes that define an MPU region. The remaining portion of the MPU region definition is held by the BASE field of the mpubase register.

An SOPC Builder generation-time option controls whether the mpuacc register contains a MASK or LIMIT field. Table 3–25 shows the layout of the mpuacc register with the MASK field. Table 3–26 shows the layout of the mpuacc register with the LIMIT field.

Table 3–25. mpuacc Control Register Fields for MASK Variation

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	MASK *(1)*																									C	PERM			RD	WR

Note to Table 3–25:

(1) This field size is variable. Unused upper bits and unused lower bits must written as zero.

Table 3–26. mpuacc Control Register Fields for LIMIT Variation

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LIMIT *(1)*																										C	PERM			RD	WR

Note to Table 3–26:

(1) This field size is variable. Unused upper bits and unused lower bits must written as zero.

Table 3–27 gives details of the fields defined in the `mpuacc` register.

Table 3–27. mpuacc Control Register Field Descriptions

Field	Description	Access	Reset	Available
`MASK` *(1)*	`MASK` specifies the size of the region.	Read/Write	0	Only with MPU
`LIMIT` *(1)*	`LIMIT` specifies the upper address limit of the region.	Read/Write	0	Only with MPU
`C`	`C` is the data cacheable flag. `C` only applies to MPU data regions and determines the default cacheability of a data region. When `C` = 0, the data region is uncacheable. When `C` = 1, the data region is cacheable.	Read/Write	0	Only with MPU
`PERM`	`PERM` specifies the access permissions for the region.	Read/Write	0	Only with MPU
`RD`	`RD` is the read region flag. When `RD` = 1, `wrctl` instructions to the `mpuacc` register perform a read operation.	Write	0	Only with MPU
`WE`	`WR` is the write region flag. When `WR` = 1, `wrctl` instructions to the `mpuacc` register perform a write operation.	Write	0	Only with MPU

Note to Table 3–27:

(1) The `MASK` and `LIMIT` fields are mutually exclusive. Refer to Table 3–25 and Table 3–26.

The sections below provide further details of the `mpuacc` fields.

The MASK Field

When the amount of memory reserved for a region is defined by size, the `MASK` field specifies the size of the memory region. The `MASK` field is the same number of bits as the `BASE` field of the `mpubase` register.

Unused high-order or low-order bits must be written as zero and are read as zero.

Table 3–28 shows the `MASK` field encodings for all possible region sizes in a full 31-bit byte address space.

Table 3–28. MASK Region Size Encodings

MASK Encoding	Region Size
`0x1FFFFFF`	64 bytes
`0x1FFFFFE`	128 bytes
`0x1FFFFFC`	256 bytes
`0x1FFFFF8`	512 bytes
`0x1FFFFF0`	1 Kbyte
`0x1FFFFE0`	2 Kbytes
`0x1FFFFC0`	4 Kbytes
`0x1FFFF80`	8 Kbytes
`0x1FFFF00`	16 Kbytes
`0x1FFFE00`	32 Kbytes
`0x1FFFC00`	64 Kbytes
`0x1FFF800`	128 Kbytes
`0x1FFF000`	256 Kbytes
`0x1FFE000`	512 Kbytes
`0x1FFC000`	1 Mbyte
`0x1FF8000`	2 Mbytes
`0x1FF0000`	4 Mbytes
`0x1FE0000`	8 Mbytes
`0x1FC0000`	16 Mbytes
`0x1F80000`	32 Mbytes
`0x1F00000`	64 Mbytes
`0x1E00000`	128 Mbytes
`0x1C00000`	256 Mbytes
`0x1800000`	512 Mbytes
`0x1000000`	1 Gbyte
`0x0000000`	2 Gbytes

Bit 31 of the `mpuacc` register is not used by the `MASK` field. Because memory region size is already a power of two, one less bit is needed. The `MASK` field contains the following value, where `region_size` is in bytes:

```
MASK = 0x1FFFFFF << log2(region_size >> 6)
```

The LIMIT Field

When the amount of memory reserved for a region is defined by an upper address limit, the `LIMIT` field specifies the upper address of the memory region plus one. For example, to achieve a memory range for byte addresses `0x4000` to `0x4fff` with a 256 byte minimum region size, the `BASE` field of the `mpubase` register is set to `0x40` (`0x4000 >> 8`) and the `LIMIT` field is set to `0x50` (`0x5000 >> 8`). Because the `LIMIT` field is one more bit than the number of bits of the `BASE` field of the `mpubase` register, bit 31 of the `mpuacc` register is available to the `LIMIT` field.

The C Flag

The C flag determines the default data cacheability of an MPU region. The C flag only applies to data regions. For instruction regions, the C bit must be written with 0 and is always read as 0.

When data cacheability is enabled on a data region, a data access to that region can be cached, if a data cache is present in the system. You can override the default cacheability and force an address to non-cacheable with an `ldio` or `stio` instruction.

☞ The bit 31 cache bypass feature is supported when the MPU is present. Refer to "Cache Memory" on page 3–38 for more information on cache bypass.

The PERM Field

The PERM field specifies the allowed access permissions. Table 3–29 shows possible values of the PERM field for instruction regions and Table 3–30 shows possible values of the PERM field for data regions.

Table 3–29. Instruction Region Permission Values

Value	Supervisor Permissions	User Permissions
0	None	None
1	Execute	None
2	Execute	Execute

Table 3–30. Data Region Permission Values

Value	Supervisor Permissions	User Permissions
0	None	None
1	Read	None
2	Read	Read
4	Read/Write	None
5	Read/Write	Read
6	Read/Write	Read/Write

☞ Unlisted table values are reserved and must not be used. If you use reserved values, the resulting behavior is undefined.

The RD Flag

Setting the RD flag signifies that an MPU region read operation should be performed when a `wrctl` instruction is issued to the `mpuacc` register. Refer to "MPU Region Read and Write Operations" on page 3–24 for more information. The RD flag always returns 0 when read by a `rdctl` instruction.

The WR Flag

Setting the WR flag signifies that an MPU region write operation should be performed when a `wrctl` instruction is issued to the `mpuacc` register. Refer to "MPU Region Read and Write Operations" on page 3–24 for more information. The WR flag always returns 0 when read by a `rdctl` instruction.

☞ Setting both the `RD` and `WR` flags to one results in undefined behavior.

Working with the MPU

This section provides a basic overview of MPU initialization and the MPU region read and write operations.

MPU Region Read and Write Operations

MPU region read and write operations are operations that access MPU region attributes through the `mpubase` and `mpuacc` control registers. The `mpubase.BASE`, `mpuacc.MASK`, `mpuacc.LIMIT`, `mpuacc.C`, and `mpuacc.PERM` fields comprise the MPU region attributes.

MPU region read operations retrieve the current values for the attributes of a region. Each MPU region read operation consists of the following actions:

- Execute a `wrctl` instruction to the `mpubase` register with the `mpubase.INDEX` and `mpubase.D` fields set to identify the MPU region.
- Execute a `wrctl` instruction to the `mpuacc` register with the `mpuacc.RD` field set to one and the `mpuacc.WR` field cleared to zero. This action loads the `mpubase` and `mpuacc` register values.
- Execute a `rdctl` instruction to the `mpubase` register to read the loaded the `mpubase` register value.
- Execute a `rdctl` instruction to the `mpuacc` register to read the loaded the `mpuacc` register value.

The MPU region read operation retrieves `mpubase.BASE`, `mpuacc.MASK` or `mpuacc.LIMIT`, `mpuacc.C`, and `mpuacc.PERM` values for the MPU region.

☞ Values for the `mpubase` register are not actually retrieved until the `wrctl` instruction to the `mpuacc` register is performed.

MPU region write operations set new values for the attributes of a region. Each MPU region write operation consists of the following actions:

- Execute a `wrctl` instruction to the `mpubase` register with the `mpubase.INDEX` and `mpubase.D` fields set to identify the MPU region.
- Execute a `wrctl` instruction to the `mpuacc` register with the `mpuacc.WR` field set to one and the `mpuacc.RD` field cleared to zero.

The MPU region write operation sets the values for `mpubase.BASE`, `mpuacc.MASK` or `mpuacc.LIMIT`, `mpuacc.C`, and `mpuacc.PERM` as the new attributes for the MPU region.

Normally, a `wrctl` instruction flushes the pipeline to guarantee that any side effects of writing control registers take effect immediately after the `wrctl` instruction completes execution. However, `wrctl` instructions to the `mpubase` and `mpuacc` control registers do not automatically flush the pipeline. Instead, system software is responsible for flushing the pipeline as needed (either by using a `flushp` instruction or a `wrctl` instruction to a register that does flush the pipeline). Because a context switch typically requires reprogramming the MPU regions for the new thread, flushing the pipeline on each `wrctl` instruction would create unnecessary overhead.

MPU Initialization

Your system software must provide a data structure that contains the region information described in "Memory Regions" on page 3–8 for each active thread. The data structure ideally contains two 32-bit values that correspond to the `mpubase` and `mpuacc` register formats.

The MPU is disabled on system reset. Before enabling the MPU, Altera recommends initializing all MPU regions. Enable desired instruction and data regions by writing each region's attributes to the `mpubase` and `mpuacc` registers as described in "MPU Region Read and Write Operations" on page 3–24. You must also disable unused regions. When using region size, clear `mpuacc.MASK` to zero. When using limit, set the `mpubase.BASE` to a non-zero value and clear `mpuacc.LIMIT` to zero.

☞ You must enable at least one instruction and one data region, otherwise unpredictable behavior might occur.

To perform a context switch, use a `wrctl` to write a zero to the `PE` field of the `config` register to disable the MPU, define all MPU regions from the new thread's data structure, and then use another `wrctl` to write a one to `config.PE` to enable the MPU.

Define each region using the pair of `wrctl` instructions described in "MPU Region Read and Write Operations" on page 3–24. Repeat this dual `wrctl` instruction sequence until all desired regions are defined.

Debugger Access

The debugger can access all MPU-related control registers using the normal `wrctl` and `rdctl` instructions. During debugging, the Nios II ignores the MPU, effectively temporarily disabling it.

Exception Processing

An exception is a transfer of control away from a program's normal flow of execution, caused by an event, either internal or external to the processor, which requires immediate attention. Exception processing is the act of responding to an exception, and then returning to the pre-exception execution state.

Each of the Nios II exceptions falls into one of the following categories:

- Reset exceptions
- Break exceptions

- Interrupt exceptions
- Instruction-related exceptions

Table 3–31 shows all possible Nios II exceptions in order of highest to lowest priority. The following table columns specify information for the exceptions:

- Exception—Gives the name of the exception.
- Type—Specifies the exception type.
- Available—Specifies when support for that exception is present.
- Cause—Specifies the value of the `CAUSE` field of the `exception` register, for exceptions that write the `exception.CAUSE` field.
- Address—Specifies the instruction or data address associated with the exception.
- Vector—Specifies which exception vector address the processor passes control to when the exception occurs.

Table 3–31. Nios II Exceptions (In Decreasing Priority Order) (Part 1 of 2)

Exception	Type	Available	Cause	Address	Vector
Reset	Reset	Always	0		Reset
Hardware Break	Break	Always	—		Break
Processor-only Reset Request	Reset	Always	1		Reset
Interrupt	Interrupt	Always	2	`ea-4` *(2)*	General exception
Supervisor-only Instruction Address *(1)*	Instruction-related	MMU	9	`ea-4` *(2)*	General exception
Fast TLB Miss (instruction) *(1)*	Instruction-related	MMU	12	`pteaddr.VPN`, `ea-4` *(2)*	Fast TLB Miss exception
Double TLB Miss (instruction) *(1)*	Instruction-related	MMU	12	`pteaddr.VPN`, `ea-4` *(2)*	General exception
TLB Permission Violation (execute) *(1)*	Instruction-related	MMU	13	`pteaddr.VPN`, `ea-4` *(2)*	General exception
MPU Region Violation (instruction) *(1)*	Instruction-related	MPU	16	`ea-4` *(2)*	General exception
Supervisor-only Instruction	Instruction-related	MMU or MPU	10	`ea-4` *(2)*	General exception
Trap Instruction	Instruction-related	Always	3	`ea-4` *(2)*	General exception
Illegal Instruction	Instruction-related	Illegal instruction detection on, MMU, or MPU	5	`ea-4` *(2)*	General exception
Unimplemented Instruction	Instruction-related	Always	4	`ea-4` *(2)*	General exception
Break Instruction	Instruction-related	Always	—	`ba-4` *(2)*	Break
Supervisor-only Data Address	Instruction-related	MMU	11	`badaddr` (data address)	General exception
Misaligned Data Address	Instruction-related	Illegal memory access detection on, MMU, or MPU	6	`badaddr` (data address)	General exception

Table 3–31. Nios II Exceptions (In Decreasing Priority Order) (Part 2 of 2)

Exception	Type	Available	Cause	Address	Vector
Misaligned Destination Address	Instruction-related	Illegal memory access detection on, MMU, or MPU	7	`badaddr` (destination address)	General exception
Division Error	Instruction-related	Division error detection on	8	`ea-4` *(2)*	General exception
Fast TLB Miss (data)	Instruction-related	MMU	12	`pteaddr.VPN`, `badaddr` (data address)	Fast TLB Miss exception
Double TLB Miss (data)	Instruction-related	MMU	12	`pteaddr.VPN`, `badaddr` (data address)	General exception
TLB Permission Violation (read)	Instruction-related	MMU	14	`pteaddr.VPN`, `badaddr` (data address)	General exception
TLB Permission Violation (write)	Instruction-related	MMU	15	`pteaddr.VPN`, `badaddr` (data address)	General exception
MPU Region Violation (data)	Instruction-related	MPU	17	`badaddr` (data address)	General exception

Notes to Table 3–31:

(1) It is possible for any instruction fetch to cause this exception.

(2) Refer to Table 3–6 on page 3–11 for descriptions of the `ea` and `ba` registers.

The following sections describe each exception type in detail.

Reset Exceptions

When a processor reset signal is asserted, the Nios II processor performs the following steps:

1. Clears the `status` register to 0x0.
2. Invalidates the instruction-cache line associated with the reset vector.
3. Begins executing the reset handler, located at the reset vector.

Clearing the `status` register disables hardware interrupts. If the MMU or MPU is present, clearing the `status` register forces the processor into supervisor mode.

Invalidating the reset cache line guarantees that instruction fetches for reset code comes from uncached memory.

Aside from the instruction-cache line associated with the reset vector, the contents of the cache memories are indeterminate after reset. To ensure cache coherency after reset, the reset handler located at the reset vector must immediately initialize the instruction cache. Next, either the reset handler or a subsequent routine should proceed to initialize the data cache.

The reset state is undefined for all other system components, including but not limited to:

- General-purpose registers, except for `zero` (`r0`) which is permanently zero.

- Control registers, except for `status` which is reset to 0x0.
- Instruction and data memory.
- Cache memory, except for the instruction-cache line associated with the reset vector.
- Peripherals. Refer to the appropriate peripheral data sheet or specification for reset conditions.
- Custom instruction logic. Refer to the *Nios II Custom Instruction User Guide* for reset conditions.
- Nios II C-to-hardware (C2H) acceleration compiler logic.

Break Exceptions

A break is a transfer of control away from a program's normal flow of execution for the purpose of debugging. Software debugging tools can take control of the Nios II processor via the JTAG debug module.

Break processing is the means by which software debugging tools implement debug and diagnostic features, such as breakpoints and watchpoints. Break processing is a type of exception processing, but the break mechanism is independent from general exception processing. A break can occur during exception processing, enabling debug tools to debug exception handlers.

The processor enters the break processing state under either of the following conditions:

- The processor executes the `break` instruction. This is often referred to as a software break.
- The JTAG debug module asserts a hardware break.

Processing a Break

A break causes the processor to take the following steps:

1. Stores the contents of the `status` register to `bstatus`.
2. Clears `status.PIE` to zero, disabling external processor interrupts.
3. Writes the address of the instruction following the break to the `ba` register (`r30`).
4. Clears `status.U` to zero, forcing the processor into supervisor mode, when the system contains an MMU or MPU.
5. Sets `status.EH` to one, indicating the processor is handling an exception, when the system contains an MMU.
6. Transfers execution to the break handler, stored at the break vector specified at system generation time.

Register Usage

The `bstatus` control register and general-purpose registers `bt` (`r25`) and `ba` (`r30`) are reserved for debugging. Code is not prevented from writing to these registers, but debug code might overwrite the values. The break handler can use `bt` (`r25`) to help save additional registers.

Returning From a Break

After processing a break, the break handler releases control of the processor by executing a `bret` instruction. The `bret` instruction restores `status` by copying the contents of `bstatus` and returns program execution to the address in the `ba` register (r30). Aside from `bt`, all registers are guaranteed to be returned to their pre-break state after returning from the break handler.

Interrupt Exceptions

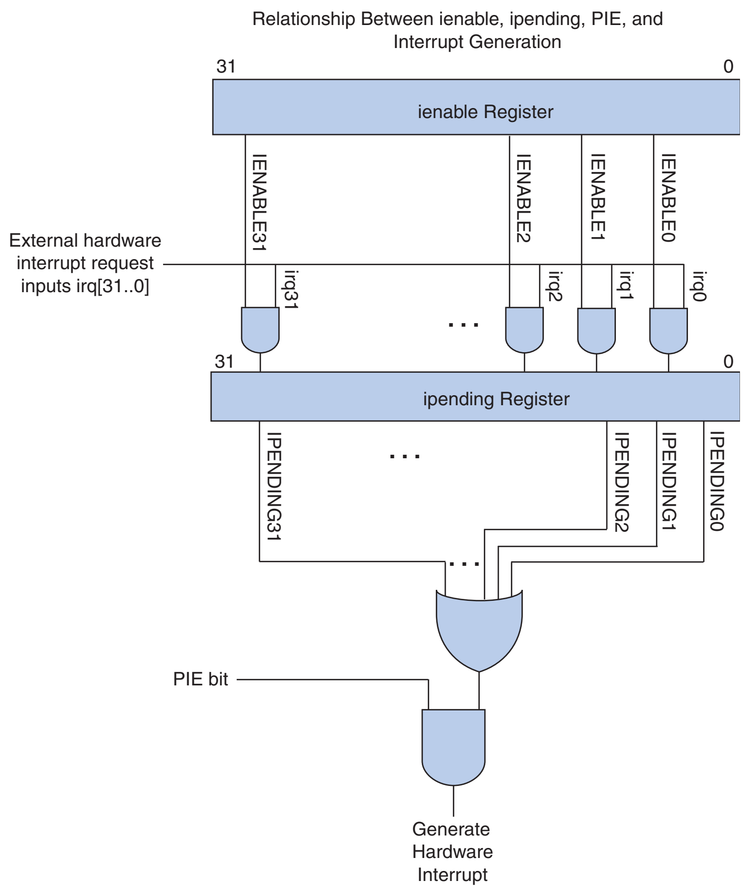
An external source such as a peripheral device can request a hardware interrupt by asserting one of the processor's 32 interrupt-request inputs, `irq0` through `irq31`. A hardware interrupt is generated if and only if all three of these conditions are true:

- The `PIE` bit of the `status` control register is one.
- An interrupt-request input, `irq`*n*, is asserted.
- The corresponding bit *n* of the `ienable` control register is one.

Upon hardware interrupt, the processor clears the `PIE` bit to zero, disabling further interrupts, and performs the other steps outlined in "Processing Interrupt and Instruction-Related Exceptions" on page 3–35.

The value of the `ipending` control register shows which interrupt requests (IRQ) are pending. By peripheral design, an IRQ bit is guaranteed to remain asserted until the processor explicitly responds to the peripheral. Figure 3–2 shows the relationship between `ipending`, `ienable`, `PIE`, and the generation of an interrupt.

Figure 3–2. Relationship Between ienable, ipending, PIE and Hardware Interrupts


Relationship Between ienable, ipending, PIE, and Interrupt Generation
31
0
ienable Register
IENABLE31
IENABLE2
IENABLE1
IENABLE0
External hardware interrupt request inputs irq[31..0]
irq31
irq2
irq1
irq0
31
0
ipending Register
IPENDING31
IPENDING2
IPENDING1
IPENDING0
PIE bit
Generate Hardware Interrupt

A software exception routine determines which of the pending interrupts has the highest priority, and then transfers control to the appropriate interrupt service routine (ISR). The ISR stops the interrupt from being visible (either by clearing it at the source or masking it using `ienable`) before returning and/or before re-enabling `PIE`. The ISR also saves `estatus` and `ea` (`r29`) before re-enabling `PIE`.

Interrupts can be re-enabled by writing one to the `PIE` bit, thereby allowing the current ISR to be interrupted. Typically, the exception routine adjusts `ienable` so that IRQs of equal or lower priority are disabled before re-enabling interrupts. Refer to "Nested Exception Precautions" on page 3–37 for more information.

Instruction-Related Exceptions

Instruction-related exceptions occur during execution of Nios II instructions and perform the steps outlined in "Processing Interrupt and Instruction-Related Exceptions" on page 3–35.

The Nios II processor generates the following instruction-related exceptions. All instruction-related exceptions are precise.

- Trap instruction
- Break instruction

- Unimplemented instruction
- Illegal instruction
- Supervisor-only instruction
- Supervisor-only instruction address
- Supervisor-only data address
- Misaligned data address
- Misaligned destination address
- Division error
- Fast TLB miss
- Double TLB miss
- TLB permission violation
- MPU region violation

Trap Instruction

When a program issues the `trap` instruction, it generates a software trap exception. A program typically issues a software trap when the program requires servicing by the operating system. The general exception handler for the operating system determines the reason for the trap and responds appropriately.

Break Instruction

The break instruction is treated as a break exception. Refer to "Break Exceptions" on page 3–28 for details.

Unimplemented Instruction

When the processor issues a valid instruction that is not implemented in hardware, an unimplemented instruction exception is generated. The general exception handler determines which instruction generated the exception. If the instruction is not implemented in hardware, control is passed to an exception routine that might choose to emulate the instruction in software. Refer to "Potential Unimplemented Instructions" on page 3–44 for more information.

Illegal Instruction

Illegal instructions are instructions with an undefined opcode or opcode-extension field. The Nios II processor can check for illegal instructions and generate an exception when an illegal instruction is encountered. When your system contains an MMU or MPU, illegal instruction checking is always on. When no MMU or MPU is present, you have the option to have the processor check for illegal instructions.

To see how to control this option, refer to the *Instantiating the Nios II Processor in SOPC Builder* chapter of the *Nios II Processor Reference Handbook.*

When the processor issues an instruction with an undefined opcode or opcode-extension field, and illegal instruction exception checking is turned on, an illegal instruction exception is generated.

Refer to the OP Encodings and OPX Encodings for R-Type Instructions tables in the *Instruction Set Reference* chapter of the *Nios II Processor Reference Handbook* to see the unused opcodes and opcode extensions.

All undefined opcodes are reserved. The processor does occasionally use some undefined encodings internally. Executing one of these undefined opcodes does not trigger an illegal instruction exception. Refer to the *Nios II Core Implementation Details* chapter of the *Nios II Processor Reference Handbook* for details on each specific Nios II core.

Supervisor-only Instruction

When your system contains an MMU or MPU and the processor is in user mode (`status.U = 1`), executing a supervisor-only instruction results in a supervisor-only instruction exception. The supervisor-only instructions are `initd`, `initi`, `eret`, `bret`, `rdctl`, and `wrctl`.

This exception is implemented only in Nios II processors configured to use supervisor mode and user mode. Refer to "Operating Modes" on page 3–2 for more information.

Supervisor-only Instruction Address

When your system contains an MMU and the processor is in user mode (`status.U = 1`), attempts to access a supervisor-only instruction address result in a supervisor-only instruction address exception. Any instruction fetch can cause this exception. For definitions of supervisor-only address ranges, refer to Table 3–2 on page 3–5.

This exception is implemented only in Nios II processors that include the MMU.

Supervisor-only Data Address

When your system contains an MMU and the processor is in user mode (`status.U = 1`), any attempt to access a supervisor-only data address results in a supervisor-only data address exception. Instructions that can cause a supervisor-only data address exception are all loads, all stores, and `flushda`.

This exception is implemented only in Nios II processors that include the MMU.

Misaligned Data Address

The Nios II processor can check for misaligned data addresses of load and store instructions and generate an exception when a misaligned data address is encountered. When your system contains an MMU or MPU, misaligned data address checking is always on. When no MMU or MPU is present, you have the option to have the processor check for misaligned data addresses.

To see how to control this option, refer to the *Instantiating the Nios II Processor in SOPC Builder* chapter of the *Nios II Processor Reference Handbook.*

A data address is considered misaligned if the byte address is not a multiple of the width of the load or store instruction data width (four bytes for word, two bytes for half-word). Byte load and store instructions are always aligned so never take a misaligned address exception.

Misaligned Destination Address

The Nios II processor can check for misaligned destination addresses of the `callr`, `jmp`, `ret`, `eret`, `bret`, and all branch instructions and generate an exception when a misaligned destination address is encountered. When your system contains an MMU or MPU, misaligned destination address checking is always on. When no MMU or MPU is present, you have the option to have the processor check for misaligned destination addresses.

To see how to control this option, refer to the *Instantiating the Nios II Processor in SOPC Builder* chapter of the *Nios II Processor Reference Handbook.*

A destination address is considered misaligned if the target byte address of the instruction is not a multiple of four.

Division Error

The Nios II processor can check for division errors and generate an exception when a division error is encountered.

To see how to control this option, refer to the *Instantiating the Nios II Processor in SOPC Builder* chapter of the *Nios II Processor Reference Handbook.*

The division error exception detects divide instructions that produce a quotient that can't be represented. The two cases are divide by zero and a signed division that divides the largest negative number -2147483648 (0x80000000) by -1 (0xffffffff). Division error detection is only available if divide instructions are supported by hardware.

Fast TLB Miss

Fast TLB miss exceptions are implemented only in Nios II processors that include the MMU. The MMU has a special exception vector (fast TLB miss), specified in SOPC Builder at system generation time, specifically to handle TLB miss exceptions quickly. Whenever the processor cannot find a TLB entry matching the VPN (optionally extended by a process identifier), the result is a TLB miss exception. At the time of the exception, the processor first checks `status.EH`. When `status.EH` = 0, no other exception is already in process, so the processor considers the TLB miss a fast TLB miss, sets `status.EH` to one, and transfers control to the fast TLB miss exception handler (rather than to the general exception handler).

There are two kinds of fast TLB miss exceptions:

- Fast TLB miss (instruction)—Any instruction fetch can cause this exception.
- Fast TLB miss (data)—Load, store, `initda`, and `flushda` instructions can cause this exception.

The fast TLB miss exception handler can inspect the `tlbmisc.D` field to determine which kind of fast TLB miss exception occurred.

Double TLB Miss

Double TLB miss exceptions are implemented only in Nios II processors that include the MMU. When a TLB miss exception occurs while software is currently processing an exception (that is, when `status.EH` = 1), a double TLB miss exception is generated. Specifically, whenever the processor cannot find a TLB entry matching the VPN (optionally extended by a process identifier) and `status.EH` = `1`, the result is a double TLB miss exception. The most common scenario is that a double TLB miss exception occurs during processing of a fast TLB miss exception. The processor preserves register values from the original exception and transfers control to the general exception handler which processes the newly-generated exception.

There are two kinds of double TLB miss exceptions:

- Double TLB miss (instruction)—Any instruction fetch can cause this exception.
- Double TLB miss (data)—Load, store, `initda`, and `flushda` instructions can cause this exception.

The general exception handler can inspect either the `exception.CAUSE` or `tlbmisc.D` field to determine which kind of double TLB miss exception occurred.

TLB Permission Violation

TLB permission violation exceptions are implemented only in Nios II processors that include the MMU. When a TLB entry is found matching the VPN (optionally extended by a process identifier), but the permissions do not allow the access to complete, a TLB permission violation exception is generated.

There are three kinds of TLB permission violation exceptions:

- TLB permission violation (execute)—Any instruction fetch can cause this exception.
- TLB permission violation (read)—Any load instruction can cause this exception.
- TLB permission violation (write)—Any store instruction can cause this exception.

The general exception handler can inspect the `exception.CAUSE` field to determine which permissions were violated.

The data cache management instructions (`initd`, `initda`, `flushd`, and `flushda`) ignore the TLB permissions and do not generate TLB permission violation exceptions.

MPU Region Violation

MPU region violation exceptions are implemented only in Nios II processors that include the MPU. An MPU region violation exception is generated under any of the following conditions:

- An instruction fetch or data address matched a region but the permissions for that region did not allow the action to complete.
- An instruction fetch or data address did not match any region.

The general exception handler reads the MPU region attributes to determine if the address did not match any region or which permissions were violated.

There are two kinds of MPU region violation exceptions:

- MPU region violation (instruction)—Any instruction fetch can cause this exception.
- MPU region violation (data)—Load, store, `initda`, and `flushda` instructions can cause this exception.

The general exception handler can inspect the `exception.CAUSE` field to determine which kind of MPU region violation exception occurred.

Other Exceptions

The previous sections describe all of the exception types defined by the Nios II architecture at the time of publishing. However, some processor implementations might generate exceptions that do not fall into the above categories. Therefore, a robust exception handler must provide a safe response (such as issuing a warning) in the event that it cannot identify the cause of an exception.

Processing Interrupt and Instruction-Related Exceptions

Except for the break instruction (refer to "Processing a Break" on page 3–28), this section describes the actions the processor takes in response to interrupt and instruction-related exceptions. Table 3–32 lists all possible non-break exception processing actions performed by hardware. Check marks indicate which actions apply to each of the processor scenarios, namely, systems without an MMU, systems with an MMU, and systems with an MMU that is currently processing an exception. For systems with an MMU, `status.EH` indicates whether or not exception processing is already in progress. When `status.EH` = 1, exception processing is already in progress and the states of the exception registers are preserved to retain the original exception states.

Table 3–32. Non-Break Exception Processing Actions (Part 1 of 2)

Processor Actions (in order of occurrence)	No MMU	MMU and EH = 0	MMU and EH = 1
Copies the contents of the `status` control register to the `estatus` control register, saving the processor's pre-exception status.	✓	✓	
Clears `status.PIE` to zero, disabling external processor interrupts.	✓	✓	✓
Writes the address of the instruction following the exception to the `ea` register (`r29`).	✓	✓	
Clears `status.U` to zero, forcing the processor into supervisor mode.		✓	✓
Sets `status.EH` to one, indicating the processor is handling an exception.		✓	
If fast TLB miss or a TLB permission violation exception, writes the VPN of the address triggering the exception to `pteaddr.VPN`.		✓	
Conditionally writes to `tlbmisc.D`. Refer to "The D Flag" on page 3–18 for more information.		✓	
Conditionally writes to `tlbmisc.DBL`. Refer to "The DBL Flag" on page 3–17 for more information.		✓	✓
Conditionally writes to `tlbmisc.PERM`. Refer to "The PERM Flag" on page 3–18 for more information.		✓	✓
Conditionally writes to `tlbmisc.BAD`. Refer to "The BAD Flag" on page 3–18 for more information.		✓	✓

Table 3–32. Non-Break Exception Processing Actions (Part 2 of 2)

Processor Actions (in order of occurrence)	No MMU	MMU and EH = 0	MMU and EH = 1
Passes control to the general exception vector, invoking the general exception handler	✓		✓
Passes control to an exception handler: ■ If the exception is a TLB miss, control passes to the fast TLB miss exception vector, invoking the fast TLB miss handler. ■ If the exception is not a TLB miss, control passes to the general exception vector, invoking the general exception handler		✓	

The fast TLB miss exception handler is a routine that handles only the fast TLB miss execption. It is built for speed to process TLB misses quickly.

The general exception handler is a routine that determines the cause of each exception (including the double TLB miss exception), and then dispatches an exception routine to respond to the exception. The address of the general exception handler, specified in SOPC Builder at system generation time, is called the exception vector in the Nios II Megawizard interface. At run time this address is fixed, and software cannot modify it. Programmers do not directly access exception vectors, and can write programs without awareness of the address.

The fast TLB miss exception handler only handles the fast TLB miss exception. The fast TLB miss exception handler address, specified in SOPC Builder at system generation time, is called the fast TLB miss exception vector in the Nios II Megawizard interface.

For a detailed discussion of writing programs to take advantage of exception and interrupt handling, refer to the *Exception Handling* chapter of the *Nios II Software Developer's Handbook*.

Determining the Cause of Interrupt and Instruction-Related Exceptions

The general exception handler must determine the cause of each exception and then transfer control to an appropriate exception routine.

With Extra Exception Information

When you have included the extra exception information in your Nios II system, the `CAUSE` field of the `exception` register (refer to "The exception Register" on page 3–13) contains a code for the highest-priority exception occurring at the time and the `BADDR` field of the `badaddr` register (refer to "The badaddr Register" on page 3–18) contains the byte instruction address or data address for certain exceptions. Refer to Table 3–31 on page 3–26 for more information.

To determine the cause of an exception, simply read the cause of the exception from `exception.CAUSE` and then transfer control to the appropriate exception routine.

Extra exception information is always enabled in Nios II systems containing an MMU or MPU.

Without Extra Exception Information

When you have not included the extra exception information in your Nios II system, your exception handler must determine the cause of exception itself. For this reason, Altera recommends always enabling the extra exception information.

When the extra exception information is not available, use the following sequence to determine the cause of an exception:

```
/* Check for interrupt exceptions first*/
if (estatus.EPIE == 1 and ipending != 0) {
   handle interrupt

/* Decode exception from instruction */
/* Note: Because the exception register is included with the MMU and */
/* MPU, you never need to determine MMU or MPU exceptions by decoding */
} else {
   decode instruction at $ea-4
   if (instruction is trap)
      handle trap exception
   else if (instruction is load or store)
      handle misaligned data address exception
   else if (instruction is branch, bret, callr, eret, jmp, or ret)
      handle misaligned destination address exception
   else if (instruction is unimplemented)
      handle unimplemented instruction exception
   else if (instruction is illegal)
      handle illegal instruction exception
   else if (instruction is divide) {
      if (denominator == 0)
         handle division error exception
      else if (instruction is signed divide and numerator == 0x80000000
                                        and denominator == 0xffffffff)
         handle division error exception
      }
   }

   /* Not any known exception */
   } else {
      handle unknown exception (could be spurious interrupt)
   }
}
```

Nested Exception Precautions

Exception routines must take special precautions before any of the following actions:

- Issuing a `trap` instruction
- Issuing a potentially unimplemented instruction
- Re-enabling hardware interrupts

For details about unimplemented instructions, refer to the *Processor Architecture* chapter of the *Nios II Processor Reference Handbook.*

Before allowing any of these actions, the exception routine must save `estatus` and `ea` (r29), then restore them properly before returning to preserve the pre-exception state of the exception registers.

Returning From Interrupt and Instruction-Related Exceptions

The `eret` instruction is used to resume execution at the pre-exception address. Except for the `et` register (r24), the exception routine must restore any registers modified during exception processing before returning.

When executing the `eret` instruction, the processor performs the following tasks:

1. Copies the contents of `estatus` to `status`
2. Transfers program execution to the address in the `ea` register (r29)

Return Address Considerations

The return address requires some consideration when returning from exception processing routines. After an exception occurs, `ea` contains the address of the instruction following the point where the exception occurred.

When returning from instruction-related exceptions, execution must resume from the instruction following the instruction where the exception occurred. Therefore, `ea` contains the correct return address.

On the other hand, hardware interrupt exceptions must resume execution from the interrupted instruction itself. In this case, the exception handler must subtract 4 from `ea` to point to the interrupted instruction.

Memory and Peripheral Access

Nios II addresses are 32 bits, allowing access up to a 4 gigabyte address space. Nios II core implementations without MMUs restrict addresses to 31 bits or fewer. The MMU supports the full 32-bit physical address.

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

Peripherals, data memory, and program memory are mapped into the same address space. The locations of memory and peripherals within the address space are determined at system generation time. Reading or writing to an address that does not map to a memory or peripheral produces an undefined result.

The processor's data bus is 32 bits wide. Instructions are available to read and write byte, half-word (16-bit), or word (32-bit) data.

The Nios II architecture is little endian. For data wider than 8 bits stored in memory, the more-significant bits are located in higher addresses.

The Nios II architecture supports register+immediate addressing.

Cache Memory

The Nios II architecture and instruction set accommodate the presence of data cache and instruction cache memories. Cache management is implemented in software by using cache management instructions. Instructions are provided to initialize the cache, flush the caches whenever necessary, and to bypass the data cache to properly access memory-mapped peripherals.

The Nios II architecture provides the following mechanisms to bypass the cache:

- When no MMU is present, bit 31 of the address is reserved for bit-31 cache bypass. With bit-31 cache bypass, the address space of processor cores is 2 GBytes, and the high bit of the address controls the caching of data memory accesses.
- When the MMU is present, cacheability is controlled by the MMU, and bit 31 functions as a normal address bit. For details, refer to "Address Space and Memory Partitions" on page 3–4, and "TLB Organization" on page 3–6.
- Cache bypass instructions, such as `ldwio` and `stwio`.

Refer to the *Nios II Core Implementation Details* chapter of the *Nios II Processor Reference Handbook* for details of which processor cores implement bit-31 cache bypass. Refer to *Instruction Set Reference* chapter of the *Nios II Processor Reference Handbook* for details of the cache bypass instructions.

Code written for a processor core with cache memory behaves correctly on a processor core without cache memory. The reverse is not true. If it is necessary for a program to work properly on multiple Nios II processor core implementations, the program must behave as if the instruction and data caches exist. In systems without cache memory, the cache management instructions perform no operation, and their effects are benign.

For a complete discussion of cache management, refer to the *Cache and Tightly Coupled Memory* chapter of the *Nios II Software Developer's Handbook*.

Some consideration is necessary to ensure cache coherency after processor reset. Refer to "Reset Exceptions" on page 3–27 for more information.

For details on the cache architecture and the memory hierarchy refer to the *Processor Architecture* chapter of the *Nios II Processor Reference Handbook*.

Virtual Address Aliasing

A virtual address alias occurs when two virtual addresses map to the same physical address. When an MMU and caches are present and the caches are larger than a page (4 KBytes), the operating system must prevent illegal virtual address aliases. Because the caches are virtually-indexed and physically-tagged, a portion of the virtual address is used to select the cache line. If the cache is 4 KBytes or less in size, the portion of the virtual address used to select the cache line fits with bits 11:0 of the virtual address which have the same value as bits 11:0 of the physical address (they are untranslated bits of the page offset). However, if the cache is larger than 4 KBytes, bits beyond the page offset (bits 12 and up) are used to select the cache line and these bits are allowed to be different than the corresponding physical address.

For example, in a 64 KByte direct-mapped cache with a 16-byte line, bits 15:4 are used to select the line. Assume that virtual address `0x1000` is mapped to physical address `0xF000` and virtual address `0x2000` is also mapped to physical address `0xF000`. This is an illegal virtual address alias because accesses to virtual address `0x1000` use line 0x1 and accesses to virtual address `0x2000` use line 0x2 even though they map to the same physical address. This results in two copies of the same physical address in the cache. With an *n*-byte direct-mapped cache, there could be $n/4096$ copies of the same physical address in the cache if illegal virtual address aliases are not prevented. The bits of the virtual address that are used to select the line and are translated bits

(bits 12 and up) are known as the color of the address. An operating system avoids illegal virtual address aliases by ensuring that if multiple virtual addresses map the same physical address, the virtual addresses have the same color. Note though, the color of the virtual addresses does not need to be the same as the color as the physical address because the cache tag contains all the bits of the PFN.

Instruction Set Categories

This section introduces the Nios II instructions categorized by type of operation performed.

Data Transfer Instructions

The Nios II architecture is a load-store architecture. Load and store instructions handle all data movement between registers, memory, and peripherals. Memories and peripherals share a common address space. Some Nios II processor cores use memory caching and/or write buffering to improve memory bandwidth. The architecture provides instructions for both cached and uncached accesses.

Table 3–33 describes the wide (32-bit) load and store instructions.

Table 3–33. Wide Data Transfer Instructions

Instruction	Description
`ldw` `stw`	The `ldw` and `stw` instructions load and store 32-bit data words from/to memory. The effective address is the sum of a register's contents and a signed immediate value contained in the instruction. Memory transfers can be cached or buffered to improve program performance. This caching and buffering might cause memory cycles to occur out of order, and caching might suppress some cycles entirely. Data transfers for I/O peripherals should use `ldwio` and `stwio`.
`ldwio` `stwio`	`ldwio` and `stwio` instructions load and store 32-bit data words from/to peripherals without caching and buffering. Access cycles for `ldwio` and `stwio` instructions are guaranteed to occur in instruction order and are never suppressed.

The data transfer instructions in Table 3–34 support byte and half-word transfers.

Table 3–34. Narrow Data Transfer Instructions

Instruction	Description

`ldb`
`ldbu`
`stb`
`ldh`
`ldhu`
`sth`
	`ldb`, `ldbu`, `ldh` and `ldhu` load a byte or half-word from memory to a register. `ldb` and `ldh` sign-extend the value to 32 bits, and `ldbu` and `ldhu` zero-extend the value to 32 bits.
	`stb` and `sth` store byte and half-word values, respectively.
	Memory accesses can be cached or buffered to improve performance. To transfer data to I/O peripherals, use the “io” versions of the instructions, described below.

`ldbio`
`ldbuio`
`stbio`
`ldhio`
`ldhuio`
`sthio`
	These operations load/store byte and half-word data from/to peripherals without caching or buffering.

Arithmetic and Logical Instructions

Logical instructions support `and`, `or`, `xor`, and `nor` operations. Arithmetic instructions support addition, subtraction, multiplication, and division operations. Refer to Table 3–35.

Table 3–35. Arithmetic and Logical Instructions

Instruction	Description

`and`
`or`
`xor`
`nor`
	These are the standard 32-bit logical operations. These operations take two register values and combine them bit-wise to form a result for a third register.

`andi`
`ori`
`xori`
	These operations are immediate versions of the `and`, `or`, and `xor` instructions. The 16-bit immediate value is zero-extended to 32 bits, and then combined with a register value to form the result.

`andhi`
`orhi`
`xorhi`
	In these versions of `and`, `or`, and `xor`, the 16-bit immediate value is shifted logically left by 16 bits to form a 32-bit operand. Zeroes are shifted in from the right.

`add`
`sub`
`mul`
`div`
`divu`
	These are the standard 32-bit arithmetic operations. These operations take two registers as input and store the result in a third register.

`addi`
`subi`
`muli`
	These instructions are immediate versions of the `add`, `sub`, and `mul` instructions. The instruction word includes a 16-bit signed value.

`mulxss` `mulxuu`	These instructions provide access to the upper 32 bits of a 32x32 multiplication operation. Choose the appropriate instruction depending on whether the operands should be treated as signed or unsigned values. It is not necessary to precede these instructions with a `mul`.
`mulxsu`	This instruction is used in computing a 128-bit result of a 64x64 signed multiplication.

Move Instructions

These instructions provide move operations to copy the value of a register or an immediate value to another register. Refer to Table 3–36.

Table 3–36. Move Instructions

Instruction	Description

`mov`
`movhi`
`movi`
`movui`
`movia`
	`mov` copies the value of one register to another register. `movi` moves a 16-bit signed immediate value to a register, and sign-extends the value to 32 bits. `movui` and `movhi` move an immediate 16-bit value into the lower or upper 16-bits of a register, inserting zeros in the remaining bit positions. Use `movia` to load a register with an address.

Comparison Instructions

The Nios II architecture supports a number of comparison instructions. All of these compare two registers or a register and an immediate value, and write either one (if true) or zero to the result register. These instructions perform all the equality and relational operators of the C programming language. Refer to Table 3–37.

Table 3–37. Comparison Instructions

Instruction	Description
`cmpeq`	==
`cmpne`	!=
`cmpge`	signed >=
`cmpgeu`	unsigned >=
`cmpgt`	signed >
`cmpgtu`	unsigned >
`cmple`	unsigned <=
`cmpleu`	unsigned <=
`cmplt`	signed <
`cmpltu`	unsigned <

`cmpeqi`
`cmpnei`
`cmpgei`
`cmpgeui`
`cmpgti`
`cmpgtui`
`cmplei`
`cmpleui`
`cmplti`
`cmpltui`
	These instructions are immediate versions of the comparison operations. They compare the value of a register and a 16-bit immediate value. Signed operations sign-extend the immediate value to 32-bits. Unsigned operations fill the upper bits with zero.

Shift and Rotate Instructions

The following instructions provide shift and rotate operations. The number of bits to rotate or shift can be specified in a register or an immediate value. Refer to Table 3–38.

Table 3–38. Shift and Rotate Instructions

Instructio n	Description

`rol`
`ror`
`roli`
	The `rol` and `roli` instructions provide left bit-rotation. `roli` uses an immediate value to specify the number of bits to rotate. The `ror` instructions provides right bit-rotation.
	There is no immediate version of `ror`, because `roli` can be used to implement the equivalent operation.

`sll`
`slli`
`sra`
`srl`
`srai`
`srli`
	These shift instructions implement the `<<` and `>>` operators of the C programming language. The `sll`, `slli`, `srl`, `srli` instructions provide left and right logical bit-shifting operations, inserting zeros. The `sra` and `srai` instructions provide arithmetic right bit-shifting, duplicating the sign bit in the most significant bit. `slli`, `srli` and `srai` use an immediate value to specify the number of bits to shift.

Program Control Instructions

The Nios II architecture supports the unconditional jump and call instructions listed in Table 3–39. These instructions do not have delay slots.

Table 3–39. Unconditional Jump and Call Instructions

Instruction	Description
`call`	This instruction calls a subroutine using an immediate value as the subroutine's absolute address, and stores the return address in register `ra`.
`callr`	This instruction calls a subroutine at the absolute address contained in a register, and stores the return address in register `ra`. This instruction serves the roll of dereferencing a C function pointer.
`ret`	The `ret` instruction is used to return from subroutines called by `call` or `callr`. `ret` loads and executes the instruction specified by the address in register `ra`.
`jmp`	The `jmp` instruction jumps to an absolute address contained in a register. `jmp` is used to implement switch statements of the C programming language.
`jmpi`	The `jmpi` instruction jumps to an absolute address using an immediate value to determine the absolute address.
`or`	This instruction branches relative to the current instruction. A signed immediate value gives the offset of the next instruction to execute.

The conditional-branch instructions compare register values directly, and branch if the expression is true. Refer to Table 3–40. The conditional branches support the equality and relational comparisons of the C programming language:

- == and !=
- < and <= (signed and unsigned)
- > and >= (signed and unsigned)

The conditional-branch instructions do not have delay slots.

Table 3–40. Conditional-Branch Instructions

Instruction	Description

`bge`
`bgeu`
`bgt`
`bgtu`
`ble`
`bleu`
`blt`
`bltu`
`beq`
`bne`
	These instructions provide relative branches that compare two register values and branch if the expression is true. Refer to "Comparison Instructions" on page 3–41 for a description of the relational operations implemented.

Other Control Instructions

Table 3–41 shows other control instructions.

Table 3–41. Other Control Instructions

Instruction	Description
`trap` `eret`	The `trap` and `eret` instructions generate and return from exceptions. These instructions are similar to the `call/ret` pair, but are used for exceptions. `trap` saves the `status` register in the `estatus` register, saves the return address in the `ea` register, and then transfers execution to the general exception handler. `eret` returns from exception processing by restoring `status` from `estatus`, and executing the instruction specified by the address in `ea`.
`break` `bret`	The `break` and `bret` instructions generate and return from breaks. `break` and `bret` are used exclusively by software debugging tools. Programmers never use these instructions in application code.
`rdctl` `wrctl`	These instructions read and write control registers, such as the `status` register. The value is read from or stored to a general-purpose register.

`flushd`
`flushda`
`flushi`
`initd`
`initda`
`initi`
	These instructions are used to manage the data and instruction cache memories.

`flushp`	This instruction flushes all pre-fetched instructions from the pipeline. This is necessary before jumping to recently-modified instruction memory.
`sync`	This instruction ensures that all previously-issued operations have completed before allowing execution of subsequent load and store operations.

Custom Instructions

The `custom` instruction provides low-level access to custom instruction logic. The inclusion of custom instructions is specified in SOPC Builder at system generation time, and the function implemented by custom instruction logic is design dependent.

For further details, refer to the "Custom Instructions" section of the *Processor Architecture* chapter of the *Nios II Processor Reference Handbook* and the *Nios II Custom Instruction User Guide*.

Machine-generated C functions and assembly macros provide access to custom instructions, and hide implementation details from the user. Therefore, most software developers never use the `custom` assembly instruction directly.

No-Operation Instruction

The Nios II assembler provides a no-operation instruction, `nop`.

Potential Unimplemented Instructions

Some Nios II processor cores do not support all instructions in hardware. In this case, the processor generates an exception after issuing an unimplemented instruction. Only the following instructions can generate an unimplemented instruction exception:

- `mul`
- `muli`
- `mulxss`

- `mulxsu`
- `mulxuu`
- `div`
- `divu`
- `initda`

All other instructions are guaranteed not to generate an unimplemented instruction exception.

An exception routine must exercise caution if it uses these instructions, because they could generate another exception before the previous exception is properly handled. Refer to "Unimplemented Instruction" on page 3–31 for more information regarding unimplemented instruction processing.

Referenced Documents

This chapter references the following documents:

- *Nios II Software Developer's Handbook*
- *Processor Architecture* chapter of the *Nios II Processor Reference Handbook*
- *Application Binary Interface* chapter of the *Nios II Processor Reference Handbook*
- *Instruction Set Reference* chapter of the *Nios II Processor Reference Handbook*
- *Instantiating the Nios II Processor in SOPC Builder* chapter of the *Nios II Processor Reference Handbook*
- *Nios II Core Implementation Details* chapter of the *Nios II Processor Reference Handbook*
- *Exception Handling* chapter of the *Nios II Software Developer's Handbook*
- *Cache and Tightly Coupled Memory* chapter of the *Nios II Software Developer's Handbook*
- *Processor Architecture* chapter of the *Nios II Processor Reference Handbook*
- *Nios II Custom Instruction User Guide*

Document Revision History

Table 3–42 shows the revision history for this document.

Table 3–42. Document Revision History (Part 1 of 2)

Date & Document Version	Changes Made	Summary of Changes
March 2009 v9.0.0	Maintenance release.	—
November 2008 v8.1.0	Maintenance release.	—

Table 3–42. Document Revision History (Part 2 of 2)

Date & Document Version	Changes Made	Summary of Changes
May 2008 v8.0.0	Added text to describe the MMU, MPU, and advanced exceptions.	Added MMU, MPU, and advanced exceptions.
October 2007 v7.2.0	■ Reworked text to refer to break and reset as exceptions. ■ Grouped exceptions, break, reset, and interrupts all under Exception Processing. ■ Added table showing all Nios II exceptions (by priority). ■ Removed "ctl" references to control registers. ■ Added `jmpi` instruction to tables.	—
May 2007 v7.1.0	■ Added table of contents to Introduction section. ■ Added Referenced Documents section.	—
March 2007 v7.0.0	Maintenance release.	—
November 2006 v6.1.0	Maintenance release.	—
May 2006 v6.0.0	Maintenance release.	—
October 2005 v5.1.0	Maintenance release.	—
May 2005 v5.0.0	Maintenance release.	—
September 2004 v1.1	■ Added details for new control register `ctl5`. ■ Updated details of debug and break processing to reflect new behavior of the `break` instruction.	—
May 2004 v1.0	Initial release.	—