This section provides information about the Nios® II processor.

This section includes the following chapters:

- Chapter 1, Introduction
- Chapter 2, Processor Architecture
- Chapter 3, Programming Model
- Chapter 4, Instantiating the Nios II Processor

For information about the revision history for chapters in this section, refer to “Document Revision History” in each individual chapter.
This handbook describes the Nios II processor from a high-level conceptual description to the low-level details of implementation. The chapters in this handbook describe the Nios II processor architecture, the programming model, and the instruction set.

This handbook is the primary reference for the Nios® II family of embedded processors and is part of a larger collection of documents covering the Nios II processor and its usage that you can find on the Literature: Nios II Processor page of the Altera® website.

This handbook assumes you have a basic familiarity with embedded processor concepts. You do not need to be familiar with any specific Altera technology or with Altera development tools. This handbook limits discussion of hardware implementation details of the processor system. The Nios II processors are designed for Altera FPGA devices, and so this handbook does describe some FPGA implementation concepts. Your familiarity with FPGA technology provides a deeper understanding of the engineering trade-offs related to the design and implementation of the Nios II processor.

This chapter introduces the Altera Nios II embedded processor family and describes the similarities and differences between the Nios II processor and traditional embedded processors. This chapter contains the following sections:

- “Nios II Processor System Basics”
- “Getting Started with the Nios II Processor” on page 1–2
- “Customizing Nios II Processor Designs” on page 1–3
- “Configurable Soft Processor Core Concepts” on page 1–4
- “OpenCore Plus Evaluation” on page 1–6

### Nios II Processor System Basics

The Nios II processor is a general-purpose RISC processor core with the following features:

- Full 32-bit instruction set, data path, and address space
- 32 general-purpose registers
- Optional shadow register sets
- 32 interrupt sources
- External interrupt controller interface for more interrupt sources
- Single-instruction $32 \times 32$ multiply and divide producing a 32-bit result
- Dedicated instructions for computing 64-bit and 128-bit products of multiplication
Floating-point instructions for single-precision floating-point operations

- Single-instruction barrel shifter
- Access to a variety of on-chip peripherals, and interfaces to off-chip memories and peripherals
- Hardware-assisted debug module enabling processor start, stop, step, and trace under control of the Nios II software development tools
- Optional memory management unit (MMU) to support operating systems that require MMUs
- Optional memory protection unit (MPU)
- Software development environment based on the GNU C/C++ tool chain and the Nios II Software Build Tools (SBT) for Eclipse
- Integration with Altera’s SignalTap® II Embedded Logic Analyzer, enabling real-time analysis of instructions and data along with other signals in the FPGA design
- Instruction set architecture (ISA) compatible across all Nios II processor systems
- Performance up to 250 DMIPS

A Nios II processor system is equivalent to a microcontroller or “computer on a chip” that includes a processor and a combination of peripherals and memory on a single chip. A Nios II processor system consists of a Nios II processor core, a set of on-chip peripherals, on-chip memory, and interfaces to off-chip memory, all implemented on a single Altera device. Like a microcontroller family, all Nios II processor systems use a consistent instruction set and programming model.

Getting Started with the Nios II Processor

The easiest way to start designing effectively is to use an Altera development kit that includes a ready-made development board and the Nios II Embedded Design Suite (EDS) containing all the software development tools necessary to write Nios II software.

For a list of available development kits, refer to the All Development Kits page of the Altera website.

The Nios II EDS includes the following two closely-related software development tool flows:

- The Nios II SBT
- The Nios II SBT for Eclipse

Both tools flows are based on the GNU C/C++ compiler. The Nios II SBT for Eclipse provides a familiar and established environment for software development. Using the Nios II SBT for Eclipse, you can immediately begin developing and simulating Nios II software applications.

The Nios II SBT also provides a command line interface.
Using the Nios II hardware reference designs included in an Altera development kit, you can prototype an application running on a board before building a custom hardware platform. Figure 1–1 shows an example of a Nios II processor reference design available in an Altera development kit.

Figure 1–1. Example of a Nios II Processor System

If the prototype system adequately meets design requirements using an Altera-provided reference design, you can copy the reference design and use it without modification in the final hardware platform. Otherwise, you can customize the Nios II processor system until it meets cost or performance requirements.

Customizing Nios II Processor Designs

In practice, most FPGA designs implement some extra logic in addition to the processor system. Altera FPGAs provide flexibility to add features and enhance performance of the Nios II processor system. You can also eliminate unnecessary processor features and peripherals to fit the design in a smaller, lower-cost device.

Because the pins and logic resources in Altera devices are programmable, many customizations are possible:

- You can rearrange the pins on the chip to simplify the board design. For example, you can move address and data pins for external SDRAM memory to any side of the chip to shorten board traces.

- You can use extra pins and logic resources on the chip for functions unrelated to the processor. Extra resources can provide a few extra gates and registers as glue logic for the board design; or extra resources can implement entire systems. For example, a Nios II processor system consumes only 5% of a large Altera FPGA, leaving the rest of the chip’s resources available to implement other functions.
You can use extra pins and logic on the chip to implement additional peripherals for the Nios II processor system. Altera offers a library of peripherals that easily connect to Nios II processor systems.

**Configurable Soft Processor Core Concepts**

This section introduces Nios II concepts that are unique or different from other discrete microcontrollers. The concepts described in this section provide a foundation for understanding the rest of the features discussed in this handbook.

**Configurable Soft Processor Core**

The Nios II processor is a configurable soft IP core, as opposed to a fixed, off-the-shelf microcontroller. You can add or remove features on a system-by-system basis to meet performance or price goals. *Soft* means the processor core is not fixed in silicon and can be targeted to any Altera FPGA family.

You are not required to create a new Nios II processor configuration for every new design. Altera provides ready-made Nios II system designs that you can use as is. If these designs meet your system requirements, there is no need to configure the design further. In addition, you can use the Nios II instruction set simulator to begin writing and debugging Nios II applications before the final hardware configuration is determined.

**Flexible Peripheral Set and Address Map**

A flexible peripheral set is one of the most notable differences between Nios II processor systems and fixed microcontrollers. Because the Nios II processor is implemented in programmable logic, you can easily build made-to-order Nios II processor systems with the exact peripheral set required for the target applications.

A corollary of flexible peripherals is a flexible address map. Altera provides software constructs to access memory and peripherals generically, independently of address location. Therefore, the flexible peripheral set and address map does not affect application developers.

There are two broad classes of peripherals: standard peripherals and custom peripherals.

**Standard Peripherals**

Altera provides a set of peripherals commonly used in microcontrollers, such as timers, serial communication interfaces, general-purpose I/O, SDRAM controllers, and other memory interfaces. The list of available peripherals continues to increase as Altera and third-party vendors release new peripherals.

For information about the Altera-provided cores, refer to the *Embedded Peripherals IP User Guide*. 
**Custom Components**

You can also create custom components and integrate them in Nios II processor systems. For performance-critical systems that spend most CPU cycles executing a specific section of code, it is a common technique to create a custom peripheral that implements the same function in hardware. This approach offers a double performance benefit: the hardware implementation is faster than software; and the processor is free to perform other functions in parallel while the custom peripheral operates on data.

For information about creating custom components in Qsys, refer to the *Creating Qsys Components* chapter in volume 1 of the *Quartus II Handbook*.

For information about creating custom components in SOPC Builder, refer to the *SOPC Builder Components* and *Component Editor* chapters in the *SOPC Builder User Guide*.

**Custom Instructions**

Like custom peripherals, custom instructions allow you to increase system performance by augmenting the processor with custom hardware. You can achieve significant performance improvements, often on the order of 10 to 100 times, by implementing performance-critical operations in hardware using custom instruction logic.

The custom logic is integrated into the Nios II processor’s arithmetic logic unit (ALU). Similar to native Nios II instructions, custom instruction logic can take values from up to two source registers and optionally write back a result to a destination register.

Because the processor is implemented on reprogrammable Altera FPGAs, software and hardware engineers can work together to iteratively optimize the hardware and test the results of software running on hardware.

From the software perspective, custom instructions appear as machine-generated assembly macros or C functions, so programmers do not need to understand assembly language to use custom instructions.

For information about using SOPC Builder designs with custom instruction in Qsys, refer to the *SOPC Builder to Qsys Migration Guidelines*.

**Automated System Generation**

Altera’s Qsys and SOPC Builder system integration tools fully automate the process of configuring processor features and generating a hardware design that you program in an Altera device. The Qsys and SOPC Builder graphical user interfaces (GUI) enable you to configure Nios II processor systems with any number of peripherals and memory interfaces. You can create entire processor systems without performing any schematic or HDL design entry. Qsys and SOPC Builder can also import HDL design files, providing an easy mechanism to integrate custom logic in a Nios II processor system.
After system generation, you can download the design onto a board, and debug software executing on the board. To the software developer, the processor architecture of the design is set. Software development proceeds in the same manner as for traditional, nonconfigurable processors.

For information about using existing SOPC Builder designs in Qsys, refer to the SOPC Builder to Qsys Migration Guidelines.

**OpenCore Plus Evaluation**

You can evaluate the Nios II processor without a license. With Altera’s free OpenCore Plus evaluation feature, you can perform the following actions:

- Simulate the behavior of a Nios II processor within your system.
- Verify the functionality of your design, as well as evaluate its size and speed quickly and easily.
- Generate time-limited device programming files for designs that include Nios II processors.
- Program a device and verify your design in hardware.

You only need to purchase a license for the Nios II processor when you are completely satisfied with its functionality and performance, and want to take your design to production.

For more information about OpenCore Plus, refer to AN 320: OpenCore Plus Evaluation of Megafunctions.

**Document Revision History**

Table 1–1 lists the revision history for this document.

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>May 2011</td>
<td>11.0.0</td>
<td>Added references to new Qsys system integration tool.</td>
</tr>
<tr>
<td>December 2010</td>
<td>10.1.0</td>
<td>Maintenance release.</td>
</tr>
<tr>
<td>July 2010</td>
<td>10.0.0</td>
<td>Maintenance release.</td>
</tr>
<tr>
<td>November 2009</td>
<td>9.1.0</td>
<td>Added external interrupt controller interface information.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Added shadow register set information.</td>
</tr>
<tr>
<td>March 2009</td>
<td>9.0.0</td>
<td>Maintenance release.</td>
</tr>
<tr>
<td>November 2008</td>
<td>8.1.0</td>
<td>Maintenance release.</td>
</tr>
<tr>
<td>May 2008</td>
<td>8.0.0</td>
<td>Added MMU and MPU to bullet list of features.</td>
</tr>
<tr>
<td>October 2007</td>
<td>7.2.0</td>
<td>Added OpenCore Plus section.</td>
</tr>
<tr>
<td>May 2007</td>
<td>7.1.0</td>
<td>Added table of contents to Introduction section.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Added Referenced Documents section.</td>
</tr>
<tr>
<td>March 2007</td>
<td>7.0.0</td>
<td>Maintenance release.</td>
</tr>
<tr>
<td>November 2006</td>
<td>6.1.0</td>
<td>Maintenance release.</td>
</tr>
</tbody>
</table>
Table 1–1. Document Revision History (Part 2 of 2)

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>May 2006</td>
<td>6.0.0</td>
<td>- Added single precision floating-point and integration with SignalTap® II logic analyzer to features list.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Updated performance to 250 DMIPS.</td>
</tr>
<tr>
<td>October 2005</td>
<td>5.1.0</td>
<td>Maintenance release.</td>
</tr>
<tr>
<td>May 2005</td>
<td>5.0.0</td>
<td>Maintenance release.</td>
</tr>
<tr>
<td>September 2004</td>
<td>1.1</td>
<td>Maintenance release.</td>
</tr>
<tr>
<td>May 2004</td>
<td>1.0</td>
<td>Initial release.</td>
</tr>
</tbody>
</table>
This chapter describes the hardware structure of the Nios® II processor, including a discussion of all the functional units of the Nios II architecture and the fundamentals of the Nios II processor hardware implementation. This chapter contains the following sections:

- “Processor Implementation” on page 2–2
- “Register File” on page 2–3
- “Arithmetic Logic Unit” on page 2–4
- “Reset and Debug Signals” on page 2–8
- “Exception and Interrupt Controllers” on page 2–8
- “Memory and I/O Organization” on page 2–10
- “JTAG Debug Module” on page 2–17

The Nios II architecture describes an instruction set architecture (ISA). The ISA in turn necessitates a set of functional units that implement the instructions. A Nios II processor core is a hardware design that implements the Nios II instruction set and supports the functional units described in this document. The processor core does not include peripherals or the connection logic to the outside world. It includes only the circuits required to implement the Nios II architecture.

The Nios II architecture defines the following functional units:

- Register file
- Arithmetic logic unit (ALU)
- Interface to custom instruction logic
- Exception controller
- Internal or external interrupt controller
- Instruction bus
- Data bus
- Memory management unit (MMU)
- Memory protection unit (MPU)
- Instruction and data cache memories
- Tightly-coupled memory interfaces for instructions and data
- JTAG debug module
The following sections discuss hardware implementation details related to each functional unit.

**Processor Implementation**

The functional units of the Nios II architecture form the foundation for the Nios II instruction set. However, this does not indicate that any unit is implemented in hardware. The Nios II architecture describes an instruction set, not a particular hardware implementation. A functional unit can be implemented in hardware, emulated in software, or omitted entirely.

A Nios II implementation is a set of design choices embodied by a particular Nios II processor core. All implementations support the instruction set defined in the *Instruction Set Reference* chapter of the *Nios II Processor Reference Handbook*. Each implementation achieves specific objectives, such as smaller core size or higher performance. This flexibility allows the Nios II architecture to adapt to different target applications.
Implementation variables generally fit one of three trade-off patterns: more or less of a feature; inclusion or exclusion of a feature; hardware implementation or software emulation of a feature. An example of each trade-off follows:

- More or less of a feature—For example, to fine-tune performance, you can increase or decrease the amount of instruction cache memory. A larger cache increases execution speed of large programs, while a smaller cache conserves on-chip memory resources.

- Inclusion or exclusion of a feature—For example, to reduce cost, you can choose to omit the JTAG debug module. This decision conserves on-chip logic and memory resources, but it eliminates the ability to use a software debugger to debug applications.

- Hardware implementation or software emulation—For example, in control applications that rarely perform complex arithmetic, you can choose for the division instruction to be emulated in software. Removing the divide hardware conserves on-chip resources but increases the execution time of division operations.

For information about which Nios II cores supports what features, refer to the Nios II Core Implementation Details chapter of the Nios II Processor Reference Handbook. For complete details about user-selectable parameters for the Nios II processor, refer to the Instantiating the Nios II Processor chapter of the Nios II Processor Reference Handbook.

## Register File

The Nios II architecture supports a flat register file, consisting of thirty-two 32-bit general-purpose integer registers, and up to thirty-two 32-bit control registers. The architecture supports supervisor and user modes that allow system code to protect the control registers from errant applications.

The Nios II processor can optionally have one or more shadow register sets. A shadow register set is a complete set of Nios II general-purpose registers. When shadow register sets are implemented, the CRS field of the status register indicates which register set is currently in use. An instruction access to a general-purpose register uses whichever register set is active.

A typical use of shadow register sets is to accelerate context switching. When shadow register sets are implemented, the Nios II processor has two special instructions, `rdprs` and `wrprs`, for moving data between register sets. Shadow register sets are typically manipulated by an operating system kernel, and are transparent to application code. A Nios II processor can have up to 63 shadow register sets.

For details about shadow register set implementation and usage, refer to “Registers” and “Exception Processing” in the Programming Model chapter of the Nios II Processor Reference Handbook. For details about the `rdprs` and `wrprs` instructions, refer to the Instruction Set Reference chapter of the Nios II Processor Reference Handbook.

The Nios II architecture allows for the future addition of floating-point registers.
Arithmetic Logic Unit

The Nios II ALU operates on data stored in general-purpose registers. ALU operations take one or two inputs from registers, and store a result back in a register. The ALU supports the data operations described in Table 2–1.

Table 2–1. Operations Supported by the Nios II ALU

<table>
<thead>
<tr>
<th>Category</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic</td>
<td>The ALU supports addition, subtraction, multiplication, and division on signed and unsigned operands.</td>
</tr>
<tr>
<td>Relational</td>
<td>The ALU supports the equal, not-equal, greater-than-or-equal, and less-than relational operations (==, !=, &gt;=, &lt;) on signed and unsigned operands.</td>
</tr>
<tr>
<td>Logical</td>
<td>The ALU supports AND, OR, NOR, and XOR logical operations.</td>
</tr>
<tr>
<td>Shift and Rotate</td>
<td>The ALU supports shift and rotate operations, and can shift/rotate data by 0 to 31 bit positions per instruction. The ALU supports arithmetic shift right and logical shift right/left. The ALU supports rotate left/right.</td>
</tr>
</tbody>
</table>

To implement any other operation, software computes the result by performing a combination of the fundamental operations in Table 2–1.

Unimplemented Instructions

Some Nios II processor core implementations do not provide hardware to support the entire Nios II instruction set. In such a core, instructions without hardware support are known as unimplemented instructions.

The processor generates an exception whenever it issues an unimplemented instruction, so your exception handler can call a routine that emulates the operation in software. Unimplemented instructions do not affect the programmer’s view of the processor.

For a list of potential unimplemented instructions, refer to the Programming Model chapter of the Nios II Processor Reference Handbook.

Custom Instructions

The Nios II architecture supports user-defined custom instructions. The Nios II ALU connects directly to custom instruction logic, enabling you to implement operations in hardware that are accessed and used exactly like native instructions.

For more information, refer to the Nios II Custom Instruction User Guide and “Custom Instruction Tab” in the Instantiating the Nios II Processor chapter of the Nios II Processor Reference Handbook.
Floating-Point Instructions

The Nios II architecture supports single precision floating-point instructions as specified by the IEEE Std 754-1985. The basic set of floating-point custom instructions includes single precision floating-point addition, subtraction, and multiplication. Floating-point division is available as an extension to the basic instruction set. These floating-point instructions are implemented as custom instructions. Table 2–2 lists a detailed description of the conformance to IEEE 754-1985.

Table 2–2. Hardware Conformance with IEEE 754-1985 Floating-Point Standard

<table>
<thead>
<tr>
<th>Feature</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Operations (1)</strong></td>
<td></td>
</tr>
<tr>
<td>Addition</td>
<td>Implemented</td>
</tr>
<tr>
<td>Subtraction</td>
<td>Implemented</td>
</tr>
<tr>
<td>Multiplication</td>
<td>Implemented</td>
</tr>
<tr>
<td>Division</td>
<td>Optional</td>
</tr>
<tr>
<td><strong>Precision</strong></td>
<td></td>
</tr>
<tr>
<td>Single</td>
<td>Implemented</td>
</tr>
<tr>
<td>Double</td>
<td>Not implemented. Double precision operations are implemented in software.</td>
</tr>
<tr>
<td><strong>Exception conditions</strong></td>
<td></td>
</tr>
<tr>
<td>Invalid operation</td>
<td>Result is Not a Number (NaN)</td>
</tr>
<tr>
<td>Division by zero</td>
<td>Result is ±infinity</td>
</tr>
<tr>
<td>Overflow</td>
<td>Result is ±infinity</td>
</tr>
<tr>
<td>Inexact</td>
<td>Result is a normal number</td>
</tr>
<tr>
<td>Underflow</td>
<td>Result is ±0</td>
</tr>
<tr>
<td><strong>Rounding Modes</strong></td>
<td></td>
</tr>
<tr>
<td>Round to nearest</td>
<td>Implemented</td>
</tr>
<tr>
<td>Round toward zero</td>
<td>Not implemented</td>
</tr>
<tr>
<td>Round toward +infinity</td>
<td>Not implemented</td>
</tr>
<tr>
<td>Round toward –infinity</td>
<td>Not implemented</td>
</tr>
<tr>
<td><strong>NaN</strong></td>
<td></td>
</tr>
<tr>
<td>Quiet</td>
<td>Implemented</td>
</tr>
<tr>
<td>Signaling</td>
<td>Not implemented</td>
</tr>
<tr>
<td><strong>Subnormal (denormalized) numbers</strong></td>
<td>Subnormal operands are treated as zero. The floating-point custom instructions do not generate subnormal numbers.</td>
</tr>
<tr>
<td><strong>Software exceptions</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Not implemented. IEEE 754-1985 exception conditions are detected and handled as described elsewhere in this table.</td>
</tr>
<tr>
<td><strong>Status flags</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Not implemented. IEEE 754-1985 exception conditions are detected and handled as described elsewhere in this table.</td>
</tr>
</tbody>
</table>

Note to Table 2–2:

(1) The Nios II Embedded Design Suite (EDS) provides software implementations of primitive floating-point operations other than addition, subtraction, multiplication, and division. This includes operations such as floating-point conversions and comparisons. The software implementations of these primitives are 100% compliant with IEEE 754-1985.

You can add floating-point custom instructions to any Nios II processor design. The floating-point division hardware requires more resources than the other instructions. The Floating Point Hardware parameter editor allows you to omit the floating-point division hardware for cases in which code running on your hardware design does not make heavy use of floating-point division. When you omit the floating-point divide instruction, the Nios II compiler implements floating-point division in software.
In Qsys, the **Floating Point Hardware** component under **Custom Instruction Modules** on the **Component Library** tab contains the floating-point custom instructions.

To add floating-point custom instructions to your Nios II processor core in SOPC Builder, refer to “Custom Instruction Tab” in the **Instantiating the Nios II Processor** chapter of the **Nios II Processor Reference Handbook**.

The Nios II floating-point custom instructions are based on the Altera® floating-point megafunctions: ALTFP_MULT, ALTFP_ADD_SUB, and ALTFP_DIV.

For information about each individual floating-point megafunction, including acceleration factors and device resource usage, refer to the megafunction user guides, available on the **IP and Megafuctions** literature page of the Altera website.

The Nios II software development tools recognize C code that takes advantage of the floating-point instructions present in the processor core. When the floating-point custom instructions are present in your target hardware, the Nios II compiler compiles your code to use the custom instructions for floating-point operations, including addition, subtraction, multiplication, division and the newlib math library.

**Software Development Considerations**

The best choice for your hardware design depends on a balance among floating-point usage, hardware resource usage, and performance. While the floating-point custom instructions speed up floating-point arithmetic, they substantially add to the size of your hardware design. If resource usage is an issue, consider reworking your algorithms to minimize floating-point arithmetic.

You can use **#pragma** directives in your software to compare hardware and software implementations of the floating-point instructions. The following **#pragma** directives instruct the Nios II compiler to ignore the floating-point instructions and generate software implementations. The scope of these **#pragma** directives is the entire C file.

- **#pragma no_custom_fadds**—Forces software implementation of floating-point add
- **#pragma no_custom_fsubs**—Forces software implementation of floating-point subtract
- **#pragma no_custom_fmuls**—Forces software implementation of floating-point multiply
- **#pragma no_custom_fdivs**—Forces software implementation of floating-point divide

The Nios II instruction set simulator (ISS) in the Nios II integrated development environment (IDE) does not support custom instructions. If you need to run your software on the ISS, disable the floating-point custom instructions in software with the **#pragma** directives.

All the floating-point custom instructions are single-precision operations. Double-precision floating-point operations are implemented in software.
When the floating-point custom instructions are not present, the Nios II compiler treats floating-point constants as double-precision values. However, with the floating-point custom instructions, the Nios II compiler treats floating-point constants as single-precision numbers by default. This allows all floating-point expressions to be evaluated in hardware, at a possible cost in precision.

If you do not wish floating-point constants to be cast down to single precision values, append L to each constant value, to instruct the compiler to treat the constant as a double-precision floating-point value. In this case, if an expression contains a floating-point constant, each term in the expression is cast to double precision. As a result, the expression is computed with software-implemented double-precision arithmetic, at a possible cost in computation speed.

Table 2–3 describes how computation is implemented for code that multiplies single-precision variables with floating-point constants.

**Table 2–3. Floating-Point Constant Examples**

<table>
<thead>
<tr>
<th>Example Code</th>
<th>Floating-Point Custom Instructions Present?</th>
<th>Precision</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>b = a × 4.67</td>
<td>No</td>
<td>Double</td>
<td>Software</td>
</tr>
<tr>
<td>b = a × 4.67f</td>
<td>No</td>
<td>Single</td>
<td>Software</td>
</tr>
<tr>
<td>b = a × 4.67L</td>
<td>No</td>
<td>Double</td>
<td>Software</td>
</tr>
<tr>
<td>b = a × 4.67</td>
<td>Yes</td>
<td>Single</td>
<td>Hardware</td>
</tr>
<tr>
<td>b = a × 4.67f</td>
<td>Yes</td>
<td>Single</td>
<td>Hardware</td>
</tr>
<tr>
<td>b = a × 4.67L</td>
<td>Yes</td>
<td>Double</td>
<td>Software</td>
</tr>
</tbody>
</table>

With the GCC 4 compiler toolchain, precompiled libraries are compiled with double-precision floating-point constants. The behavior of precompiled floating-point library functions such as sin() and cos() is unaffected by the presence of the floating-point custom instructions.
Reset and Debug Signals

Table 2–4 describes the reset and debug signals that the Nios II processor core supports.

Table 2–4. Nios II Processor Debug and Reset Signals

<table>
<thead>
<tr>
<th>Signal Name</th>
<th>Type</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>reset</td>
<td>Reset</td>
<td>This is a global hardware reset signal that forces the processor core to reset immediately.</td>
</tr>
<tr>
<td>cpu_resetrequest</td>
<td>Reset</td>
<td>This is an optional, local reset signal that causes the processor to reset without affecting other components in the Nios II system. The processor finishes executing any instructions in the pipeline, and then enters the reset state. This process can take several clock cycles, so be sure to continue asserting the cpu_resetrequest signal until the processor core asserts a cpu_resettaken signal. The processor core asserts a cpu_resettaken signal for 1 cycle when the reset is complete and then periodically if cpu_resetrequest remains asserted. The processor remains in the reset state for as long as cpu_resetrequest is asserted. While the processor is in the reset state, it periodically reads from the reset address. It discards the result of the read, and remains in the reset state. The processor does not respond to cpu_resetrequest when the processor is under the control of the JTAG debug module, that is, when the processor is paused. The processor responds to the cpu_resetrequest signal if the signal is asserted when the JTAG debug module relinquishes control, both momentarily during each single step as well as when you resume execution.</td>
</tr>
<tr>
<td>debugreq</td>
<td>Debug</td>
<td>This is an optional signal that temporarily suspends the processor for debugging purposes. When you assert the signal, the processor pauses in the same manner as when a breakpoint is encountered, transfers execution to the routine located at the break address, and asserts a debugack signal. Asserting the debugreq signal when the processor is already paused has no effect.</td>
</tr>
</tbody>
</table>

For more information on adding reset signals to the Nios II processor, refer to “Advanced Features Tab” in the Instantiating the Nios II Processor chapter of the Nios II Processor Reference Handbook. For more information on the break vector and adding debug signals to the Nios II processor, refer to “JTAG Debug Module Tab” in the Instantiating the Nios II Processor chapter of the Nios II Processor Reference Handbook.

Exception and Interrupt Controllers

The Nios II processor includes hardware for handling exceptions, including hardware interrupts. It also includes an optional external interrupt controller (EIC) interface. The EIC interface enables you to speed up interrupt handling in a complex system by adding a custom interrupt controller.

Exception Controller

The Nios II architecture provides a simple, nonvectored exception controller to handle all exception types. Each exception, including internal hardware interrupts, causes the processor to transfer execution to an exception address. An exception handler at this address determines the cause of the exception and dispatches an appropriate exception routine.
Exception addresses are specified with the Nios II Processor parameter editor in Qsys and SOPC Builder.

All exceptions are precise. Precise means that the processor has completed execution of all instructions preceding the faulting instruction and not started execution of instructions following the faulting instruction. Precise exceptions allow the processor to resume program execution once the exception handler clears the exception.

**EIC Interface**

An EIC provides high performance hardware interrupts to reduce your program's interrupt latency. An EIC is typically used in conjunction with shadow register sets and when you need more than the 32 interrupts provided by the Nios II internal interrupt controller.

The Nios II processor connects to an EIC through the EIC interface. When an EIC is present, the internal interrupt controller is not implemented; Qsys and SOPC Builder connect interrupts to the EIC.

The EIC selects among active interrupts and presents one interrupt to the Nios II processor, with interrupt handler address and register set selection information. The interrupt selection algorithm is specific to the EIC implementation, and is typically based on interrupt priorities. The Nios II processor does not depend on any specific interrupt prioritization scheme in the EIC.

For every external interrupt, the EIC presents an interrupt level. The Nios II processor uses the interrupt level in determining when to service the interrupt.

Any external interrupt can be configured as an NMI. NMIs are not masked by the status.PIE bit, and have no interrupt level.

An EIC can be software-configurable.

When the EIC interface and shadow register sets are implemented on the Nios II core, you must ensure that your software is built with the Nios II EDS version 9.0 or higher. Earlier versions have an implementation of the _eret_ instruction that is incompatible with shadow register sets.

For a typical example of an EIC, refer to the _Vectored Interrupt Controller_ chapter in the *Embedded Peripherals IP User Guide*. For details about EIC usage, refer to “Exception Processing” in the *Programming Model* chapter of the *Nios II Processor Reference Handbook*.

**Internal Interrupt Controller**

The Nios II architecture supports 32 internal hardware interrupts. The processor core has 32 level-sensitive interrupt request (IRQ) inputs, _irq0_ through _irq31_, providing a unique input for each interrupt source. IRQ priority is determined by software. The architecture supports nested interrupts.

Your software can enable and disable any interrupt source individually through the _ienable_ control register, which contains an interrupt-enable bit for each of the IRQ inputs. Software can enable and disable interrupts globally using the _PIE_ bit of the _status_ control register. A hardware interrupt is generated if and only if all of the following conditions are true:
The PIE bit of the status register is 1
- An interrupt-request input, irq<n>, is asserted
- The corresponding bit n of the ienable register is 1

In SOPC Builder, the Nios II processor core offers an interrupt vector custom instruction, which accelerates interrupt vector dispatch to reduce your program’s interrupt latency. The interrupt vector custom instruction is less efficient than using the EIC interface with the Altera vectored interrupt controller component, and thus is deprecated in Qsys. Altera recommends using the EIC interface.

For information about adding the interrupt vector custom instruction to the Nios II processor in SOPC Builder, refer to “Custom Instruction Tab” in the Instantiating the Nios II Processor chapter of the Nios II Processor Reference Handbook.

Memory and I/O Organization

This section explains hardware implementation details of the Nios II memory and I/O organization. The discussion covers both general concepts true of all Nios II processor systems, as well as features that might change from system to system.

The flexible nature of the Nios II memory and I/O organization are the most notable difference between Nios II processor systems and traditional microcontrollers. Because Nios II processor systems are configurable, the memories and peripherals vary from system to system. As a result, the memory and I/O organization varies from system to system.

A Nios II core uses one or more of the following to provide memory and I/O access:
- Instruction master port—An Avalon® Memory-Mapped (Avalon-MM) master port that connects to instruction memory via system interconnect fabric
- Instruction cache—Fast cache memory internal to the Nios II core
- Data master port—An Avalon-MM master port that connects to data memory and peripherals via system interconnect fabric
- Data cache—Fast cache memory internal to the Nios II core
- Tightly-coupled instruction or data memory port—Interface to fast on-chip memory outside the Nios II core

The Nios II architecture handles the hardware details for the programmer, so programmers can develop Nios II applications without specific knowledge of the hardware implementation.

For details that affect programming issues, refer to the Programming Model chapter of the Nios II Processor Reference Handbook.
Figure 2–2 shows a diagram of the memory and I/O organization for a Nios II processor core.

**Instruction and Data Buses**

The Nios II architecture supports separate instruction and data buses, classifying it as a Harvard architecture. Both the instruction and data buses are implemented as Avalon-MM master ports that adhere to the Avalon-MM interface specification. The data master port connects to both memory and peripheral components, while the instruction master port connects only to memory components.

Refer to the *Avalon Interface Specifications* for details of the Avalon-MM interface.
Memory and Peripheral Access
The Nios II architecture provides memory-mapped I/O access. Both data memory and peripherals are mapped into the address space of the data master port. The Nios II architecture uses little-endian byte ordering. Words and halfwords are stored in memory with the more-significant bytes at higher addresses.

The Nios II architecture does not specify anything about the existence of memory and peripherals; the quantity, type, and connection of memory and peripherals are system-dependent. Typically, Nios II processor systems contain a mix of fast on-chip memory and slower off-chip memory. Peripherals typically reside on-chip, although interfaces to off-chip peripherals also exist.

Instruction Master Port
The Nios II instruction bus is implemented as a 32-bit Avalon-MM master port. The instruction master port performs a single function: it fetches instructions to be executed by the processor. The instruction master port does not perform any write operations.

The instruction master port is a pipelined Avalon-MM master port. Support for pipelined Avalon-MM transfers minimizes the impact of synchronous memory with pipeline latency and increases the overall fMAX of the system. The instruction master port can issue successive read requests before data has returned from prior requests. The Nios II processor can prefetch sequential instructions and perform branch prediction to keep the instruction pipe as active as possible.

The instruction master port always retrieves 32 bits of data. The instruction master port relies on dynamic bus-sizing logic contained in the system interconnect fabric. By virtue of dynamic bus sizing, every instruction fetch returns a full instruction word, regardless of the width of the target memory. Consequently, programs do not need to be aware of the widths of memory in the Nios II processor system.

The Nios II architecture supports on-chip cache memory for improving average instruction fetch performance when accessing slower memory. Refer to “Cache Memory” on page 2–13 for details. The Nios II architecture supports tightly-coupled memory, which provides guaranteed low-latency access to on-chip memory. Refer to “Tightly-Coupled Memory” on page 2–14 for details.

Data Master Port
The Nios II data bus is implemented as a 32-bit Avalon-MM master port. The data master port performs two functions:

- Read data from memory or a peripheral when the processor executes a load instruction
- Write data to memory or a peripheral when the processor executes a store instruction

Byte-enable signals on the master port specify which of the four byte-lane(s) to write during store operations. When the Nios II core is configured with a data cache line size greater than four bytes, the data master port supports pipelined Avalon-MM transfers. When the data cache line size is only four bytes, any memory pipeline latency is perceived by the data master port as wait states. Load and store operations can complete in a single clock cycle when the data master port is connected to zero-wait-state memory.
The Nios II architecture supports on-chip cache memory for improving average data transfer performance when accessing slower memory. Refer to “Cache Memory” on page 2–13 for details. The Nios II architecture supports tightly-coupled memory, which provides guaranteed low-latency access to on-chip memory. Refer to “Tightly-Coupled Memory” on page 2–14 for details.

Shared Memory for Instructions and Data

Usually the instruction and data master ports share a single memory that contains both instructions and data. While the processor core has separate instruction and data buses, the overall Nios II processor system might present a single, shared instruction/data bus to the outside world. The outside view of the Nios II processor system depends on the memory and peripherals in the system and the structure of the system interconnect fabric.

The data and instruction master ports never cause a gridlock condition in which one port starves the other. For highest performance, assign the data master port higher arbitration priority on any memory that is shared by both instruction and data master ports.

Cache Memory

The Nios II architecture supports cache memories on both the instruction master port (instruction cache) and the data master port (data cache). Cache memory resides on-chip as an integral part of the Nios II processor core. The cache memories can improve the average memory access time for Nios II processor systems that use slow off-chip memory such as SDRAM for program and data storage.

The instruction and data caches are enabled perpetually at run-time, but methods are provided for software to bypass the data cache so that peripheral accesses do not return cached data. Cache management and cache coherency are handled by software. The Nios II instruction set provides instructions for cache management.

Configurable Cache Memory Options

The cache memories are optional. The need for higher memory performance (and by association, the need for cache memory) is application dependent. Many applications require the smallest possible processor core, and can trade-off performance for size.

A Nios II processor core might include one, both, or neither of the cache memories. Furthermore, for cores that provide data and/or instruction cache, the sizes of the cache memories are user-configurable. The inclusion of cache memory does not affect the functionality of programs, but it does affect the speed at which the processor fetches instructions and reads/writes data.

Effective Use of Cache Memory

The effectiveness of cache memory to improve performance is based on the following premises:

- Regular memory is located off-chip, and access time is long compared to on-chip memory
- The largest, performance-critical instruction loop is smaller than the instruction cache
The largest block of performance-critical data is smaller than the data cache. Optimal cache configuration is application specific, although you can make decisions that are effective across a range of applications. For example, if a Nios II processor system includes only fast, on-chip memory (i.e., it never accesses slow, off-chip memory), an instruction or data cache is unlikely to offer any performance gain. As another example, if the critical loop of a program is 2 KB, but the size of the instruction cache is 1 KB, an instruction cache does not improve execution speed. In fact, an instruction cache may degrade performance in this situation.

If an application always requires certain data or sections of code to be located in cache memory for performance reasons, the tightly-coupled memory feature might provide a more appropriate solution. Refer to “Tightly-Coupled Memory” on page 2–14 for details.

**Cache Bypass Methods**

The Nios II architecture provides the following methods for bypassing the data cache:

- I/O load and store instructions
- Bit-31 cache bypass

**I/O Load and Store Instructions Method**

The load and store I/O instructions such as `ldio` and `stio` bypass the data cache and force an Avalon-MM data transfer to a specified address.

**The Bit-31 Cache Bypass Method**

The bit-31 cache bypass method on the data master port uses bit 31 of the address as a tag that indicates whether the processor should transfer data to/from cache, or bypass it. This is a convenience for software, which might need to cache certain addresses and bypass others. Software can pass addresses as parameters between functions, without having to specify any further information about whether the addressed data is cached or not.

To determine which cores implement which cache bypass methods, refer to the *Nios II Core Implementation Details* chapter of the *Nios II Processor Reference Handbook*.

**Tightly-Coupled Memory**

Tightly-coupled memory provides guaranteed low-latency memory access for performance-critical applications. Compared to cache memory, tightly-coupled memory provides the following benefits:

- Performance similar to cache memory
- Software can guarantee that performance-critical code or data is located in tightly-coupled memory
- No real-time caching overhead, such as loading, invalidating, or flushing memory
Physically, a tightly-coupled memory port is a separate master port on the Nios II processor core, similar to the instruction or data master port. A Nios II core can have zero, one, or multiple tightly-coupled memories. The Nios II architecture supports tightly-coupled memory for both instruction and data access. Each tightly-coupled memory port connects directly to exactly one memory with guaranteed low, fixed latency. The memory is external to the Nios II core and is located on chip.

**Accessing Tightly-Coupled Memory**

Tightly-coupled memories occupy normal address space, the same as other memory devices connected via system interconnect fabric. The address ranges for tightly-coupled memories (if any) are determined at system generation time.

Software accesses tightly-coupled memory using regular load and store instructions. From the software’s perspective, there is no difference accessing tightly-coupled memory compared to other memory.

**Effective Use of Tightly-Coupled Memory**

A system can use tightly-coupled memory to achieve maximum performance for accessing a specific section of code or data. For example, interrupt-intensive applications can place exception handler code into a tightly-coupled memory to minimize interrupt latency. Similarly, compute-intensive digital signal processing (DSP) applications can place data buffers into tightly-coupled memory for the fastest possible data access.

If the application’s memory requirements are small enough to fit entirely on chip, it is possible to use tightly-coupled memory exclusively for code and data. Larger applications must selectively choose what to include in tightly-coupled memory to maximize the cost-performance trade-off.

For additional tightly-coupled memory guidelines, refer to the *Using Tightly Coupled Memory with the Nios II Processor* tutorial.

**Address Map**

The address map for memories and peripherals in a Nios II processor system is design dependent. You specify the address map in Qsys and SOPC Builder.

There are three addresses that are part of the processor and deserve special mention:

- Reset address
- Exception address
- Break handler address

Programmers access memories and peripherals by using macros and drivers. Therefore, the flexible address map does not affect application developers.

**Memory Management Unit**

The optional Nios II MMU provides the following features and functionality:

- Virtual to physical address mapping
- Memory protection
32-bit virtual and physical addresses, mapping a 4-GB virtual address space into as much as 4 GB of physical memory

4-KB page and frame size

Low 512 MB of physical address space available for direct access

Hardware translation lookaside buffers (TLBs), accelerating address translation
- Separate TLBs for instruction and data accesses
- Read, write, and execute permissions controlled per page
- Default caching behavior controlled per page
- TLBs acting as $n$-way set-associative caches for software page tables
- TLB sizes and associativities configurable in the Nios II Processor parameter editor
- Format of page tables (or equivalent data structures) determined by system software
- Replacement policy for TLB entries determined by system software
- Write policy for TLB entries determined by system software

For more information about the MMU implementation, refer to the Programming Model chapter of the Nios II Processor Reference Handbook.

You can optionally include the MMU when you instantiate the Nios II processor in your Nios II hardware system. When present, the MMU is always enabled, and the data and instruction caches are virtually-indexed, physically-tagged caches. Several parameters are available, allowing you to optimize the MMU for your system needs.

For complete details about user-selectable parameters for the Nios II MMU, refer to the Instantiating the Nios II Processor chapter of the Nios II Processor Reference Handbook.

The Nios II MMU is optional and mutually exclusive from the Nios II MPU. Nios II systems can include either an MMU or MPU, but cannot include both an MMU and MPU on the same Nios II processor core.

**Memory Protection Unit**

The optional Nios II MPU provides the following features and functionality:
- Memory protection
- Up to 32 instruction regions and 32 data regions
- Variable instruction and data region sizes
- Amount of region memory defined by size or upper address limit
- Read and write access permissions for data regions
- Execute access permissions for instruction regions
- Overlapping regions
For more information about the MPU implementation, refer to the Programming Model chapter of the Nios II Processor Reference Handbook.

You can optionally include the MPU when you instantiate the Nios II processor in your Nios II hardware system. When present, the MPU is always enabled. Several parameters are available, allowing you to optimize the MPU for your system needs.

For complete details about user-selectable parameters for the Nios II MPU, refer to the Instantiating the Nios II Processor chapter of the Nios II Processor Reference Handbook.

The Nios II MPU is optional and mutually exclusive from the Nios II MMU. Nios II systems can include either an MPU or MMU, but cannot include both an MPU and MMU on the same Nios II processor core.

**JTAG Debug Module**

The Nios II architecture supports a JTAG debug module that provides on-chip emulation features to control the processor remotely from a host PC. PC-based software debugging tools communicate with the JTAG debug module and provide facilities, such as the following features:

- Downloading programs to memory
- Starting and stopping execution
- Setting breakpoints and watchpoints
- Analyzing registers and memory
- Collecting real-time execution trace data

The Nios II MMU does not support the JTAG debug module trace.

The debug module connects to the JTAG circuitry in an Altera FPGA. External debugging probes can then access the processor via the standard JTAG interface on the FPGA. On the processor side, the debug module connects to signals inside the processor core. The debug module has nonmaskable control over the processor, and does not require a software stub linked into the application under test. All system resources visible to the processor in supervisor mode are available to the debug module. For trace data collection, the debug module stores trace data in memory either on-chip or in the debug probe.

The debug module gains control of the processor either by asserting a hardware break signal, or by writing a break instruction into program memory to be executed. In both cases, the processor transfers execution to the routine located at the break address. The break address is specified with the Nios II Processor parameter editor in Qsys and SOPC Builder.

Soft processor cores such as the Nios II processor offer unique debug capabilities beyond the features of traditional, fixed processors. The soft nature of the Nios II processor allows you to debug a system in development using a full-featured debug core, and later remove the debug features to conserve logic resources. For the release version of a product, the JTAG debug module functionality can be reduced, or removed altogether.
The following sections describe the capabilities of the Nios II JTAG debug module hardware. The usage of all hardware features is dependent on host software, such as the Nios II Software Build Tools for Eclipse, which manages the connection to the target processor and controls the debug process.

**JTAG Target Connection**

The JTAG target connection provides the ability to connect to the processor through the standard JTAG pins on the Altera FPGA. This provides basic capabilities to start and stop the processor, and examine and edit registers and memory. The JTAG target connection is the minimum requirement for the Nios II flash programmer.

While the processor has no minimum clock frequency requirements, Altera recommends that your design’s system clock frequency be at least four times the JTAG clock frequency to ensure that the on-chip instrumentation (OCI) core functions properly.

**Download and Execute Software**

Downloading software refers to the ability to download executable code and data to the processor’s memory via the JTAG connection. After downloading software to memory, the JTAG debug module can then exit debug mode and transfer execution to the start of executable code.

**Software Breakpoints**

Software breakpoints allow you to set a breakpoint on instructions residing in RAM. The software breakpoint mechanism writes a break instruction into executable code stored in RAM. When the processor executes the break instruction, control is transferred to the JTAG debug module.

**Hardware Breakpoints**

Hardware breakpoints allow you to set a breakpoint on instructions residing in nonvolatile memory, such as flash memory. The hardware breakpoint mechanism continuously monitors the processor’s current instruction address. If the instruction address matches the hardware breakpoint address, the JTAG debug module takes control of the processor.

Hardware breakpoints are implemented using the JTAG debug module’s hardware trigger feature.

**Hardware Triggers**

Hardware triggers activate a debug action based on conditions on the instruction or data bus during real-time program execution. Triggers can do more than halt processor execution. For example, a trigger can be used to enable trace data collection during real-time processor execution.
Table 2–5 lists all the conditions that can cause a trigger. Hardware trigger conditions are based on either the instruction or data bus. Trigger conditions on the same bus can be logically ANDed, enabling the JTAG debug module to trigger, for example, only on write cycles to a specific address.

### Table 2–5. Trigger Conditions

<table>
<thead>
<tr>
<th>Condition</th>
<th>Bus</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Specific address</td>
<td>Data, Instruction</td>
<td>Trigger when the bus accesses a specific address.</td>
</tr>
<tr>
<td>Specific data value</td>
<td>Data</td>
<td>Trigger when a specific data value appears on the bus.</td>
</tr>
<tr>
<td>Read cycle</td>
<td>Data</td>
<td>Trigger on a read bus cycle.</td>
</tr>
<tr>
<td>Write cycle</td>
<td>Data</td>
<td>Trigger on a write bus cycle.</td>
</tr>
<tr>
<td>Armed</td>
<td>Data, Instruction</td>
<td>Trigger only after an armed trigger event. Refer to “Armed Triggers” on page 2–19.</td>
</tr>
<tr>
<td>Range</td>
<td>Data</td>
<td>Trigger on a range of address values, data values, or both. Refer to “Triggering on Ranges of Values” on page 2–19.</td>
</tr>
</tbody>
</table>

When a trigger condition occurs during processor execution, the JTAG debug module triggers an action, such as halting execution, or starting trace capture. Table 2–6 lists the trigger actions supported by the Nios II JTAG debug module.

### Table 2–6. Trigger Actions

<table>
<thead>
<tr>
<th>Action</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Break</td>
<td>Halt execution and transfer control to the JTAG debug module.</td>
</tr>
<tr>
<td>External trigger</td>
<td>Assert a trigger signal output. This trigger output can be used, for example, to trigger an external logic analyzer.</td>
</tr>
<tr>
<td>Trace on</td>
<td>Turn on trace collection.</td>
</tr>
<tr>
<td>Trace off</td>
<td>Turn off trace collection.</td>
</tr>
<tr>
<td>Trace sample (1)</td>
<td>Store one sample of the bus to trace buffer.</td>
</tr>
<tr>
<td>Arm</td>
<td>Enable an armed trigger.</td>
</tr>
</tbody>
</table>

**Note to Table 2–6:**

(1) Only conditions on the data bus can trigger this action.

### Armed Triggers

The JTAG debug module provides a two-level trigger capability, called armed triggers. Armed triggers enable the JTAG debug module to trigger on event B, only after event A. In this example, event A causes a trigger action that enables the trigger for event B.

### Triggering on Ranges of Values

The JTAG debug module can trigger on ranges of data or address values on the data bus. This mechanism uses two hardware triggers together to create a trigger condition that activates on a range of values within a specified range.
Trace Capture

Trace capture refers to ability to record the instruction-by-instruction execution of the processor as it executes code in real-time. The JTAG debug module offers the following trace features:

- Capture execution trace (instruction bus cycles).
- Capture data trace (data bus cycles).
- For each data bus cycle, capture address, data, or both.
- Start and stop capturing trace in real time, based on triggers.
- Manually start and stop trace under host control.
- Optionally stop capturing trace when trace buffer is full, leaving the processor executing.
- Store trace data in on-chip memory buffer in the JTAG debug module. (This memory is accessible only through the JTAG connection.)
- Store trace data to larger buffers in an off-chip debug probe.

Certain trace features require additional licensing or debug tools from third-party debug providers. For example, an on-chip trace buffer is a standard feature of the Nios II processor, but using an off-chip trace buffer requires additional debug software and hardware provided by MIPS Technologies or Lauterbach GmbH.

For details, search for “Nios II” on the MIPS Technologies website (www.mips.com) and the Lauterbach GmbH website (www.lauterbach.com).

Execution vs. Data Trace

The JTAG debug module supports tracing the instruction bus (execution trace), the data bus (data trace), or both simultaneously. Execution trace records only the addresses of the instructions executed, enabling you to analyze where in memory (that is, in which functions) code executed. Data trace records the data associated with each load and store operation on the data bus.

The JTAG debug module can filter the data bus trace in real time to capture the following:

- Load addresses only
- Store addresses only
- Both load and store addresses
- Load data only
- Load address and data
- Store address and data
- Address and data for both loads and stores
- Single sample of the data bus upon trigger event
Trace Frames
A frame is a unit of memory allocated for collecting trace data. However, a frame is not an absolute measure of the trace depth.

To keep pace with the processor executing in real time, execution trace is optimized to store only selected addresses, such as branches, calls, traps, and interrupts. From these addresses, host-side debug software can later reconstruct an exact instruction-by-instruction execution trace. Furthermore, execution trace data is stored in a compressed format, such that one frame represents more than one instruction. As a result of these optimizations, the actual start and stop points for trace collection during execution might vary slightly from the user-specified start and stop points.

Data trace stores 100% of requested loads and stores to the trace buffer in real time. When storing to the trace buffer, data trace frames have lower priority than execution trace frames. Therefore, while data frames are always stored in chronological order, execution and data trace are not guaranteed to be exactly synchronized with each other.

Document Revision History
Table 2–7 lists the revision history for this document.

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>May 2011</td>
<td>11.0.0</td>
<td>■ Added references to new Qsys system integration tool.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>■ Moved interrupt vector custom instruction information to the Instantiating the Nios II Processor chapter.</td>
</tr>
<tr>
<td>December 2010</td>
<td>10.1.0</td>
<td>Added reference to tightly-coupled memory tutorial.</td>
</tr>
<tr>
<td>July 2010</td>
<td>10.0.0</td>
<td>Maintenance release.</td>
</tr>
<tr>
<td>November 2009</td>
<td>9.1.0</td>
<td>■ Added external interrupt controller interface information.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>■ Added shadow register set information.</td>
</tr>
<tr>
<td>March 2009</td>
<td>9.0.0</td>
<td>Maintenance release.</td>
</tr>
<tr>
<td>November 2008</td>
<td>8.1.0</td>
<td>■ Expanded floating-point instructions information.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>■ Updated description of optional cpu_resetrequest and cpu_resettaken signals.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>■ Added description of optional debugreq and debugack signals.</td>
</tr>
<tr>
<td>May 2008</td>
<td>8.0.0</td>
<td>Added MMU and MPU sections.</td>
</tr>
<tr>
<td>October 2007</td>
<td>7.2.0</td>
<td>Maintenance release.</td>
</tr>
<tr>
<td>May 2007</td>
<td>7.1.0</td>
<td>■ Added table of contents to Introduction section.</td>
</tr>
<tr>
<td>March 2007</td>
<td>7.0.0</td>
<td>■ Added Referenced Documents section.</td>
</tr>
<tr>
<td>November 2006</td>
<td>6.1.0</td>
<td>Maintenance release.</td>
</tr>
<tr>
<td>May 2006</td>
<td>6.0.0</td>
<td>■ Added description of optional cpu_resetrequest and cpu_resettaken.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>■ Added section on single precision floating-point instructions.</td>
</tr>
<tr>
<td>October 2005</td>
<td>5.1.0</td>
<td>Maintenance release.</td>
</tr>
<tr>
<td>May 2005</td>
<td>5.0.0</td>
<td>Added tightly-coupled memory.</td>
</tr>
<tr>
<td>December 2004</td>
<td>1.2</td>
<td>Added new control register ctl15.</td>
</tr>
</tbody>
</table>
### Table 2-7. Document Revision History (Part 2 of 2)

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>September 2004</td>
<td>1.1</td>
<td>Updates for Nios II 1.01 release.</td>
</tr>
<tr>
<td>May 2004</td>
<td>1.0</td>
<td>Initial release.</td>
</tr>
</tbody>
</table>
3. Programming Model

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 these 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 3–1—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.
- Registers, page 3–10—Describes the Nios II register sets.
- Working With the MPU, page 3–29—Provides an overview of MPU initialization and operation.
- Exception processing, page 3–30—Describes how the Nios II processor responds to exceptions.
- Memory and Peripheral Access, page 3–53—Describes Nios II addressing.
- Instruction set categories, page 3–55—Introduces the Nios II instruction set.

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 ignore 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
Chapter 3: Programming Model
Operating Modes

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 the Altera® hardware abstraction library (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-KB pages and physical memory into 4-KB 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

<table>
<thead>
<tr>
<th>Position</th>
<th>Virtual Page Number</th>
<th>Page Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>31-0</td>
<td>29-22</td>
<td>23-12</td>
</tr>
<tr>
<td></td>
<td>21</td>
<td>20-11</td>
</tr>
</tbody>
</table>

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 an exception. The Nios II processor’s 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-GB virtual address space, and is capable of addressing up to 4 GB of physical memory.

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

### Virtual Memory Address Space

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

### Table 3–2. Virtual Memory Partitions (Part 1 of 2)

<table>
<thead>
<tr>
<th>Partition</th>
<th>Virtual Address Range</th>
<th>Used By</th>
<th>Memory Access</th>
<th>User Mode Access</th>
<th>Default Data Cacheability</th>
</tr>
</thead>
<tbody>
<tr>
<td>I/O (1)</td>
<td>0xE0000000–0xFFFFFFFF</td>
<td>Operating system</td>
<td>Bypasses TLB</td>
<td>No</td>
<td>Disabled</td>
</tr>
<tr>
<td>Kernel (1)</td>
<td>0xC0000000–0xDFFFFFFF</td>
<td>Operating system</td>
<td>Bypasses TLB</td>
<td>No</td>
<td>Enabled</td>
</tr>
<tr>
<td>Kernel MMU (1)</td>
<td>0x80000000–0xBFFFFFFF</td>
<td>Operating system</td>
<td>Uses TLB</td>
<td>No</td>
<td>Set by TLB</td>
</tr>
</tbody>
</table>
Each partition has a specific size, purpose, and relationship to the TLB:

- The 512-MB I/O partition provides access to peripherals.
- The 512-MB kernel partition provides space for the operating system kernel.
- The 1-GB kernel MMU partition is used by the TLB miss handler and kernel processes.
- The 2-GB 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-GB physical memory is divided into low memory and high memory. The lowest ½ GB of physical address space is low memory. The upper 3½ GB of physical address space is high memory. Figure 3–1 shows how physical memory is divided.

![Figure 3–1. Division of Physical Memory](image)

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. The Nios II Processor parameter editor in Qsys and SOPC Builder prevent you from choosing an address outside of low physical memory.
**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 described 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 noncacheable, 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 with the Nios II Processor parameter editor in Qsys and SOPC Builder. By default, the TLB is a 16-way cache. The default number of entries depends on the target device, as follows:

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

For more information, refer to the *Instantiating the Nios II Processor* 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 information about 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**

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VPN</td>
<td>VPN is the virtual page number field. This field is compared with the top 20 bits of the virtual address.</td>
</tr>
<tr>
<td>PID</td>
<td>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 in the Nios_II Processor parameter editor, and can be between 8 and 14 bits.</td>
</tr>
<tr>
<td>G</td>
<td>G is the global flag. When ( G = 1 ), the PID is ignored in the TLB lookup.</td>
</tr>
</tbody>
</table>

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**

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PFN</td>
<td>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.</td>
</tr>
<tr>
<td>C</td>
<td>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.</td>
</tr>
<tr>
<td>R</td>
<td>R is the readable flag. Allows load instructions to read a page.</td>
</tr>
<tr>
<td>W</td>
<td>W is the writable flag. Allows store instructions to write a page.</td>
</tr>
<tr>
<td>X</td>
<td>X is the executable flag. Allows instruction fetches from a page.</td>
</tr>
</tbody>
</table>

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 shown in Example 3–1.

**Example 3–1. TLB Lookup Algorithm for Instruction Fetches**

```c
if (VPN match && (G == 1 || 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
```

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.
The TLB lookup algorithm for data accesses is shown in Example 3–2.

Example 3–2. TLB Lookup Algorithm for Data Access Operations

```c
if (VPN match && (G == 1 || PID match))
  if ((load && R == 1) || (store && W == 1) || 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
```

Refer to “Instruction-Related Exceptions” on page 3–39 for information about TLB exceptions.

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 an 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-KB region must have a base address that is a multiple of 4 KB. 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

A Qsys and 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 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 3–53 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, growing from opposite ends of the address range. 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 3–29 for more information.

Registers

The Nios II register set includes general-purpose registers and control registers. In addition, the Nios II/f core can optionally have shadow register sets. This section discusses each register type.

General-purpose Registers

The Nios II architecture provides thirty-two 32-bit general-purpose registers, `r0` through `r31`, as described 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 the `call`, `callr`, 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>
<thead>
<tr>
<th>Register</th>
<th>Name</th>
<th>Function</th>
<th>Register</th>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>r0</td>
<td>zero</td>
<td>0x00000000</td>
<td>r16</td>
<td></td>
<td>Callee-saved register</td>
</tr>
<tr>
<td>r1</td>
<td>at</td>
<td>Assembler temporary</td>
<td>r17</td>
<td></td>
<td>Callee-saved register</td>
</tr>
<tr>
<td>r2</td>
<td></td>
<td>Return value</td>
<td>r18</td>
<td></td>
<td>Callee-saved register</td>
</tr>
<tr>
<td>r3</td>
<td></td>
<td>Return value</td>
<td>r19</td>
<td></td>
<td>Callee-saved register</td>
</tr>
</tbody>
</table>
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 lists details of the defined control registers. All nonreserved control registers have names recognized by the assembler.

### Table 3–6. Control Register Names and Bits (Part 1 of 2)

<table>
<thead>
<tr>
<th>Register</th>
<th>Name</th>
<th>Register Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>status</td>
<td>Refer to Table 3–7 on page 3–12</td>
</tr>
<tr>
<td>1</td>
<td>estatus</td>
<td>Refer to Table 3–9 on page 3–14</td>
</tr>
<tr>
<td>2</td>
<td>bstatus</td>
<td>Refer to Table 3–10 on page 3–15</td>
</tr>
<tr>
<td>3</td>
<td>ienable</td>
<td>Internal interrupt-enable bits (3)</td>
</tr>
<tr>
<td>4</td>
<td>ipending</td>
<td>Pending internal interrupt bits (3)</td>
</tr>
<tr>
<td>5</td>
<td>cpuid</td>
<td>Unique processor identifier</td>
</tr>
<tr>
<td>6</td>
<td>Reserved</td>
<td>Reserved</td>
</tr>
<tr>
<td>7</td>
<td>exception</td>
<td>Refer to Table 3–12 on page 3–16</td>
</tr>
</tbody>
</table>
The status Register

The value in the status register determines the state of the Nios II processor. All status bits are set to predefined values at processor reset. Some bits are exclusively used by and available only to certain features of the processor, such as the MMU, MPU or external interrupt controller (EIC) interface. Table 3–7 shows the layout of the status register.

Table 3–7. status Control Register Fields

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
<th>Access</th>
<th>Reset</th>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td>RSIE</td>
<td>RSIE is the register set interrupt-enable bit. When set to 1, this bit allows the processor to service external interrupts requesting the register set that is currently in use. When set to 0, this bit disallows servicing of such interrupts.</td>
<td>Read/Write</td>
<td>1</td>
<td>EIC interface and shadow register sets only (4)</td>
</tr>
<tr>
<td>NMI</td>
<td>NMI is the nonmaskable interrupt mode bit. The processor sets NMI to 1 when it takes a nonmaskable interrupt.</td>
<td>Read</td>
<td>0</td>
<td>EIC interface only (3)</td>
</tr>
</tbody>
</table>

Table 3–8 describes the fields defined in the status register.
### Registers

**(PRS)**

The processor copies the **CRS** field to the **PRS** field upon one of the following events:
- In a processor with no MMU, on any exception
- In a processor with an MMU, on one of the following:
  - Break exception
  - Nonbreak exception when **status**.**EH** is zero

The processor copies **CRS** to **PRS** immediately after copying the **status** register to **estatus**, **bstatus** or **sstatus**.

The number of significant bits in the **CRS** and **PRS** fields depends on the number of shadow register sets implemented in the Nios II core. The value of **CRS** and **PRS** can range from 0 to \( n - 1 \), where \( n \) is the number of implemented register sets. The processor core implements the number of significant bits needed to represent \( n - 1 \). Unused high-order bits are always read as 0, and must be written as 0.

Ensure that system software writes only valid register set numbers to the **PRS** field. Processor behavior is undefined with an unimplemented register set number.

**Access:** Read/Write | **Reset:** 0 | **Available:** shadow register sets only

---

**(CRS)**

**CRS** is the current register set field. **CRS** indicates which register set is currently in use. Register set 0 is the normal register set, while register sets 1 and higher are shadow register sets. The processor sets **CRS** to zero on any noninterrupt exception.

The number of significant bits in the **CRS** and **PRS** fields depends on the number of shadow register sets implemented in the Nios II core. Unused high-order bits are always read as 0, and must be written as 0.

**Access:** Read (1) | **Reset:** 0 | **Available:** shadow register sets only

---

**(IL)**

**IL** is the interrupt level field. The **IL** field controls what level of external maskable interrupts can be serviced. The processor services a maskable interrupt only if its requested interrupt level is greater than **IL**.

**Access:** Read/Write | **Reset:** 0 | **Available:** EIC interface only

---

**(IH)**

**IH** is the interrupt handler mode bit. The processor sets **IH** to one when it takes an external interrupt.

**Access:** Read/Write | **Reset:** 0 | **Available:** EIC interface only

---

**(EH)**

**EH** is the exception handler mode 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.

**Access:** Read/Write | **Reset:** 0 | **Available:** MMU only

---

**(U)**

**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.

**Access:** Read/Write | **Reset:** 0 | **Available:** MMU or MPU only

---

### Table 3–8. status Control Register Field Descriptions (Part 2 of 3)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
<th>Access</th>
<th>Reset</th>
<th>Available</th>
</tr>
</thead>
</table>
| **PRS** | **PRS** is the previous register set field. The processor copies the **CRS** field to the **PRS** field upon one of the following events:  
  - In a processor with no MMU, on any exception  
  - In a processor with an MMU, on one of the following:  
    - Break exception  
    - Nonbreak exception when **status**.**EH** is zero  
  
The processor copies **CRS** to **PRS** immediately after copying the **status** register to **estatus**, **bstatus** or **sstatus**.  
The number of significant bits in the **CRS** and **PRS** fields depends on the number of shadow register sets implemented in the Nios II core. The value of **CRS** and **PRS** can range from 0 to \( n - 1 \), where \( n \) is the number of implemented register sets. The processor core implements the number of significant bits needed to represent \( n - 1 \). Unused high-order bits are always read as 0, and must be written as 0.  
  
  Ensure that system software writes only valid register set numbers to the **PRS** field. Processor behavior is undefined with an unimplemented register set number. | Read/Write | 0 | Shadow register sets only (3) |
| **CRS** | **CRS** is the current register set field. **CRS** indicates which register set is currently in use. Register set 0 is the normal register set, while register sets 1 and higher are shadow register sets. The processor sets **CRS** to zero on any noninterrupt exception.  
The number of significant bits in the **CRS** and **PRS** fields depends on the number of shadow register sets implemented in the Nios II core. Unused high-order bits are always read as 0, and must be written as 0. | Read (1) | 0 | Shadow register sets only (3) |
| **IL** | **IL** is the interrupt level field. The **IL** field controls what level of external maskable interrupts can be serviced. The processor services a maskable interrupt only if its requested interrupt level is greater than **IL**. | Read/Write | 0 | EIC interface only (3) |
| **IH** | **IH** is the interrupt handler mode bit. The processor sets **IH** to one when it takes an external interrupt. | Read/Write | 0 | EIC interface only (3) |
| **EH** | **EH** is the exception handler mode 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 (3) |
| **U** | **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 (3) |
The estatus Register

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

All fields in the estatus register have read/write access. All fields reset to 0.

Table 3–8 describes the details of the fields defined in the estatus register.

When the Nios II processor takes an interrupt, if status.eh is zero (that is, the MMU is in nonexception mode), the processor copies the contents of the status register to estatus.

If shadow register sets are implemented, and the interrupt requests a shadow register set, the Nios II processor copies status to sstatus, not to estatus.

For details about the sstatus register, refer to “The sstatus Register” on page 3–27.

The exception handler can examine estatus to determine the pre-exception status of the processor. When returning from an exception, the eret instruction restores the pre-exception value of status. The instruction restores the pre-exception value by copying either estatus or sstatus back to status, depending on the value of status.CRS.

Refer to “Exception Processing” on page 3–30 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 bstatus 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 | RSIE | NMI | PRS | CRS | IL | IH | EH | U | PIE |

All fields in the bstatus register have read/write access. All fields reset to 0.

Table 3–8 describes the details of the fields defined in the bstatus 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 3–35 for more information.

The ienable Register

The ienable register controls the handling of internal 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 3–30 for more information.

When the internal interrupt controller is not implemented, the value of the ienable register is always 0.

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 you define in the Nios II Processor parameter editor to uniquely identify each processor in a multiprocessor system. In SOPC Builder, the value can be auto-assigned at system generation time and guaranteed to be unique for each processor in the system. In Qsys, unique values must be assigned manually. 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 Processor parameter editor gives you the option to have the processor provide the extra exception information.

For information about controlling the extra exception information option, refer to the Instantiating the Nios II Processor chapter of the Nios II Processor Reference Handbook.

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

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Access</th>
<th>Reset</th>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAUSE</td>
<td>CAUSE is written by the Nios II processor when certain exceptions occur. CAUSE contains a code for the highest-priority exception occurring at the time. The Cause column in Table 3–33 on page 3–32 lists the CAUSE field value for each exception. CAUSE is not written on a break or an external interrupt.</td>
<td>Read</td>
<td>0</td>
<td>Only with extra exception information</td>
</tr>
</tbody>
</table>

Table 3–12 describes the fields defined in the exception register.

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

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 describes the fields defined in the tlbacc register.

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).

Table 3–16. tlbacc Control Register Field Descriptions

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Access</th>
<th>Reset</th>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td>IG</td>
<td>IG is ignored by hardware and available to hold operating system specific information. Read as zero but can be written as nonzero.</td>
<td>Read/Write</td>
<td>0</td>
<td>Only with MMU</td>
</tr>
<tr>
<td>C</td>
<td>C is the data cacheable flag. When C = 0, data accesses are uncachable. When C = 1, data accesses are cacheable.</td>
<td>Read/Write</td>
<td>0</td>
<td>Only with MMU</td>
</tr>
<tr>
<td>R</td>
<td>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.</td>
<td>Read/Write</td>
<td>0</td>
<td>Only with MMU</td>
</tr>
<tr>
<td>W</td>
<td>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.</td>
<td>Read/Write</td>
<td>0</td>
<td>Only with MMU</td>
</tr>
<tr>
<td>X</td>
<td>X is the executable flag. When X = 0, instructions are not allowed to execute. When X = 1, instructions are allowed to execute.</td>
<td>Read/Write</td>
<td>0</td>
<td>Only with MMU</td>
</tr>
<tr>
<td>G</td>
<td>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.</td>
<td>Read/Write</td>
<td>0</td>
<td>Only with MMU</td>
</tr>
<tr>
<td>PFN</td>
<td>PFN is the physical frame number field. All unused upper bits must be zero.</td>
<td>Read/Write</td>
<td>0</td>
<td>Only with MMU</td>
</tr>
</tbody>
</table>
The \texttt{tlbacc} register format is the recommended format for entries in the operating system page table. The IG bits are ignored by the hardware on \texttt{wrctl} to \texttt{tlbacc} and read back as zero on \texttt{rdctl} from \texttt{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.

\textbf{The tlbmisc Register}

The \texttt{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 \texttt{tlbmisc} register.

\begin{table}[h]
\centering
\caption{tlbmisc Control Register Fields}
\begin{tabular}{ccccccccccccccc}
  \hline
  Field & Access & Description & Reset & Available \\
  \hline
  WAY & Read/Write & The \texttt{WAY} field controls the mapping from the VPN to a particular TLB entry. & 0 & Only with MMU \\
  RD & Write & \texttt{RD} is the read flag. Setting \texttt{RD} to one triggers a TLB read operation. & 0 & Only with MMU \\
  WE & Read/Write & \texttt{WE} is the TLB write enable flag. When \texttt{WE} = 1, a write to \texttt{tlbacc} writes through to a TLB entry. & 0 & Only with MMU \\
  PID & Read/Write & \texttt{PID} is the process identifier field. & 0 & Only with MMU \\
  DBL (1) & Read & DBL is the double TLB miss exception flag. & 0 & Only with MMU \\
  BAD (1) & Read & BAD is the bad virtual address exception flag. & 0 & Only with MMU \\
  PERM (1) & Read & PERM is the TLB permission violation exception flag. & 0 & Only with MMU \\
  D & Read & D is the data access exception flag. When \texttt{D} = 1, the exception is a data access exception. When \texttt{D} = 0, the exception is an instruction access exception. & 0 & Only with MMU \\
  \hline
\end{tabular}
\end{table}

\textbf{Note to Table 3–17:}

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

Table 3–18 describes the fields defined in the \texttt{tlbmisc} register.

\begin{table}[h]
\centering
\caption{tlbmisc Control Register Field Descriptions}
\begin{tabular}{cccc}
  \hline
  Field & Description & Access & Available \\
  \hline
  WAY & The \texttt{WAY} field controls the mapping from the VPN to a particular TLB entry. & Read/Write & Only with MMU \\
  RD & \texttt{RD} is the read flag. Setting \texttt{RD} to one triggers a TLB read operation. & Write & Only with MMU \\
  WE & \texttt{WE} is the TLB write enable flag. When \texttt{WE} = 1, a write to \texttt{tlbacc} writes through to a TLB entry. & Read/Write & Only with MMU \\
  PID & \texttt{PID} is the process identifier field. & Read/Write & Only with MMU \\
  DBL & DBL is the double TLB miss exception flag. & Read & Only with MMU \\
  BAD & BAD is the bad virtual address exception flag. & Read & Only with MMU \\
  PERM & PERM is the TLB permission violation exception flag. & Read & Only with MMU \\
  D & D is the data access exception flag. When \texttt{D} = 1, the exception is a data access exception. When \texttt{D} = 0, the exception is an instruction access exception. & Read & Only with MMU \\
  \hline
\end{tabular}
\end{table}

\textbf{Note to Table 3–18:}

(1) You can also use \texttt{exception.CAUSE} to determine these exceptions.

The following sections provide more information about the \texttt{tlbmisc} fields.

\textbf{The RD Flag}

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

- The tag portion of \texttt{pteaddr.VPN}
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 Qsys and 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 configurable in Qsys and 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 3–33 on page 3–32 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 nonbreak 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 Processor parameter editor gives you the option to have the processor provide the extra exception information.

For information about controlling the extra exception information option, refer to the Instantiating the Nios II Processor chapter of the Nios II Processor Reference Handbook.
When the option for extra exception information is enabled and a processor exception occurs, the \texttt{badaddr} register contains the byte instruction or data address associated with certain exceptions at the time the exception occurred. Table 3–33 on page 3–32 lists which exceptions write the \texttt{badaddr} register along with the value written. Table 3–19 shows the layout of the \texttt{badaddr} register.

Table 3–19. \texttt{badaddr} Control Register Fields

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Access</th>
<th>Reset</th>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td>BADDR</td>
<td>\begin{itemize} \item \texttt{Badaddr} contains the byte instruction address or data address associated with an exception when certain exceptions occur. The Address column of Table 3–33 on page 3–32 lists which exceptions write the \texttt{BADDR} field. \end{itemize}</td>
<td>Read</td>
<td>0</td>
<td>Only with extra exception information</td>
</tr>
</tbody>
</table>

Table 3–20 describes the fields defined in the \texttt{badaddr} register.

Table 3–20. \texttt{badaddr} Control Register Field Descriptions

The \texttt{BADDR} field allows up to a 32-bit instruction address or data address. If an MMU or MPU is present, the \texttt{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 \texttt{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 \texttt{BADDR} field.

The \texttt{config} Register

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

Table 3–21. \texttt{config} Control Register Fields
Table 3–22 describes the fields defined in the config register.

### Table 3–22. config Control Register Field Descriptions

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Access</th>
<th>Reset</th>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td>PE</td>
<td>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.</td>
<td>Read/Write</td>
<td>0</td>
<td>Only with MPU</td>
</tr>
<tr>
<td>ANI</td>
<td>ANI is the automatic nested interrupt mode bit. If ANI is set to zero, the processor clears status.PIE on each interrupt, disabling fast nested interrupts. If ANI is set to one, the processor keeps status.PIE set to one at the time of an interrupt, enabling fast nested interrupts. If the EIC interface and shadow register sets are not implemented in the Nios II core, ANI always reads as zero, disabling fast nested interrupts.</td>
<td>Read/Write</td>
<td>0</td>
<td>Only with the EIC interface and shadow register sets</td>
</tr>
</tbody>
</table>

### 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) |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |

Notes to Table 3–23:

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

Table 3–24 describes the fields defined in the mpubase register.

### Table 3–24. mpubase Control Register Field Descriptions

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

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 Qsys and 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 3–29 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.

A Qsys and 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 | WE |

Note to Table 3–25:
(1) This field size is variable. Unused upper bits and unused lower bits must be 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 | WE |

Note to Table 3–26:
(1) This field size is variable. Unused upper bits and unused lower bits must be written as zero.

Table 3–27 describes the fields defined in the mpuacc register.

Table 3–27. mpuacc Control Register Field Descriptions (Part 1 of 2)

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Access</th>
<th>Reset</th>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td>MASK (1)</td>
<td>MASK specifies the size of the region.</td>
<td>Read/Write</td>
<td>0</td>
<td>Only with MPU</td>
</tr>
<tr>
<td>LIMIT (1)</td>
<td>LIMIT specifies the upper address limit of the region.</td>
<td>Read/Write</td>
<td>0</td>
<td>Only with MPU</td>
</tr>
<tr>
<td>C</td>
<td>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.</td>
<td>Read/Write</td>
<td>0</td>
<td>Only with MPU</td>
</tr>
<tr>
<td>PERM</td>
<td>PERM specifies the access permissions for the region.</td>
<td>Read/Write</td>
<td>0</td>
<td>Only with MPU</td>
</tr>
</tbody>
</table>
The following sections provide more information about 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 lists the MASK field encodings for all possible region sizes in a full 31-bit byte address space.

**Table 3–28. MASK Region Size Encodings (Part 1 of 2)**

<table>
<thead>
<tr>
<th>MASK Encoding</th>
<th>Region Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1FFFFFFF</td>
<td>64 bytes</td>
</tr>
<tr>
<td>0x1FFFFFE</td>
<td>128 bytes</td>
</tr>
<tr>
<td>0x1FFFFFC</td>
<td>256 bytes</td>
</tr>
<tr>
<td>0x1FFFFF8</td>
<td>512 bytes</td>
</tr>
<tr>
<td>0x1FFFFF0</td>
<td>1 KB</td>
</tr>
<tr>
<td>0x1FFFFF0</td>
<td>2 KB</td>
</tr>
<tr>
<td>0x1FFFFC0</td>
<td>4 KB</td>
</tr>
<tr>
<td>0x1FFFF80</td>
<td>8 KB</td>
</tr>
<tr>
<td>0x1FFFF00</td>
<td>16 KB</td>
</tr>
<tr>
<td>0x1FFFFE0</td>
<td>32 KB</td>
</tr>
<tr>
<td>0x1FFFC00</td>
<td>64 KB</td>
</tr>
<tr>
<td>0x1FFFF00</td>
<td>128 KB</td>
</tr>
<tr>
<td>0x1FFF000</td>
<td>256 KB</td>
</tr>
<tr>
<td>0x1FFE000</td>
<td>512 KB</td>
</tr>
<tr>
<td>0x1FFC000</td>
<td>1 MB</td>
</tr>
<tr>
<td>0x1FF8000</td>
<td>2 MB</td>
</tr>
<tr>
<td>0x1FF0000</td>
<td>4 MB</td>
</tr>
<tr>
<td>0x1FE0000</td>
<td>8 MB</td>
</tr>
<tr>
<td>0x1FC0000</td>
<td>16 MB</td>
</tr>
<tr>
<td>0x1F80000</td>
<td>32 MB</td>
</tr>
<tr>
<td>0x1F00000</td>
<td>64 MB</td>
</tr>
</tbody>
</table>

Note to Table 3–27:

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

\[
\text{MASK} = 0x1FFFFFF << \log_2(\text{region\_size} >> 6)
\]

**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 noncacheable 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 3–53 for more information on cache bypass.
The PERM Field
The **PERM** field specifies the allowed access permissions. Table 3–29 lists possible values of the **PERM** field for instruction regions and Table 3–30 lists possible values of the **PERM** field for data regions.

Table 3–29. Instruction Region Permission Values

<table>
<thead>
<tr>
<th>Value</th>
<th>Supervisor Permissions</th>
<th>User Permissions</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>1</td>
<td>Execute</td>
<td>None</td>
</tr>
<tr>
<td>2</td>
<td>Execute</td>
<td>Execute</td>
</tr>
</tbody>
</table>

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 `wrc tl` instruction is issued to the `mpuacc` register. Refer to “MPU Region Read and Write Operations” on page 3–29 for more information. The **RD** flag always returns 0 when read by a `rdct l` instruction.

The WR Flag
Setting the **WR** flag signifies that an MPU region write operation should be performed when a `wrc tl` instruction is issued to the `mpuacc` register. Refer to “MPU Region Read and Write Operations” on page 3–29 for more information. The **WR** flag always returns 0 when read by a `rdct l` instruction.

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

Shadow Register Sets
The Nios II processor can optionally have one or more shadow register sets. A shadow register set is a complete alternate set of Nios II general-purpose registers, which can be used to maintain a separate runtime context for an interrupt service routine (ISR).
When shadow register sets are implemented, \texttt{status.CRS} indicates the register set currently in use. A Nios II core can have up to 63 shadow register sets. If \( n \) is the configured number of shadow register sets, the shadow register sets are numbered from 1 to \( n \). Register set 0 is the normal register set.

A shadow register set behaves precisely the same as the normal register set. The register set currently in use can only be determined by examining \texttt{status.CRS}.

When shadow register sets and the EIC interface are implemented on the Nios II core, you must ensure that your software is built with the Nios II EDS version 9.0 or later. Earlier versions have an implementation of the \texttt{eret} instruction that is incompatible with shadow register sets.

Shadow register sets are typically used in conjunction with the EIC interface. This combination can substantially reduce interrupt latency.

For details of EIC interface usage, refer to “Exception Processing” on page 3–30.

System software can read from and write to any shadow register set by setting \texttt{status.PRS} and using the \texttt{rdprs} and \texttt{wrprs} instructions.

For details of the \texttt{rdprs} and \texttt{wrprs} instructions, refer to the \textit{Instruction Set Reference} chapter of the \textit{Nios II Processor Reference Handbook}.

### The \texttt{sstatus} Register

The value in the \texttt{sstatus} register preserves the state of the Nios II processor during external interrupt handling. The value of \texttt{sstatus} is undefined at processor reset. Some bits are exclusively used by and available only to certain features of the processor. Table 3–31 shows the layout of the \texttt{sstatus} register.

The \texttt{sstatus} register is physically stored in general-purpose register \texttt{r30} in each shadow register set. The normal register set does not have an \texttt{sstatus} register, but each shadow register set has a separate \texttt{sstatus} register.

#### Table 3–31. \texttt{sstatus} Control Register Fields

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
<th>Access</th>
<th>Reset</th>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td>SRS (2)</td>
<td>SRS is the switched register set bit. The processor sets SRS to 1 when an external interrupt occurs, if the interrupt required the processor to switch to a different register set.</td>
<td>Read/Write</td>
<td>Undefined</td>
<td>EIC interface and shadow register sets only</td>
</tr>
<tr>
<td>RSIE</td>
<td>(1)</td>
<td>Read/Write</td>
<td>Undefined</td>
<td>(1)</td>
</tr>
<tr>
<td>NMI</td>
<td>(1)</td>
<td>Read/Write</td>
<td>Undefined</td>
<td>(1)</td>
</tr>
<tr>
<td>PRS</td>
<td>(1)</td>
<td>Read/Write</td>
<td>Undefined</td>
<td>(1)</td>
</tr>
</tbody>
</table>

Table 3–32 describes the fields defined in the \texttt{sstatus} register.

#### Table 3–32. \texttt{sstatus} Control Register Field Descriptions (Part 1 of 2)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
<th>Access</th>
<th>Reset</th>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td>SRS (2)</td>
<td>SRS is the switched register set bit. The processor sets SRS to 1 when an external interrupt occurs, if the interrupt required the processor to switch to a different register set.</td>
<td>Read/Write</td>
<td>Undefined</td>
<td>EIC interface and shadow register sets only</td>
</tr>
<tr>
<td>RSIE</td>
<td>(1)</td>
<td>Read/Write</td>
<td>Undefined</td>
<td>(1)</td>
</tr>
<tr>
<td>NMI</td>
<td>(1)</td>
<td>Read/Write</td>
<td>Undefined</td>
<td>(1)</td>
</tr>
<tr>
<td>PRS</td>
<td>(1)</td>
<td>Read/Write</td>
<td>Undefined</td>
<td>(1)</td>
</tr>
</tbody>
</table>
The \texttt{sstatus} register is present in the Nios II core if both the EIC interface and shadow register sets are implemented. There is one copy of \texttt{sstatus} for each shadow register set.

When the Nios II processor takes an interrupt, if a shadow register set is requested (RRS = 0) and the MMU is not in exception handler mode (\texttt{status.EH} = 0), the processor copies \texttt{status} to \texttt{sstatus}.

For details about RRS, refer to “Requested Register Set” on page 3–37. For details about \texttt{status.EH}, refer to Table 3–35 on page 3–46.

### Changing Register Sets

Modifying \texttt{status.CRS} immediately switches the Nios II processor to another register set. However, software cannot write to \texttt{status.CRS} directly. To modify \texttt{status.CRS}, insert the desired value into the saved copy of the \texttt{status} register, and then execute the \texttt{eret} instruction, as follows:

- If the processor is currently running in the normal register set, insert the new register set number in \texttt{estatus.CRS}, and execute \texttt{eret}.
- If the processor is currently running in a shadow register set, insert the new register set number in \texttt{sstatus.CRS}, and execute \texttt{eret}.

Before executing \texttt{eret} to change the register set, system software must set individual external interrupt masks correctly to ensure that registers in the shadow register set cannot be corrupted. If an interrupt is assigned to the register set, system software must ensure that one of the following conditions is true:

- The ISR is written to preserve register contents.
- The individual interrupt is disabled. The method for disabling an individual external interrupt is specific to the EIC implementation.

### Stacks and Shadow Register Sets

Depending on system requirements, the system software can create a dedicated stack for each register set, or share a stack among several register sets. If a stack is shared, the system software must copy the stack pointer each time the register set changes. Use the \texttt{rdprs} instruction to copy the stack register between the current register set and another register set.

---

**Table 3–32. \texttt{sstatus} Control Register Field Descriptions (Part 2 of 2)**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
<th>Access</th>
<th>Reset</th>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRS</td>
<td>(1)</td>
<td>Read/Write</td>
<td>Undefined</td>
<td>(1)</td>
</tr>
<tr>
<td>IL</td>
<td>(1)</td>
<td>Read/Write</td>
<td>Undefined</td>
<td>(1)</td>
</tr>
<tr>
<td>IH</td>
<td>(1)</td>
<td>Read/Write</td>
<td>Undefined</td>
<td>(1)</td>
</tr>
<tr>
<td>EH</td>
<td>(1)</td>
<td>Read/Write</td>
<td>Undefined</td>
<td>(1)</td>
</tr>
<tr>
<td>U</td>
<td>(1)</td>
<td>Read/Write</td>
<td>Undefined</td>
<td>(1)</td>
</tr>
<tr>
<td>PIE</td>
<td>(1)</td>
<td>Read/Write</td>
<td>Undefined</td>
<td>(1)</td>
</tr>
</tbody>
</table>

**Notes to Table 3–32:**

1. Refer to Table 3–8 on page 3–12.
2. If the EIC interface and shadow register sets are not present, \texttt{SRS} always reads as 0, and the processor behaves accordingly.

---

The \texttt{sstatus} register is present in the Nios II core if both the EIC interface and shadow register sets are implemented. There is one copy of \texttt{sstatus} for each shadow register set.

When the Nios II processor takes an interrupt, if a shadow register set is requested (RRS = 0) and the MMU is not in exception handler mode (\texttt{status.EH} = 0), the processor copies \texttt{status} to \texttt{sstatus}.

For details about RRS, refer to “Requested Register Set” on page 3–37. For details about \texttt{status.EH}, refer to Table 3–35 on page 3–46.
Initialization with Shadow Register Sets

At initialization, system software must carry out the following tasks to ensure correct software functioning with shadow register sets:

- After the gp register is initialized in the normal register set, copy it to all shadow register sets, to ensure that all code can correctly address the small data sections.
- Copy the zero register from the normal register set to all shadow register sets, using the wrprs instruction.

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 3–29. You must also disable unused regions. When using region size, clear `mpuacc.MASK` to zero. When using limit, set the `mpubase.BASE` to a nonzero 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 3–29. 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**

Exception processing is the act of responding to an exception, and then returning, if possible, to the pre-exception execution state.

All Nios II exceptions are precise. Precise exceptions enable the system software to re-execute the instruction, if desired, after handling the exception.

**Terminology**

Altera Nios II documentation uses the following terminology to discuss exception processing:
Exception—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.

Interrupt—an exception caused by an explicit request signal from an external device; also: hardware interrupt.

Interrupt controller—hardware that interfaces the processor to interrupt request signals from external devices.

Internal interrupt controller—the nonvectored interrupt controller that is integral to the Nios II processor. The internal interrupt controller is available in all revisions of the Nios II processor.

Vectored interrupt controller (VIC)—an Altera-provided external interrupt controller.

Exception (interrupt) latency—The time elapsed between the event that causes the exception (assertion of an interrupt request) and the execution of the first instruction at the handler address.

Exception (interrupt) response time—The time elapsed between the event that causes the exception (assertion of an interrupt request) and the execution of nonoverhead exception code, that is, specific to the exception type (device).

Global interrupts—All maskable exceptions on the Nios II processor, including internal interrupts and maskable external interrupts, but not including nonmaskable interrupts.

Worst-case latency—The value of the exception (interrupt) latency, assuming the maximum disabled time or maximum masked time, and assuming that the exception (interrupt) occurs at the beginning of the masked/disabled time.

Maximum disabled time—The maximum amount of continuous time that the system spends with maskable interrupts disabled.

Maximum masked time—The maximum amount of continuous time that the system spends with a single interrupt masked.

Shadow register set—a complete alternate set of Nios II general-purpose registers, which can be used to maintain a separate runtime context for an ISR.

**Exception Overview**

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

- Reset exception—Occurs when the Nios II processor is reset. Control is transferred to the reset address you specify in the Nios II processor IP core setup parameters.

- Break exception—Occurs when the JTAG debug module requests control. Control is transferred to the break address you specify in the Nios II processor IP core setup parameters.

- Interrupt exception—Occurs when a peripheral device signals a condition requiring service

- Instruction-related exception—Occurs when any of several internal conditions occurs, as detailed in Table 3–33 on page 3–32. Control is transferred to the exception address you specify in the Nios II processor IP core setup parameters.
Table 3–33 lists 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>
<thead>
<tr>
<th>Exception</th>
<th>Type</th>
<th>Available</th>
<th>Cause</th>
<th>Address</th>
<th>Vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reset</td>
<td>Reset</td>
<td>Always</td>
<td>0</td>
<td></td>
<td>Reset</td>
</tr>
<tr>
<td>Hardware break</td>
<td>Break</td>
<td>Always</td>
<td>—</td>
<td></td>
<td>Break</td>
</tr>
<tr>
<td>Processor-only reset request</td>
<td>Reset</td>
<td>Always</td>
<td>1</td>
<td></td>
<td>Reset</td>
</tr>
<tr>
<td>Internal interrupt</td>
<td>Interrupt</td>
<td>Internal interrupt controller</td>
<td>2</td>
<td>ea-4 (2)</td>
<td>General exception</td>
</tr>
<tr>
<td>External nonmaskable interrupt</td>
<td>Interrupt</td>
<td>External interrupt controller interface</td>
<td>—</td>
<td>ea-4 (2)</td>
<td>Requested handler address (3)</td>
</tr>
<tr>
<td>External maskable interrupt</td>
<td>Interrupt</td>
<td>External interrupt controller interface</td>
<td>2</td>
<td>ea-4 (2)</td>
<td>Requested handler address (3)</td>
</tr>
<tr>
<td>Supervisor-only instruction address (1)</td>
<td>Instruction-related</td>
<td>MMU</td>
<td>9</td>
<td>ea-4 (2)</td>
<td>General exception</td>
</tr>
<tr>
<td>Fast TLB miss (instruction) (1)</td>
<td>Instruction-related</td>
<td>MMU</td>
<td>12</td>
<td>pteaddr.VPN, ea-4 (2)</td>
<td>Fast TLB Miss exception</td>
</tr>
<tr>
<td>Double TLB miss (instruction) (1)</td>
<td>Instruction-related</td>
<td>MMU</td>
<td>12</td>
<td>pteaddr.VPN, ea-4 (2)</td>
<td>General exception</td>
</tr>
<tr>
<td>TLB permission violation (execute) (1)</td>
<td>Instruction-related</td>
<td>MMU</td>
<td>13</td>
<td>pteaddr.VPN, ea-4 (2)</td>
<td>General exception</td>
</tr>
<tr>
<td>MPU region violation (instruction) (1)</td>
<td>Instruction-related</td>
<td>MPU</td>
<td>16</td>
<td>ea-4 (2)</td>
<td>General exception</td>
</tr>
<tr>
<td>Supervisor-only instruction</td>
<td>Instruction-related</td>
<td>MMU or MPU</td>
<td>10</td>
<td>ea-4 (2)</td>
<td>General exception</td>
</tr>
<tr>
<td>Trap instruction</td>
<td>Instruction-related</td>
<td>Always</td>
<td>3</td>
<td>ea-4 (2)</td>
<td>General exception</td>
</tr>
<tr>
<td>Illegal instruction</td>
<td>Instruction-related</td>
<td>Illegal instruction detection on, MMU, or MPU</td>
<td>5</td>
<td>ea-4 (2)</td>
<td>General exception</td>
</tr>
<tr>
<td>Unimplemented instruction</td>
<td>Instruction-related</td>
<td>Always</td>
<td>4</td>
<td>ea-4 (2)</td>
<td>General exception</td>
</tr>
<tr>
<td>Break instruction</td>
<td>Instruction-related</td>
<td>Always</td>
<td>—</td>
<td>ba-4 (2)</td>
<td>Break</td>
</tr>
</tbody>
</table>
### Table 3–33. Nios II Exceptions (In Decreasing Priority Order) (Part 2 of 2)

<table>
<thead>
<tr>
<th>Exception</th>
<th>Type</th>
<th>Available</th>
<th>Cause</th>
<th>Address</th>
<th>Vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>Supervisor-only data address</td>
<td>Instruction-related</td>
<td>MMU</td>
<td>11 badaddr (data address)</td>
<td></td>
<td>General exception</td>
</tr>
<tr>
<td>Misaligned data address</td>
<td>Instruction-related</td>
<td>MMU</td>
<td>6 badaddr (data address)</td>
<td></td>
<td>General exception</td>
</tr>
<tr>
<td>Misaligned destination address</td>
<td>Instruction-related</td>
<td>MMU</td>
<td>7 badaddr (destination address)</td>
<td></td>
<td>General exception</td>
</tr>
<tr>
<td>division error</td>
<td>Instruction-related</td>
<td>Division error detection on</td>
<td>8 ea–4 (2)</td>
<td></td>
<td>General exception</td>
</tr>
<tr>
<td>Fast TLB miss (data)</td>
<td>Instruction-related</td>
<td>MMU</td>
<td>12 pteaddr.VPN, badaddr (data address)</td>
<td></td>
<td>Fast TLB Miss exception</td>
</tr>
<tr>
<td>Double TLB miss (data)</td>
<td>Instruction-related</td>
<td>MMU</td>
<td>12 pteaddr.VPN, badaddr (data address)</td>
<td></td>
<td>General exception</td>
</tr>
<tr>
<td>TLB permission violation (read)</td>
<td>Instruction-related</td>
<td>MMU</td>
<td>14 pteaddr.VPN, badaddr (data address)</td>
<td></td>
<td>General exception</td>
</tr>
<tr>
<td>TLB permission violation (write)</td>
<td>Instruction-related</td>
<td>MMU</td>
<td>15 pteaddr.VPN, badaddr (data address)</td>
<td></td>
<td>General exception</td>
</tr>
<tr>
<td>MPU region violation (data)</td>
<td>Instruction-related</td>
<td>MPU</td>
<td>17 badaddr (data address)</td>
<td></td>
<td>General exception</td>
</tr>
</tbody>
</table>

**Notes to Table 3–33:**
(1) It is possible for any instruction fetch to cause this exception.
(2) Refer to Table 3–5 on page 3–10 for descriptions of the ea and ba registers.
(3) For a description of the requested handler address, refer to “Requested Handler Address” on page 3–36.

---

**Exception Latency**

Exception latency specifies how quickly the system can respond to an exception. Exception latency depends on the type of exception, the software and hardware configuration, and the processor state.

**Interrupt Latency**

The interrupt controller can mask individual interrupts. Each interrupt can have a different maximum masked time. The worst-case interrupt latency for interrupt $i$ is determined by that interrupt’s maximum masked time, or by the maximum disabled time, whichever is greater.

**Reset Exceptions**

When a processor reset signal is asserted, the Nios II processor performs the following steps:
1. Sets `status.RSIE` to 1, and clears all other fields of the `status` register.
2. Invalidates the instruction cache line associated with the reset vector.
3. Begins executing the reset handler, located at the reset vector.

All noninterrupt exception handlers must run in the normal register set.

Clearing the `status.PIE` field disables maskable interrupts. If the MMU or MPU is present, clearing the `status.U` field forces the processor into supervisor mode.

Nonmaskable interrupts (NMIs) are not affected by `status.PIE`, and can be taken while processing a reset exception.

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) in the normal register set, which is permanently zero.
- Control registers, except for `status.status.RSIE` is reset to 1, and the remaining fields are reset to 0.
- 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 maskable interrupts.

   Nonmaskable interrupts (NMIs) are not affected by `status.PIE`, and can be taken while processing a break exception.

3. Writes the address of the instruction following the break to the `ba` register (`r30`) in the normal register set.
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. Copies `status.CRS` to `status.PRS` and then sets `status.CRS` to 0.
7. Transfers execution to the break handler, stored at the break vector specified in the Nios II Processor parameter editor.

All noninterrupt exception handlers, including the break handler, must run in the normal register set.

**Understanding Register Usage**

The `bstatus` control register and general-purpose registers `bt` (`r25`) and `ba` (`r30`) in the normal register set 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`) in the normal register set. Aside from `bt` and `ba`, all registers are guaranteed to be returned to their pre-break state after returning from the break handler.

**Interrupt Exceptions**

A peripheral device can request an interrupt by asserting an interrupt request (IRQ) signal. IRQs interface to the Nios II processor through an interrupt controller. You can configure the Nios II processor with either of the following interrupt controller options:
The external interrupt controller interface

The internal interrupt controller

**External Interrupt Controller Interface**

The Nios II EIC interface enables you to connect the Nios II processor to an external interrupt controller component. The EIC can monitor and prioritize IRQ signals, and determine which interrupt to present to the Nios II processor. An EIC can be software-configurable.

The Nios II processor does not depend on any particular implementation of an EIC. The degree of EIC configurability, and EIC configuration methods, are implementation-specific. This section discusses the EIC interface, and general features of EICs. For usage details, refer to the documentation for the specific EIC in your system.

For a typical EIC implementation, refer to the *Vectored Interrupt Controller* chapter in the *Embedded Peripherals IP User Guide*.

When an IRQ is asserted, the EIC presents the following information to the Nios II processor:

- The requested handler address (RHA)—Refer to “Requested Handler Address”
- The requested interrupt level (RIL)—Refer to “Requested Interrupt Level”
- The requested register set (RRS)—Refer to “Requested Register Set”
- Requested nonmaskable interrupt (RNMI) mode—Refer to “Requested NMI Mode”

The Nios II processor EIC interface connects to a single EIC, but an EIC can support a daisy-chained configuration. In a daisy-chained configuration, multiple EICs can monitor and prioritize interrupts. The EIC directly connected to the processor presents the processor with the highest-priority interrupt from all EICs in the daisy chain.

An EIC component can support an arbitrary level of daisy-chaining, potentially allowing the Nios II processor to handle an arbitrary number of prioritized interrupts.

**Requested Handler Address**

The RHA specifies the address of the handler associated with the interrupt. The availability of an RHA for each interrupt allows the Nios II processor to jump directly to the interrupt handler, reducing interrupt latency.

The RHA for each interrupt is typically software-configurable. The method for specifying the RHA is dependent on the specific EIC implementation.

If the Nios II processor is implemented with an MMU, the processor treats handler addresses as virtual addresses.

**Requested Interrupt Level**

The Nios II processor uses the RIL to decide when to take a maskable interrupt. The interrupt is taken only when the RIL is greater than \text{status.IL}.

The RIL is ignored for nonmaskable interrupts.
**Requested Register Set**

If shadow register sets are implemented on the Nios II core, the EIC specifies a register set when it asserts an interrupt request. When it takes the interrupt, the Nios II processor switches to the requested register set. When an interrupt has a dedicated register set, the interrupt handler avoids the overhead of saving registers.

The method of assigning register sets to interrupts depends on the specific EIC implementation. Register set assignments can be software-configurable.

Multiple interrupts can be configured to share a register set. In this case, the interrupt handlers must be written so as to avoid register corruption. For example, one of the following conditions must be true:

- The interrupts cannot pre-empt one another. For example, all interrupts are at the same level.
- Registers are saved in software. For example, each interrupt handler saves its own registers on entry, and restores them on exit.

Typically, the Nios II processor is configured so that when it takes an interrupt, other interrupts in the same register set are disabled. If interrupt preemption within a register set is desired, the interrupt handler can re-enable interrupts in its register set.

By default, the Nios II processor disables maskable interrupts when it takes an interrupt request. To enable nested interrupts, system software or the ISR itself must re-enable interrupts after the interrupt is taken.

Alternatively, to take full advantage of nested interrupts with shadow register sets, system software can set the \texttt{config.ANI} flag. When \texttt{config.ANI = 1}, the Nios II processor keeps maskable interrupts enabled after it takes an interrupt.

**Requested NMI Mode**

Any interrupt can be nonmaskable, depending on the configuration of the EIC. An NMI typically signals a critical system event requiring immediate handling, to ensure either system stability or real-time performance.

\texttt{status.IL} and \texttt{RIL} are ignored for nonmaskable interrupts.

**Shadow Register Sets**

Although shadow register sets can be implemented independently of the EIC interface, typically the two features are used together. Combining shadow register sets with an appropriate EIC, you can minimize or eliminate the context switch overhead for critical interrupts.

For the best interrupt performance, assign a dedicated register set to each of the most time-critical interrupts. Less-critical interrupts can share register sets, provided the ISRs are protected from register corruption as noted in “Requested Register Set”.

The method for mapping interrupts to register sets is specific to the particular EIC implementation.
Internal Interrupt Controller

When the internal interrupt controller is implemented, a peripheral device can request a hardware interrupt by asserting one of the Nios II 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, irqn, 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 “Exception Processing Flow” on page 3–43.

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.

Although shadow register sets can be implemented in any Nios II/f processor, the internal interrupt controller does not have features to take advantage of it as external interrupt controllers do.

![Figure 3–2. Relationship Between ienable, ipending, PIE and Hardware Interrupts](image-url)
Instruction-Related Exceptions

Instruction-related exceptions occur during execution of Nios II instructions. When they occur, the processor perform the steps outlined in “Exception Processing Flow” on page 3–43.

The Nios II processor generates the following instruction-related exceptions:

- 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

All noninterrupt exception handlers must run in the normal register set.

Trap Instruction

When a program issues the trap instruction, the processor 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. For more information, refer to “Break Exceptions” on page 3–34.

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. For more information, refer to “Potential Unimplemented Instructions” on page 3–60.
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.

For information about controlling this option, refer to the Instantiating the Nios II Processor 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 information about 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 3–1 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 3–4.

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.

For information about controlling this option, refer to the *Instantiating the Nios II Processor* 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.

For information about controlling this option, refer to the *Instantiating the Nios II Processor* 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.

For information about controlling this option, refer to the *Instantiating the Nios II Processor* 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 with the Nios II Processor parameter editor in Qsys and SOPC Builder, 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 preceding 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 categories listed in the preceding sections. 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.

**Exception Processing Flow**

Except for the break exception (refer to “Processing a Break” on page 3–35), this section describes how the processor responds to exceptions, including interrupts and instruction-related exceptions.

For details about writing programs to take advantage of exception and interrupt handling, refer to the Exception Handling chapter of the Nios II Software Developer’s Handbook.
Processing General Exceptions

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 with the Nios II Processor parameter editor in Qsys and SOPC Builder, is called the exception vector in the Nios II Processor parameter editor. 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.

If the EIC interface is present, the general exception handler processes only noninterrupt exceptions.

The fast TLB miss exception handler only handles the fast TLB miss exception. It is built for speed to process TLB misses quickly. The fast TLB miss exception handler address, specified with the Nios II Processor parameter editor in Qsys and SOPC Builder, is called the fast TLB miss exception vector in the Nios II Processor parameter editor.

Exception Flow with the EIC Interface

If the EIC interface is present, interrupt processing differs markedly from noninterrupt exception processing. The EIC interface provides the following information to the Nios II processor for each interrupt request:

- **RHA**—The requested handler address for the interrupt handler assigned to the requested interrupt.
- **RRS**—The requested register set to be used when the interrupt handler executes. If shadow register sets are not implemented, RRS must always be 0.
- **RIL**—The requested interrupt level specifies the priority of the interrupt.
- **RNMI**—The requested NMI flag specifies whether to treat the interrupt as nonmaskable.

For further information about the RHA, RRS, RIL and RNMI, refer to “The Nios II/f Core” in the *Nios II Core Implementation Details* chapter of the *Nios II Processor Reference Handbook*.

When the EIC interface presents an interrupt to the Nios II processor, the processor uses several criteria, as follows, to determine whether to take the interrupt:

- **Nonmaskable interrupts**—The processor takes any NMI as long as it is not processing a previous NMI.
- **Maskable interrupts**—The processor takes a maskable interrupt if maskable interrupts are enabled, and if the requested interrupt level is higher than that of the interrupt currently being processed (if any). However, if shadow register sets are implemented, the processor takes the interrupt only if the interrupt requests a register set different from the current register set, or if the register set interrupt enable flag (status.RSIE) is set.
Table 3–34 summarizes the conditions under which the Nios II processor takes an external interrupt.

Table 3–34. Conditions Required to Take External Interrupt

<table>
<thead>
<tr>
<th>RNMI == 1</th>
<th>RNMI == 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>status.NMI == 0</td>
<td>status.PIE == 1</td>
</tr>
<tr>
<td>status.NMI == 1</td>
<td>status.PIE == 1</td>
</tr>
<tr>
<td>status.IL &lt;= status.PIER</td>
<td>status.IL &gt; status.PIER</td>
</tr>
</tbody>
</table>

- Processor Has Shadow Register Sets
  - RRS == status.CRS
  - status.RSIE == 0
- No Shadow Register Sets
  - RRS != status.CRS
  - status.RSIE == 1

| (1) | Yes | No | No | No | Yes | Yes | Yes | Yes |

Note to Table 3–34:
(1) Nested interrupts using the same register set are allowed only if system software has explicitly permitted them by setting status.RSIE. This restriction ensures that such interrupts are taken only if the handler is coded to save the register context.

The Nios II processor supports fast nested interrupts with shadow register sets, as described in “Shadow Register Sets” on page 3–26. When shadow register sets are implemented, the config.ANI field is set to 0 at reset.

Software must set config.ANI to 1 to enable fast nested interrupts. If config.ANI is set to 1 when a maskable external interrupt occurs, status.PIE not cleared. Keeping status.PIE set allows higher level interrupts to be taken immediate, without requiring the interrupt handler to set status.PIE to 1.

System software can disable fast nested interrupts by setting config.ANI to 0. In this state, the processor disables maskable interrupts when taking an exception, just as it does without shadow register sets. An individual interrupt handler can re-enable interrupts by setting status.PIE to 1, if desired.

Exception Flow with the Internal Interrupt Controller

A general exception handler determines which of the pending interrupts has the highest priority, and then transfers control to the appropriate 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 savesestatus 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 “Handling Nested Exceptions” on page 3–48 for more information.
### Exceptions and Processor Status

Table 3–35 lists all changes to the Nios II processor state as a result of nonbreak exception processing actions performed by hardware. 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 3–35. Nios II Processor Status After Taking Exception

<table>
<thead>
<tr>
<th>Processor Status Register or Field</th>
<th>System Status Before Taking Exception</th>
<th>Internal Interrupt Asserted or Noninterrupt Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>External Interrupt Asserted (1)</td>
<td>status.EH=1 (2)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>status.EH=0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>status.EH=1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>status.EH=0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TLB Miss (4)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>No TLB Miss</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TLB Permission Violation (4)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>No TLB Permission Violation</td>
</tr>
<tr>
<td>pteaddr.VPN</td>
<td>No change</td>
<td>VPN (6)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>No change</td>
</tr>
<tr>
<td>status.PRS</td>
<td>No change</td>
<td>status.CRS (3)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(7) No change</td>
</tr>
<tr>
<td>pc</td>
<td>RHA</td>
<td>General exception vector (8)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Fast TLB exception vector (9)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>General exception vector (3)</td>
</tr>
<tr>
<td>sstatus</td>
<td>No change</td>
<td>status (7)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(12) No change</td>
</tr>
<tr>
<td>estatus</td>
<td>No change</td>
<td>status (7)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(11) No change</td>
</tr>
<tr>
<td>ea</td>
<td>No change</td>
<td>return address (13)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(13) No change</td>
</tr>
<tr>
<td>tlbmisc.D</td>
<td>No change</td>
<td>return address</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(14) No change</td>
</tr>
<tr>
<td>tlbmisc.DBL</td>
<td>No change</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>(15) No change</td>
</tr>
<tr>
<td>tlbmisc.PERM</td>
<td>No change</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>(16) No change</td>
</tr>
<tr>
<td>tlbmisc.BAD</td>
<td>No change</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>(17) No change</td>
</tr>
<tr>
<td>status.PIE</td>
<td>config.ANI (18)</td>
<td>0 (19)</td>
</tr>
<tr>
<td>status.EH</td>
<td>No change</td>
<td>1 (20)</td>
</tr>
<tr>
<td>status.IH</td>
<td>1</td>
<td>No change</td>
</tr>
<tr>
<td>status.NMI</td>
<td>RNMI</td>
<td>No change</td>
</tr>
<tr>
<td>status.IL</td>
<td>RIL</td>
<td>No change</td>
</tr>
<tr>
<td>status.RSIE</td>
<td>0</td>
<td>No change</td>
</tr>
<tr>
<td></td>
<td>(3) (21)</td>
<td></td>
</tr>
<tr>
<td>status.CRS</td>
<td>RRS</td>
<td>No change</td>
</tr>
<tr>
<td></td>
<td>(3) (21)</td>
<td></td>
</tr>
<tr>
<td>status.U</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(2)</td>
<td></td>
</tr>
</tbody>
</table>
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 3–16) 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 3–20) contains the byte instruction address or data address for certain exceptions. Refer to Table 3–33 on page 3–32 for more information.

External interrupts do not set exception.CAUSE.

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 sequence in Example 3–3 on page 3–48 to determine the cause of an exception.

Example 3–3. Determining Exception Cause Without Extra Exception Information

```c
/* With an internal interrupt controller, check for interrupt exceptions. With an external interrupt controller, ipending is always 0, and this check can be omitted. */
if (estatus.PIE == 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 (If internal interrupt controller is implemented, could be spurious interrupt)
}
```

Handling Nested Exceptions

The Nios II processor supports several types of nested exceptions, depending on which optional features are implemented. Nested exceptions can occur under the following circumstances:

- An exception handler enables maskable interrupts
- An EIC is present, and an NMI occurs
- An EIC is present, and the processor is configured to keep maskable interrupts enabled when taking an interrupt
- An exception handler triggers an instruction-related exception
For details about when the Nios II processor takes exceptions, refer to “Exception Processing Flow” on page 3–43. For details about unimplemented instructions, refer to the Processor Architecture chapter of the Nios II Processor Reference Handbook. For details about MMU and MPU exceptions, refer to “Instruction-Related Exceptions” on page 3–39.

A system can be designed to eliminate the possibility of nested exceptions. However, if nested exceptions are possible, the exception handlers must be carefully written to prevent each handler from corrupting the context in which a pre-empted handler runs.

If an exception handler issues a trap instruction, an optional instruction, or an instruction which could generate an MMU or MPU exception, it must save and restore the contents of the estatus and ea registers.

## Nested Exceptions with the Internal Interrupt Controller

You can enable nested exceptions in each exception handler on a case-by-case basis. If you want to allow a given exception handler to be pre-empted, set status.PIE to 1 near the beginning of the handler. Enabling maskable interrupts early in the handler minimizes the worst-case latency of any nested exceptions.

Ensure that all pre-empting handlers preserve the register contents.

## Nested Exceptions with an External Interrupt Controller

With an EIC, handling of nested interrupts is more sophisticated than with the internal interrupt controller. Handling of noninterrupt exceptions, however, is the same.

When individual external interrupts have dedicated shadow register sets, the Nios II processor supports fast interrupt handling with no overhead for saving register contents. To take full advantage of fast interrupt handling, system software must set up certain conditions. With the following conditions satisfied, ISRs need not save and restore register contents on entry and exit:

- Automatic nested interrupts are enabled (config.ANI is set to 1).
- Each interrupt is assigned to a dedicated shadow register set.
- All interrupts with the same RIL are assigned to dedicated shadow register sets.
Multiple interrupts with different RILs can be assigned to a single shadow register set. However, with multiple register sets, you must not allow the RILs assigned to one shadow register set to overlap the RILs assigned to another register set.

Table 3–36 and Table 3–37 demonstrate the validity of register set assignments when preemption within a register set is enabled.

### Table 3–36. Example of Illegal RIL Assignment

<table>
<thead>
<tr>
<th>RIL</th>
<th>Register Set 1</th>
<th>Register Set 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>IRQ0</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>IRQ1</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>IRQ1</td>
<td>IRQ2</td>
</tr>
<tr>
<td>4</td>
<td>IRQ3</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>IRQ4</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>IRQ5</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>IRQ6</td>
</tr>
</tbody>
</table>

### Table 3–37. Example of Legal RIL Assignment

<table>
<thead>
<tr>
<th>RIL</th>
<th>Register Set 1</th>
<th>Register Set 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>IRQ0</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>IRQ1</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>IRQ3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>IRQ2</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>IRQ4</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>IRQ5</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>IRQ6</td>
</tr>
</tbody>
</table>

Noninterrupt exception handlers must always save and restore the register contents, because they run in the normal register set.

Multiple interrupts can share a register set, with some loss of performance. There are two techniques for sharing register sets:

- Set `status.RSIE` to 0. When an ISR is running in a given register set, the processor does not take any maskable interrupt assigned to the same register set. Such interrupts must wait for the running ISR to complete, regardless of their interrupt level.

  This technique can result in a priority inversion.

- Ensure that each ISR saves and restores registers on entry and exit, and set `status.RSIE` to 1 after registers are saved. When an ISR is running in a given register set, the processor takes an interrupt in the same register set if it has a higher interrupt level.
System software can globally disable fast nested interrupts by setting `config.ANI` to 0. In this state, the Nios II processor disables interrupts when taking a maskable interrupt (nonmaskable interrupts always disable maskable interrupts). Individual ISRs can re-enable nested interrupts by setting `status.PIE` to 1, as described in “Nested Exceptions with the Internal Interrupt Controller” on page 3–49.

Handling Nonmaskable Interrupts

Writing an NMI handler involves the same basic techniques as writing any other interrupt handler. However, nonmaskable interrupts always preempt maskable interrupts, and cannot be preempted. This knowledge can simplify handler design in some ways, but it means that an NMI handler can have a significant impact on overall interrupt latency. For the best system performance, perform the absolute minimum of processing in your NMI handlers, and defer less-critical processing to maskable interrupt handlers or foreground software.

NMIs leave intact the processor state associated with maskable interrupts and other exceptions, as well as normal, nonexception processing, when each NMI is assigned to a dedicated shadow register set. Therefore, NMIs can be handled transparently.

1. If not assigned to a dedicated shadow register set, an NMI can overwrite the processor status associated with exception processing, making it impossible to return to the interrupted exception.

2. Do not set `status.PIE` in a nonmaskable ISR. If `status.PIE` is set, a maskable interrupt can pre-empt an NMI, and the processor exits NMI mode. It cannot be returned to NMI mode until the next nonmaskable interrupt.

Returning From Interrupt and Instruction-Related Exceptions

The `eret` instruction is used to resume execution at the pre-exception address.

You must ensure that when an exception handler modifies registers, they are restored when it returns. This can be taken care of in either of the following ways:

- In the case of ISRs, if the EIC interface and shadow register sets are implemented, and the ISR has a dedicated register set, no software action is required. The Nios II processor returns to the previous register set when it executes `eret`, which restores the register contents. For details, refer to “Nested Exceptions with an External Interrupt Controller”.

- In the case of noninterrupt exceptions, for ISRs in a system with the internal interrupt controller, and for ISRs without a dedicated shadow register set, the exception handler must save registers on entry and restore them on exit. Saving the register contents on the stack is a typical, re-entrant implementation.

1. It is not necessary to save and restore the exception temporary (`et` or `r24`) register.

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

1. Restores the previous contents of `status` as follows:
   - If `status.CRS` is 0, copies `estatus` to `status`
   - If `status.CRS` is nonzero, copies `sstatus` to `status`
2. Transfers program execution to the address in the ea register (r29) in the register set specified by the original value of status.CRS.

The eret instruction can cause the processor to exit NMI mode. However, it cannot make the processor enter NMI mode. In other words, if status.NMI is 0 and estatus.NMI (or sstatus.NMI) is 1, after an eret, status.NMI is still 0. This restriction prevents the processor from accidentally entering NMI mode.

When the EIC interface and shadow register sets are implemented on the Nios II core, you must ensure that your software, including ISRs, is built with the version of the GCC compiler included in Nios II EDS version 9.0 or later. Earlier versions have an implementation of the eret instruction that is incompatible with shadow register sets.

**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.

**Masking and Disabling Exceptions**

The Nios II processor provides several methods for temporarily turning off some or all exceptions from software. The available methods depend on the hardware configuration. This section discusses all potentially available methods.

**Disabling Maskable Interrupts**

Software can disable and enable maskable interrupts with the status.PIE bit. When PIE = 0, maskable interrupts are ignored. When PIE = 1, internal and maskable external interrupts can be taken, depending on the status of the interrupt controller.

**Masking Interrupts with an External Interrupt Controller**

Typical EIC implementations allow system software to mask individual interrupts. The method of masking individual interrupts is implementation-specific.

The status.IL field controls what level of external maskable interrupts can be serviced. The processor services a maskable interrupt only if its requested interrupt level is greater than status.IL.

An ISR can make run-time adjustments to interrupt nesting by manipulating status.IL. For example, if an ISR is running at level 5, to temporarily allow pre-emption by another level 5 interrupt, it can set status.IL to 4.

To enable all external interrupts, set status.IL to 0. To disable all external interrupts, set status.IL to 63.
Masking Interrupts with the Internal Interrupt Controller

The ienable register controls the handling of internal 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 3–30 for more information.

An ISR can adjust ienable so that IRQs of equal or lower priority are disabled. Refer to “Handling Nested Exceptions” on page 3–48 for more information.

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 uses little-endian byte ordering. 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 GB, 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 3–33 for more information.

For information about 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 KB), 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 KB 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 KB, 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-KB 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.

For information about the cache architecture and the memory hierarchy refer to the Processor Architecture chapter of the Nios II Processor Reference Handbook.
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 3–38 describes the wide (32-bit) load and store instructions.

Table 3–38. Wide Data Transfer Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ldw</td>
<td>The ldw and stw instructions load and store 32-bit data words from/to memory.</td>
</tr>
<tr>
<td>stw</td>
<td>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.</td>
</tr>
<tr>
<td>ldwio</td>
<td>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.</td>
</tr>
<tr>
<td>stwio</td>
<td></td>
</tr>
</tbody>
</table>

The data transfer instructions in Table 3–39 support byte and half-word transfers.

Table 3–39. Narrow Data Transfer Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ldb</td>
<td>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.</td>
</tr>
<tr>
<td>ldbu</td>
<td>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 in the following table cell.</td>
</tr>
<tr>
<td>ldh</td>
<td></td>
</tr>
<tr>
<td>ldhu</td>
<td></td>
</tr>
<tr>
<td>stb</td>
<td></td>
</tr>
<tr>
<td>sth</td>
<td></td>
</tr>
<tr>
<td>ldhio</td>
<td>These operations load/store byte and half-word data from/to peripherals without caching or buffering.</td>
</tr>
<tr>
<td>ldhuio</td>
<td></td>
</tr>
<tr>
<td>stbio</td>
<td></td>
</tr>
<tr>
<td>ldcio</td>
<td></td>
</tr>
<tr>
<td>ldcuio</td>
<td></td>
</tr>
</tbody>
</table>
Arithmetic and Logical Instructions

Logical instructions support \texttt{and}, \texttt{or}, \texttt{xor}, and \texttt{nor} operations. Arithmetic instructions support addition, subtraction, multiplication, and division operations. Refer to Table 3–40.

Table 3–40. Arithmetic and Logical Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{and}</td>
<td>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.</td>
</tr>
<tr>
<td>\texttt{or}</td>
<td></td>
</tr>
<tr>
<td>\texttt{xor}</td>
<td></td>
</tr>
<tr>
<td>\texttt{nor}</td>
<td></td>
</tr>
<tr>
<td>\texttt{andi}</td>
<td>These operations are immediate versions of the \texttt{and}, \texttt{or}, and \texttt{xor} instructions. The 16-bit immediate value is zero-extended to 32 bits, and then combined with a register value to form the result.</td>
</tr>
<tr>
<td>\texttt{ori}</td>
<td></td>
</tr>
<tr>
<td>\texttt{xori}</td>
<td></td>
</tr>
<tr>
<td>\texttt{andhi}</td>
<td>In these versions of \texttt{and}, \texttt{or}, and \texttt{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.</td>
</tr>
<tr>
<td>\texttt{orhi}</td>
<td></td>
</tr>
<tr>
<td>\texttt{xorhi}</td>
<td></td>
</tr>
<tr>
<td>\texttt{add}</td>
<td>These are the standard 32-bit arithmetic operations. These operations take two registers as input and store the result in a third register.</td>
</tr>
<tr>
<td>\texttt{sub}</td>
<td></td>
</tr>
<tr>
<td>\texttt{mul}</td>
<td></td>
</tr>
<tr>
<td>\texttt{div}</td>
<td></td>
</tr>
<tr>
<td>\texttt{divu}</td>
<td></td>
</tr>
<tr>
<td>\texttt{addi}</td>
<td>These instructions are immediate versions of the \texttt{add}, \texttt{sub}, and \texttt{mul} instructions. The instruction word includes a 16-bit signed value.</td>
</tr>
<tr>
<td>\texttt{subi}</td>
<td></td>
</tr>
<tr>
<td>\texttt{muli}</td>
<td></td>
</tr>
<tr>
<td>\texttt{mulxss}</td>
<td>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 \texttt{mul}.</td>
</tr>
<tr>
<td>\texttt{mulxuu}</td>
<td></td>
</tr>
<tr>
<td>\texttt{mulxsu}</td>
<td>This instruction is used in computing a 128-bit result of a 64x64 signed multiplication.</td>
</tr>
</tbody>
</table>

Move Instructions

These instructions provide move operations to copy the value of a register or an immediate value to another register. Refer to Table 3–41.

Table 3–41. Move Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{mov}</td>
<td>\texttt{mov} copies the value of one register to another register. \texttt{movi} moves a 16-bit signed immediate value to a register, and sign-extends the value to 32 bits. \texttt{movui} and \texttt{movhi} move a 16-bit immediate value into the lower or upper 16-bits of a register, inserting zeros in the remaining bit positions. Use \texttt{movia} to load a register with an address.</td>
</tr>
<tr>
<td>\texttt{movhi}</td>
<td></td>
</tr>
<tr>
<td>\texttt{movi}</td>
<td></td>
</tr>
<tr>
<td>\texttt{movui}</td>
<td></td>
</tr>
<tr>
<td>\texttt{movia}</td>
<td></td>
</tr>
</tbody>
</table>
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 3–42.

### Table 3–42. Comparison Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmpeq</td>
<td>$==$</td>
</tr>
<tr>
<td>cmpne</td>
<td>$!=$</td>
</tr>
<tr>
<td>cmpge</td>
<td>signed $\geq$</td>
</tr>
<tr>
<td>cmpgeu</td>
<td>unsigned $\geq$</td>
</tr>
<tr>
<td>cmpgt</td>
<td>signed $&gt;$</td>
</tr>
<tr>
<td>cmpgtu</td>
<td>unsigned $&gt;$</td>
</tr>
<tr>
<td>cmple</td>
<td>unsigned $\leq$</td>
</tr>
<tr>
<td>cmpleu</td>
<td>unsigned $\leq$</td>
</tr>
<tr>
<td>cmplt</td>
<td>signed $&lt;$</td>
</tr>
<tr>
<td>cmpltu</td>
<td>unsigned $&lt;$</td>
</tr>
<tr>
<td>cmpeqi</td>
<td></td>
</tr>
<tr>
<td>cmpnei</td>
<td></td>
</tr>
<tr>
<td>cmpgei</td>
<td></td>
</tr>
<tr>
<td>cmpgeui</td>
<td></td>
</tr>
<tr>
<td>cmpgti</td>
<td></td>
</tr>
<tr>
<td>cmpgtui</td>
<td></td>
</tr>
<tr>
<td>cmplei</td>
<td></td>
</tr>
<tr>
<td>cmpleui</td>
<td></td>
</tr>
<tr>
<td>cmplti</td>
<td></td>
</tr>
<tr>
<td>cmpltui</td>
<td></td>
</tr>
</tbody>
</table>

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 3–43.

### Table 3–43. Shift and Rotate Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rol</td>
<td>The rol and roli instructions provide left bit-rotation. rol 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.</td>
</tr>
<tr>
<td>ror</td>
<td></td>
</tr>
<tr>
<td>roli</td>
<td></td>
</tr>
<tr>
<td>sll</td>
<td>These shift instructions implement the $&lt;&lt;$ and $&gt;&gt;$ operators of the C programming language. The sll, slli, srl, srli, sra, and sraei 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, srl, srai and use an immediate value to specify the number of bits to shift.</td>
</tr>
</tbody>
</table>
Program Control Instructions

The Nios II architecture supports the unconditional jump, branch, and call instructions listed in Table 3–44. These instructions do not have delay slots.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>call</td>
<td>This instruction calls a subroutine using an immediate value as the subroutine’s absolute address, and stores the return address in register ra.</td>
</tr>
<tr>
<td>callr</td>
<td>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 role of dereferencing a C function pointer.</td>
</tr>
<tr>
<td>ret</td>
<td>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.</td>
</tr>
<tr>
<td>jmp</td>
<td>The jmp instruction jumps to an absolute address contained in a register. jmp is used to implement switch statements of the C programming language.</td>
</tr>
<tr>
<td>jmpi</td>
<td>The jmpi instruction jumps to an absolute address using an immediate value to determine the absolute address.</td>
</tr>
<tr>
<td>br</td>
<td>This instruction branches relative to the current instruction. A signed immediate value gives the offset of the next instruction to execute.</td>
</tr>
</tbody>
</table>

The conditional branch instructions compare register values directly, and branch if the expression is true. Refer to Table 3–45. The conditional branches support the following 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>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bge</td>
<td>These instructions provide relative branches that compare two register values and branch if the expression is true. Refer to “Comparison Instructions” on page 3–57 for a description of the relational operations implemented.</td>
</tr>
</tbody>
</table>
Other Control Instructions

Table 3–46 describes other control instructions.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>trap, eret</td>
<td>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.</td>
</tr>
<tr>
<td>break, bret</td>
<td>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.</td>
</tr>
<tr>
<td>rdctl, wrctl</td>
<td>These instructions read and write control registers, such as the status register. The value is read from or stored to a general-purpose register.</td>
</tr>
<tr>
<td>flushd, flushda, flushi, initd, initda, initi</td>
<td>These instructions are used to manage the data and instruction cache memories.</td>
</tr>
<tr>
<td>flushp</td>
<td>This instruction flushes all prefetched instructions from the pipeline. This is necessary before jumping to recently-modified instruction memory.</td>
</tr>
<tr>
<td>sync</td>
<td>This instruction ensures that all previously-issued operations have completed before allowing execution of subsequent load and store operations.</td>
</tr>
<tr>
<td>rdprs, wrprs</td>
<td>These instructions read and write a general-purpose registers between the current register set and another register set. wrprs can set r0 to 0 in a shadow register set. System software must use wrprs to initialize r0 to 0 in each shadow register set before using that register set.</td>
</tr>
</tbody>
</table>

Custom Instructions

The custom instruction provides low-level access to custom instruction logic. The inclusion of custom instructions is specified with the Nios II Processor parameter editor in Qsys and SOPC Builder, and the function implemented by custom instruction logic is design dependent.

For more information, refer to the “Custom Instructions” section of the Processor Architecture chapter of the Nios II Processor Reference Handbook and to the Nios II Custom Instruction User Guide.

Machine-generated C functions and assembly language macros provide access to custom instructions, and hide implementation details from the user. Therefore, most software developers never use the custom assembly language 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 3–39 for more information regarding unimplemented instruction processing.

### Document Revision History

Table 3–47 lists the revision history for this document.

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>May 2011</td>
<td>11.0.0</td>
<td>Added references to new Qsys system integration tool.</td>
</tr>
<tr>
<td>December 2010</td>
<td>10.1.0</td>
<td>Maintenance release.</td>
</tr>
<tr>
<td>July 2010</td>
<td>10.0.0</td>
<td>Maintenance release.</td>
</tr>
<tr>
<td>November 2009</td>
<td>9.1.0</td>
<td>Added external interrupt controller interface information.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Added shadow register set information.</td>
</tr>
<tr>
<td>March 2009</td>
<td>9.0.0</td>
<td>Maintenance release.</td>
</tr>
<tr>
<td>November 2008</td>
<td>8.1.0</td>
<td>Maintenance release.</td>
</tr>
<tr>
<td>May 2008</td>
<td>8.0.0</td>
<td>Added text to describe the MMU, MPU, and advanced exceptions.</td>
</tr>
<tr>
<td>October 2007</td>
<td>7.2.0</td>
<td>Reworked text to refer to break and reset as exceptions.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Grouped exceptions, break, reset, and interrupts all under Exception Processing.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Added table showing all Nios II exceptions (by priority).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Removed “ctl” references to control registers.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Added <code>jmpi</code> instruction to tables.</td>
</tr>
<tr>
<td>May 2007</td>
<td>7.1.0</td>
<td>Added table of contents to Introduction section.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Added Referenced Documents section.</td>
</tr>
</tbody>
</table>
### Table 3–47. Document Revision History (Part 2 of 2)

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>March 2007</td>
<td>7.0.0</td>
<td>Maintenance release.</td>
</tr>
<tr>
<td>November 2006</td>
<td>6.1.0</td>
<td>Maintenance release.</td>
</tr>
<tr>
<td>May 2006</td>
<td>6.0.0</td>
<td>Maintenance release.</td>
</tr>
<tr>
<td>October 2005</td>
<td>5.1.0</td>
<td>Maintenance release.</td>
</tr>
<tr>
<td>May 2005</td>
<td>5.0.0</td>
<td>Maintenance release.</td>
</tr>
<tr>
<td>September 2004</td>
<td>1.1</td>
<td>- Added details for new control register <code>ctl5</code>.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Updated details of debug and break processing to reflect new behavior of the <code>break</code> instruction.</td>
</tr>
<tr>
<td>May 2004</td>
<td>1.0</td>
<td>Initial release.</td>
</tr>
</tbody>
</table>
4. Instantiating the Nios II Processor

This chapter describes the Nios® II Processor parameter editor in Qsys and SOPC Builder. The Nios II Processor parameter editor allows you to specify the processor features for a particular Nios II hardware system. This chapter covers the features of the Nios II processor that you can configure with the Nios II Processor parameter editor; it is not a user guide for creating complete Nios II processor systems.

To get started designing custom Nios II systems, refer to the *Nios II Hardware Development Tutorial*. Development kits for Altera devices, available on the *All Development Kits* page of the Altera website, also provide ready-made hardware design examples that demonstrate different configurations of the Nios II processor.

The Nios II Processor parameter editor has several tabs. The following sections describe the parameters available on each tab.

- “Core Nios II Tab” on page 4–1
- “Caches and Memory Interfaces Tab” on page 4–6
- “Advanced Features Tab” on page 4–8
- “MMU and MPU Settings Tab” on page 4–12
- “JTAG Debug Module Tab” on page 4–13
- “Custom Instruction Tab” on page 4–16

**Core Nios II Tab**

The *Core Nios II* tab presents the main settings for configuring the Nios II processor. Table 4–1 lists the parameters and cross-references to their descriptions.

Table 4–1. Core Nios II Tab Parameters (Part 1 of 2)

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Select a Nios II Core</td>
<td></td>
</tr>
<tr>
<td>Nios II Core</td>
<td>Refer to “Core Selection” on page 4–2.</td>
</tr>
<tr>
<td>Hardware Arithmetic Operation</td>
<td></td>
</tr>
<tr>
<td>Hardware multiplication type</td>
<td>Refer to “Multiply and Divide Settings” on page 4–2.</td>
</tr>
<tr>
<td>Hardware divide</td>
<td></td>
</tr>
<tr>
<td>Reset Vector</td>
<td></td>
</tr>
<tr>
<td>Reset vector memory</td>
<td></td>
</tr>
<tr>
<td>Reset vector offset</td>
<td>Refer to “Reset Vector” on page 4–3.</td>
</tr>
<tr>
<td>Reset vector</td>
<td></td>
</tr>
</tbody>
</table>
The following sections describe the configuration settings available.

**Core Selection**

The main purpose of the Core Nios II tab is to select the processor core. The core you select on this tab affects other options available on this and other tabs.

Altera offers the following Nios II cores:

- **Nios II/f** — The Nios II/f fast core is designed for fast performance. As a result, this core presents the most configuration options allowing you to fine tune the processor for performance.

- **Nios II/s** — The Nios II/s standard core is designed for small size while maintaining performance.

- **Nios II/e** — The Nios II/e economy core is designed to achieve the smallest possible core size. As a result, this core has a limited feature set, and many settings are not available when the Nios II/e core is selected.

The Core Nios II tab displays a selector guide table that lists the basic properties of each core.

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

**Multiply and Divide Settings**

The Nios II/s and Nios II/f cores offer hardware multiply and divide options. You can choose the best option to balance embedded multiplier usage, logic element (LE) usage, and performance.

The Hardware multiplication type (Hardware Multiply in SOPC Builder) parameter for each core provides the following list:

---

Table 4–1. Core Nios II Tab Parameters (Part 2 of 2)

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Exception Vector</strong></td>
<td></td>
</tr>
<tr>
<td>Exception vector memory</td>
<td></td>
</tr>
<tr>
<td>Exception vector offset</td>
<td>Refer to “General Exception Vector” on page 4–4.</td>
</tr>
<tr>
<td>Exception vector</td>
<td></td>
</tr>
<tr>
<td><strong>MMU and MPU</strong></td>
<td></td>
</tr>
<tr>
<td>Include MMU</td>
<td></td>
</tr>
<tr>
<td>Fast TLB Miss Exception vector memory</td>
<td>Refer to “Memory Management Unit Settings” on page 4–4.</td>
</tr>
<tr>
<td>Fast TLB Miss Exception vector offset</td>
<td></td>
</tr>
<tr>
<td>Fast TLB Miss Exception vector</td>
<td></td>
</tr>
<tr>
<td>Include MPU</td>
<td>Refer to “Memory Protection Unit Settings” on page 4–5.</td>
</tr>
</tbody>
</table>
DSP Block—Include DSP block multipliers in the arithmetic logic unit (ALU). This option is only selectable when targeting devices that have DSP block multipliers.

Embedded Multipliers—Include embedded multipliers in the ALU. This option is only present when targeting FPGA devices that have embedded multipliers.

Logic Elements—Include LE-based multipliers in the ALU. This option achieves high multiply performance without consuming embedded multiplier resources, but with reduced fMAX.

None—This option conserves logic resources by eliminating multiply hardware. Multiply operations are implemented in software.

Shift operations use the multiplier. So, Hardware multiplication type affects shift instruction speed.

Turning on Hardware divide includes LE-based divide hardware in the ALU. The Hardware divide option achieves much greater performance than software emulation of divide operations.

For information about the performance effects of the hardware multiply and divide options, refer to the Nios II Core Implementation Details chapter of the Nios II Processor Reference Handbook.

Reset Vector

Parameters in this section select the memory module where the reset code (boot loader) resides, and the location of the reset vector (reset address). The reset vector cannot be configured until your system memory components are in place.

The Reset vector memory (Memory in SOPC Builder) list, which includes all memory modules mastered by the Nios II processor, selects the reset vector memory module. In a typical system, select a nonvolatile memory module for the reset code.

Qsys provides an Absolute option, which allows you to specify an absolute address in Reset vector offset. Use an absolute address when the memory storing the reset handler is located outside of the processor system and subsystems of the processor system.

Reset vector offset (Offset in SOPC Builder) specifies the location of the reset vector relative to the memory module’s base address. Qsys and SOPC Builder calculate the physical address of the reset vector when you modify the memory module, the offset, or the memory module’s base address. In Qsys, Reset vector displays the read-only, calculated address. In SOPC Builder, this address is displayed next to the Offset box. The address is always a physical address, even when an MMU is present.

For information about reset exceptions, refer to the Programming Model chapter of the Nios II Processor Reference Handbook.
General Exception Vector

Parameters in this section select the memory module where the general exception vector (exception address) resides, and the location of the general exception vector. The general exception vector cannot be configured until your system memory components are in place.

The Exception vector memory (Memory in SOPC Builder) list, which includes all memory modules mastered by the Nios II processor, selects the exception vector memory module. In a typical system, select a low-latency memory module for the exception code.

Qsys provides an Absolute option, which allows you to specify an absolute address in Exception vector offset. Use an absolute address when the memory storing the exception handler is located outside of the processor system and subsystems of the processor system.

Exception vector offset (Offset in SOPC Builder) specifies the location of the exception vector relative to the memory module’s base address. Qsys and SOPC Builder calculate the physical address of the exception vector when you modify the memory module, the offset, or the memory module’s base address. In Qsys, Exception vector displays the read-only, calculated address. In SOPC Builder, this address is displayed next to the Offset box. The address is always a physical address, even when an MMU is present.

For information about exceptions, refer to the Programming Model chapter of the Nios II Processor Reference Handbook.

Memory Management Unit Settings

The Nios II/f core offers a memory management unit (MMU) to support full-featured operating systems. Turning on Include MMU includes the Nios II MMU in your Nios II hardware system.

Do not include an MMU in your Nios II system unless your operating system requires it. The MMU is only useful with software that takes advantage of it. Many Nios II systems involve simpler system software, such as Altera® HAL or MicroC/OS-II. Such software is unlikely to function correctly with an MMU-based Nios II processor.

Fast TLB Miss Exception Vector

The fast TLB miss exception vector is a special exception vector used exclusively by the MMU to handle TLB miss exceptions. Parameters in this section select the memory module where the fast TLB miss exception vector (exception address) resides, and the location of the fast TLB miss exception vector. The fast TLB miss exception vector cannot be configured until your system memory components are in place.

The Fast TLB Miss Exception vector memory (Memory in SOPC Builder) list, which includes all memory modules mastered by the Nios II processor, selects the exception vector memory module. In a typical system, select a low-latency memory module for the exception code.
Qsys provides an **Absolute** option, which allows you to specify an absolute address in **Fast TLB Miss Exception vector offset**. Use an absolute address when the memory storing the exception handler is located outside of the processor system and subsystems of the processor system.

**Fast TLB Miss Exception vector offset** (Offset in SOPC Builder) specifies the location of the exception vector relative to the memory module’s base address. Qsys and SOPC Builder calculate the physical address of the exception vector when you modify the memory module, the offset, or the memory module’s base address. In Qsys, **Fast TLB Miss Exception vector** displays the read-only, calculated address. In SOPC Builder, this address is displayed next to the **Offset** box. The address is always a physical address, even when an MMU is present.

The Nios II MMU is optional and mutually exclusive from the Nios II MPU. Nios II systems can include either an MMU or MPU, but cannot include both an MMU and MPU in the same design.

**Memory Protection Unit Settings**

The Nios II/f core offers a memory protection unit (MPU) to support operating systems and runtime environments that desire memory protection without the overhead of virtual memory management. Turning on **Include MPU** includes the Nios II MPU in your Nios II hardware system.

The Nios II MPU is optional and mutually exclusive from the Nios II MMU. Nios II systems can include either an MPU or MMU, but cannot include both an MPU and MMU in the same design.

For information about the Nios II MPU, refer to the **Programming Model** chapter of the **Nios II Processor Reference Handbook**.
Caches and Memory Interfaces Tab

The Caches and Memory Interfaces tab allows you to configure the cache and tightly-coupled memory usage for the instruction and data master ports. Table 4–2 lists the parameters and cross-references to their descriptions.

Table 4–2. Caches and Memory Interfaces Tab Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction Master</td>
<td></td>
</tr>
<tr>
<td>Instruction cache</td>
<td>Refer to “Instruction Master Settings” on page 4–6.</td>
</tr>
<tr>
<td>Burst transfers (1)</td>
<td></td>
</tr>
<tr>
<td>Number of tightly coupled instruction master port(s) (1)</td>
<td></td>
</tr>
<tr>
<td>Data Master</td>
<td></td>
</tr>
<tr>
<td>Omit data master port</td>
<td>Refer to “Data Master Settings” on page 4–7.</td>
</tr>
<tr>
<td>Data cache</td>
<td></td>
</tr>
<tr>
<td>Data cache line size</td>
<td></td>
</tr>
<tr>
<td>Burst transfers (1)</td>
<td></td>
</tr>
<tr>
<td>Number of tightly coupled instruction master port(s) (2)</td>
<td></td>
</tr>
</tbody>
</table>

Notes to Table 4–2:

1. Enable Bursts in SOPC Builder.
2. In SOPC Builder, separate check box and list controls are used to include and specify the number of ports, respectively.

The following sections describe the configuration settings available.

Instruction Master Settings

The Instruction Master parameters provide the following options for the Nios II/f and Nios II/s cores:

- Instruction cache—Specifies the size of the instruction cache. Valid sizes are from 512 bytes to 64 KBytes, or None.
  
  Choosing None disables the instruction cache, which also removes the Avalon-MM instruction master port from the Nios II processor. In this case, you must include a tightly-coupled instruction memory.

- Burst transfers (Enable Bursts in SOPC Builder)—The Nios II processor can fill its instruction cache lines using burst transfers. Usually you enable bursts on the processor’s instruction master when instructions are stored in DRAM, and disable bursts when instructions are stored in SRAM.

  Bursting to DRAM typically improves memory bandwidth, but might consume additional FPGA resources. Be aware that when bursts are enabled, accesses to slaves might go through additional hardware (called burst adapters) which might decrease your fMAX.
When the Nios II processor transfers execution to the first word of a cache line, the processor fills the line by executing a sequence of word transfers that have ascending addresses, such as 0, 4, 8, 12, 16, 20, 24, 28.

However, when the Nios II processor transfers execution to an instruction that is not the first word of a cache line, the processor fetches the required (or “critical”) instruction first, and then fills the rest of the cache line. The addresses of a burst increase until the last word of the cache line is filled, and then continue with the first word of the cache line. For example, with a 32-byte cache line, transferring control to address 8 results in a burst with the following address sequence: 8, 12, 16, 20, 24, 28, 0, 4.

- **Number of tightly coupled instruction master port(s)** (Include tightly coupled instruction master port(s) and Number of ports in SOPC Builder)—Specifies one to four tightly-coupled instruction master ports for the Nios II processor. In Qsys, select the number from the Number of tightly coupled instruction master port(s) list. In SOPC Builder, turn on Include tightly coupled instruction master port(s) and select the number from the Number of ports list. Tightly-coupled memory ports appear on the connection panel of the Nios II processor on the Qsys and SOPC Builder System Contents tab. You must connect each port to exactly one memory component in the system.

### Data Master Settings

The **Data Master** parameters provide the following options for the Nios II/f core:

- **Omit data master**—Removes the Avalon-MM data master port from the Nios II processor. The port is only successfully removed when **Data cache** is set to **None** and Number of tightly coupled data master port(s) (Number of ports in SOPC Builder) is greater than zero.

Although the Nios II processor can operate entirely out of tightly-coupled memory without the need for Avalon-MM instruction or data masters, software debug is not possible when either the Avalon-MM instruction or data master is omitted.
Data cache—Specifies the size of the data cache. Valid sizes are from 512 bytes to 64 KBytes, or None. Depending on the value specified for Data cache, the following options are available:

- Data cache line size—Valid sizes are 4 bytes, 16 bytes, or 32 bytes.
- Burst transfers (Enable Bursts in SOPC Builder)—The Nios II processor can fill its data cache lines using burst transfers. Usually you enable bursts on the processor's data bus when processor data is stored in DRAM, and disable bursts when processor data is stored in SRAM.

  Bursting to DRAM typically improves memory bandwidth but might consume additional FPGA resources. Be aware that when bursts are enabled, accesses to slaves might go through additional hardware (called burst adapters) which might decrease your $f_{\text{MAX}}$.

  Bursting is only enabled for data cache line sizes greater than 4 bytes. The burst length is 4 for a 16 byte line size and 8 for a 32 byte line size. Data cache bursts are always aligned on the cache line boundary. For example, with a 32-byte Nios II data cache line, a cache miss to the address 8 results in a burst with the following address sequence: 0, 4, 8, 12, 16, 20, 24 and 28.

Number of tightly coupled data master port(s) (Include tightly coupled data master port(s) and Number of ports in SOPC Builder)—Specifies one to four tightly-coupled data master ports for the Nios II processor. In Qsys, select the number from the Number of tightly coupled data master port(s) list. In SOPC Builder, turn on Include tightly coupled data master port(s) and select the number from the Number of ports list. Tightly-coupled memory ports appear on the connection panel of the Nios II processor on the Qsys and SOPC Builder System Contents tab. You must connect each port to exactly one memory component in the system.

Advanced Features Tab

The Advanced Features tab allows you to enable specialized features of the Nios II processor. Table 4–2 lists the parameters and cross-references to their descriptions.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt controller</td>
<td>Refer to “Interrupt Controller Interfaces” on page 4–11.</td>
</tr>
<tr>
<td>Number of shadow register sets</td>
<td>Refer to “Shadow Register Sets” on page 4–11.</td>
</tr>
<tr>
<td>Include cpu_resetrequest and cpu_resettaken signals</td>
<td>Refer to “Reset Signals” on page 4–9.</td>
</tr>
<tr>
<td>Assign cpuid control register value manually</td>
<td>Refer to “Control Registers” on page 4–9.</td>
</tr>
<tr>
<td>cpuid control register value</td>
<td></td>
</tr>
</tbody>
</table>
Reset Signals

The Include cpu_resetrequest and cpu_resettaken signals reset signals setting provides the following functionality. When on, the Nios II processor includes processor-only reset request signals. These signals let another device individually reset the Nios II processor without resetting the entire system. The signals are exported to the top level of your system.

1 You must manually connect these signals to logic external to your Qsys or SOPC Builder system.

For more information on the reset signals, refer to the Processor Architecture chapter of the Nios II Processor Reference Handbook.

Control Registers

The Assign cpuid control register value manually control register setting allows you to assign the cpuid control register value yourself. In SOPC Builder, cpuid is normally automatically assigned because the assigned value is guaranteed to be unique. In Qsys, the automatically-assigned value is always 0x00000000, so Altera recommends always assigning the value manually.

To assign the value yourself, turn on Assign cpuid control register value manually and type a 32-bit value (in hexadecimal or decimal format) in the cpuid control register value box.

For information about upgrading IDs that were automatically assigned in SOPC Builder systems to manually-assigned values in Qsys, refer to the SOPC Builder to Qsys Migration Guidelines.

Exception Checking

The Exception Checking settings provide the following options:

- Illegal instruction—When Illegal instruction is on, the processor generates an illegal instruction exception when an instruction with an undefined opcode or opcode-extension field is executed.
When your system contains an MMU or MPU, the processor automatically generates illegal instruction exceptions. Therefore, the Illegal instruction setting is always disabled when the Core Nios II tab Include MMU or Include MPU are on.

**Division error**—Division error detection is only available for the Nios II/f core, and only then when Hardware divide on the Core Nios II tab is on. When divide instructions are not supported by hardware, the Division error setting is disabled. When Division error is on, the processor generates a division error exception when it detects divide instructions that produce a result that cannot be represented in the destination register. This only happens in the following two cases:

- Divide by zero
- Divide overflow—A signed division that divides the largest negative number -2,147,483,648 (0x80000000) by -1 (0xffffffff).

**Misaligned memory access**—Misaligned memory access detection is only available for the Nios II/f core. When Misaligned memory access is on, the processor checks for misaligned memory accesses.

When your system contains an MMU or MPU, the processor automatically generates misaligned memory access exceptions. Therefore, the Misaligned memory access check box is always disabled when Include MMU or Include MPU on the Core Nios II tab are on.

There are two misaligned memory address exceptions:

- Misaligned data address—Data addresses of load and store instructions are checked for misalignment. A data address is considered misaligned if the byte address is not a multiple of the data width of the load or store instruction (4 bytes for word, 2 bytes for half-word). Byte load and store instructions are always aligned so never generate a misaligned data address exception.
- Misaligned destination address—Destination instruction addresses of br, callr, jmp, ret, eret, and bret instructions are checked for misalignment. A destination instruction address is considered misaligned if the target byte address of the instruction is not a multiple of four.

**Extra exception information**—When Extra exception information is on, nonbreak exceptions store a code in the CAUSE field of the exception control register to indicate the cause of the exception.

When your system contains an MMU or MPU, the processor automatically generates extra exception information. Therefore, the Extra exception information setting is always disabled when the Core Nios II tab Include MMU or Include MPU are on.

Your exception handler can use this code to quickly determine the proper action to take, rather than have to determine the cause of an exception through instruction decoding. Additionally, some exceptions also store the instruction or data address associated with the exception in the badaddr register.
For further descriptions of exceptions, exception handling, and control registers, refer to the Programming Model chapter of the Nios II Processor Reference Handbook.

Interrupt Controller Interfaces

The Interrupt controller setting determines which of the following configurations is implemented:

- Internal interrupt controller
- External interrupt controller (EIC) interface

The EIC interface is available only on the Nios II/f core.

When the EIC interface and shadow register sets are implemented on the Nios II core, you must ensure that your software is built with the Nios II Embedded Design Suite (EDS) version 9.0 or higher. Earlier versions have an implementation of the eret instruction that is incompatible with shadow register sets.

For details about the EIC interface, refer to “Exception Processing” in the Programming Model chapter of the Nios II Processor Reference Handbook.

Shadow Register Sets

The Number of shadow register sets setting determines whether the Nios II core implements shadow register sets. The Nios II core can be configured with up to 63 shadow register sets.

Shadow register sets are available only on the Nios II/f core.

When the EIC interface and shadow register sets are implemented on the Nios II core, you must ensure that your software is built with the Nios II EDS version 9.0 or higher.

For details about shadow register sets, refer to “Registers” in the Programming Model chapter of the Nios II Processor Reference Handbook.

HardCopy Compatibility

The HardCopy Compatible parameter is only available in Qsys and determines whether the instantiated Nios II core is compatible with HardCopy® devices without recompilation. This feature allows you to migrate from an FPGA device to HardCopy device without any RTL changes to the Nios II core.

When HardCopy Compatible is on, any generated Nios II core and JTAG debug module RAM blocks are not pre-initialized.

When Device family on the Qsys Project Settings tab is a HardCopy device, HardCopy Compatible is automatically turned on and uneditable.
### MMU and MPU Settings Tab

The MMU and MPU Settings tab presents settings for configuring the MMU and MPU on the Nios II processor. You can select the features appropriate for your target application. Table 4–4 lists the parameters and cross-references to their descriptions.

**Table 4–4. MMU and MPU Settings Tab Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>MMU</strong></td>
<td></td>
</tr>
<tr>
<td>Process ID (PID) bits</td>
<td></td>
</tr>
<tr>
<td>Optimize number of TLB entries</td>
<td></td>
</tr>
<tr>
<td>based on device family</td>
<td>Refer to “MMU” on page 4–12.</td>
</tr>
<tr>
<td>TLB entries</td>
<td></td>
</tr>
<tr>
<td>TLB Set-Associativity</td>
<td></td>
</tr>
<tr>
<td>Micro DTLB entries</td>
<td></td>
</tr>
<tr>
<td>Micro ITLB entries</td>
<td></td>
</tr>
<tr>
<td><strong>MPU</strong></td>
<td></td>
</tr>
<tr>
<td>Use Limit for region range</td>
<td></td>
</tr>
<tr>
<td>Number of data regions</td>
<td>Refer to “MPU” on page 4–13.</td>
</tr>
<tr>
<td>Minimum data region size</td>
<td></td>
</tr>
<tr>
<td>Number of instruction regions</td>
<td></td>
</tr>
<tr>
<td>Minimum instruction region size</td>
<td></td>
</tr>
</tbody>
</table>

**MMU**

When Include MMU on the Core Nios II tab is on, the MMU settings on the MMU and MPU Settings tab provide the following options for the MMU in the Nios II/f core. Typically, you should not need to change any of these settings from their default values.

- **Process ID (PID) bits**—Specifies the number of bits to use to represent the process identifier.
- **Optimize number of TLB entries based on device family**—When on, specifies the optimal number of TLB entries to allocate based on the device family of the target hardware and disables TLB entries.
- **TLB entries**—Specifies the number of entries in the translation lookaside buffer (TLB).
- **TLB Set-Associativity**—Specifies the number of set-associativity ways in the TLB.
- **Micro DTLB entries**—Specifies the number of entries in the micro data TLB.
- **Micro ITLB entries**—Specifies the number of entries in the micro instruction TLB.

For information about the MMU, refer to the Programming Model chapter of the Nios II Processor Reference Handbook. For specifics on the Nios II/f core, refer to the Nios II Core Implementation Details chapter of the Nios II Processor Reference Handbook.
MPU

When Include MPU on the Core Nios II tab is on, the MPU settings on the MMU and MPU Settings tab provide the following options for the MPU in the Nios II/f core.

- **Use Limit for region range**—Controls whether the amount of memory in the region is defined by size or by upper address limit. When on, the amount of memory is based on the given upper address limit. When off, the amount of memory is based on the given size.

- **Number of data regions**—Specifies the number of data regions to allocate. Allowed values range from 2 to 32.

- **Minimum data region size**—Specifies the minimum data region size. Allowed values range from 64 bytes to 1 MB and must be a power of two.

- **Number of instruction regions**—Specifies the number of instruction regions to allocate. Allowed values range from 2 to 32.

- **Minimum instruction region size**—Specifies the minimum instruction region size. Allowed values range from 64 bytes to 1 MB and must be a power of two.

The maximum region size is the size of the Nios II instruction and data addresses automatically determined when the Nios II system is generated in Qsys and SOPC Builder. Maximum region size is based on the address range of slaves connected to the Nios II instruction and data masters.

For information about the MPU, refer to the Programming Model chapter of the Nios II Processor Reference Handbook. For specifics on the Nios II/f core, refer to the Nios II Core Implementation Details chapter of the Nios II Processor Reference Handbook.

JTAG Debug Module Tab

The JTAG Debug Module tab presents settings for configuring the JTAG debug module on the Nios II processor. You can select the debug features appropriate for your target application. Table 4–5 lists the parameters and cross-references to their descriptions.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Select a Debugging Level</td>
<td></td>
</tr>
<tr>
<td>Debug level</td>
<td>Refer to “Debug Level Settings” on page 4–14.</td>
</tr>
<tr>
<td>Include debugreq and debugack</td>
<td>Refer to “Debug Signals” on page 4–15.</td>
</tr>
<tr>
<td>Signals</td>
<td></td>
</tr>
<tr>
<td>Break vector memory</td>
<td></td>
</tr>
<tr>
<td>Break vector offset</td>
<td>Refer to “Break Vector” on page 4–15.</td>
</tr>
<tr>
<td>Break vector</td>
<td></td>
</tr>
</tbody>
</table>
Soft processor cores such as the Nios II processor offer unique debug capabilities beyond the features of traditional fixed processors. The soft nature of the Nios II processor allows you to debug a system in development using a full-featured debug core, and later remove the debug features to conserve logic resources. For the release version of a product, you might choose to reduce the JTAG debug module functionality, or remove it altogether.

Table 4–6 describes the debug features available to you for debugging your system.

### Debug Configuration Features

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>JTAG Target Connection</td>
<td>Connects to the processor through the standard JTAG pins on the Altera FPGA. This connection provides the basic capabilities to start and stop the processor, and examine/edit registers and memory.</td>
</tr>
<tr>
<td>Download Software</td>
<td>Downloads executable code to the processor’s memory via the JTAG connection.</td>
</tr>
<tr>
<td>Software Breakpoints</td>
<td>Sets a breakpoint on instructions residing in RAM.</td>
</tr>
<tr>
<td>Hardware Breakpoints</td>
<td>Sets a breakpoint on instructions residing in nonvolatile memory, such as flash memory.</td>
</tr>
<tr>
<td>Data Triggers</td>
<td>Triggers based on address value, data value, or read or write cycle. You can use a trigger to halt the processor on specific events or conditions, or to activate other events, such as starting execution trace, or sending a trigger signal to an external logic analyzer. Two data triggers can be combined to form a trigger that activates on a range of data or addresses.</td>
</tr>
<tr>
<td>Instruction Trace</td>
<td>Captures the sequence of instructions executing on the processor in real time.</td>
</tr>
<tr>
<td>Data Trace</td>
<td>Captures the addresses and data associated with read and write operations executed by the processor in real time.</td>
</tr>
<tr>
<td>On-Chip Trace</td>
<td>Stores trace data in on-chip memory.</td>
</tr>
<tr>
<td>Off-Chip Trace</td>
<td>Stores trace data in an external debug probe. Off-chip trace instantiates a PLL inside the Nios II core. Off-chip trace requires a debug probe from MIPS Technologies or Lauterbach GmbH.</td>
</tr>
</tbody>
</table>

The following sections describe the configuration settings available.

### Debug Level Settings

The following debug levels are available in the JTAG Debug Module tab:

- No Debugger
- Level 1
- Level 2
- Level 3
- Level 4
Table 4–7 is a detailed list of the characteristics of each debug level. Different levels consume different amounts of on-chip resources. Certain Nios II cores have restricted debug options, and certain options require debug tools provided by MIPS Technologies or Lauterbach GmbH.

For information about debug features available from these third parties, search for “Nios II” on the MIPS Technologies website (www.mips.com) and the Lauterbach GmbH website (www.lauterbach.com).

### Table 4–7. JTAG Debug Module Levels

<table>
<thead>
<tr>
<th>Debug Feature</th>
<th>No Debug</th>
<th>Level 1</th>
<th>Level 2</th>
<th>Level 3</th>
<th>Level 4 (1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logic Usage</td>
<td>0</td>
<td>300—400 LEs</td>
<td>800—900 LEs</td>
<td>2,400—2,700 LEs</td>
<td>3,100—3,700 LEs</td>
</tr>
<tr>
<td>On-Chip Memory Usage</td>
<td>0</td>
<td>Two M4Ks</td>
<td>Two M4Ks</td>
<td>Four M4Ks</td>
<td>Four M4Ks</td>
</tr>
<tr>
<td>External I/O Pins Required (2)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>20</td>
</tr>
<tr>
<td>JTAG Target Connection</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Download Software</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Software Breakpoints</td>
<td>None</td>
<td>Unlimited</td>
<td>Unlimited</td>
<td>Unlimited</td>
<td>Unlimited</td>
</tr>
<tr>
<td>Hardware Execution Breakpoints</td>
<td>0</td>
<td>None</td>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>Data Triggers</td>
<td>0</td>
<td>None</td>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>On-Chip Trace</td>
<td>0</td>
<td>None</td>
<td>None</td>
<td>Up to 64-KB frames (3)</td>
<td>Up to 64-KB frames</td>
</tr>
<tr>
<td>Off-Chip Trace (4)</td>
<td>0</td>
<td>None</td>
<td>None</td>
<td>None</td>
<td>128-KB frames</td>
</tr>
</tbody>
</table>

**Notes to Table 4–7:**

1. Level 4 requires the purchase of a software upgrade from MIPS Technologies or Lauterbach.
2. Not including the dedicated JTAG pins on the Altera FPGA.
3. An additional license from MIPS Technologies is required to use more than 16 frames.
4. Off-chip trace requires the purchase of additional hardware from MIPS Technologies or Lauterbach.

### Debug Signals

The Include debugreq and debugack signals debug signals setting provides the following functionality. When on, the Nios II processor includes debug request and acknowledge signals. These signals let another device temporarily suspend the Nios II processor for debug purposes. The signals are exported to the top level of your Qsys or SOPC Builder system.

For more information about the debug signals, refer to the Processor Architecture chapter of the Nios II Processor Reference Handbook.

### Break Vector

When the Nios II processor contains a JTAG debug module, Qsys and SOPC Builder determine a break vector (break address). Break vector memory (Memory in SOPC Builder) is always the processor core you are configuring. Break vector offset (Offset in SOPC Builder) is fixed at 0x20. Qsys and SOPC Builder calculate the physical address of the break vector from the memory module’s base address and the offset.
When the Nios II processor does not contain a JTAG debug module, you can edit the break vector parameters in the manner described in “General Exception Vector” on page 4–4.

**Advanced Debug Settings**

Debug levels 3 and 4 support trace data collection into an on-chip memory buffer. You can set the on-chip trace buffer size to sizes from 128 to 64K trace frames, using OCI Onchip Trace. Larger buffer sizes consume more on-chip M4K RAM blocks. Every M4K RAM block can store up to 128 trace frames.

The Nios II MMU does not support the JTAG debug module trace.

Debug level 4 also supports manual 2X clock signal specification. If you want to use a specific 2X clock signal in your FPGA design, turn off Automatically generate internal 2x clock signal and drive a 2X clock signal into your system manually.

For more information about trace frames, refer to the Processor Architecture chapter of the Nios II Processor Reference Handbook.

**Custom Instruction Tab**

In Qsys, custom instructions are components in your design that you manually connect to the processor in the Qsys System Contents tab. Existing custom instruction components are available on the Component Library tab under Custom Instruction Modules. Thus, the Custom Instruction tab in the Nios II Processor parameter editor is not used in Qsys.

In SOPC Builder, use the Custom Instructions tab to connect custom instruction logic to the Nios II arithmetic logic unit (ALU). To add a custom instruction to the Nios II processor, select the custom instruction from the list at the left side of the page, and click Add. The added instruction appears on the right side of the page.

For information about using SOPC Builder designs with custom instruction in Qsys, refer to the SOPC Builder to Qsys Migration Guidelines.

To create your own custom instruction using the component editor, click New Component on the File menu in Qsys, or Import on the Custom Instructions tab in SOPC Builder. After finishing in the component editor, the new instruction appears on the Component Library tab under Custom Instruction Modules in Qsys, or in the list at the left side of the Custom Instruction tab in SOPC Builder.

All signals in Nios II custom instructions must have the Custom Instruction Slave interface type. To guarantee the component editor automatically selects the Custom Instruction Slave interface type for your signals correctly during import, begin your signal names with the prefix ncs_. This prefix allows the component editor to determine the connection point type: a Nios II custom instruction slave. For example, if a custom instruction component has two data signals plus clock, reset, and result signals, an appropriate set of signal names is ncs_dataa, ncs_datab, ncs_clk, ncs_reset, and ncs_result.
A complete discussion of the hardware and software design process for custom instructions is beyond the scope of this chapter. For full details on the topic of custom instructions, including working example designs, refer to the Nios II Custom Instruction User Guide.

The default SOPC Builder System Contents tab connection panel display filter does not display custom instructions. To display custom instructions in the connection panel, click Filters at the bottom of the connection panel, and either select All Interfaces, or click New and create a custom filter that does not filter out custom instructions.

Altera-Provided Custom Instructions

The following sections describe the custom instructions Altera provides.

Floating-Point Hardware Custom Instruction

The Nios II processor offers a set of optional predefined custom instructions that implement floating-point arithmetic operations. You can include these custom instructions to support computation-intensive floating-point applications.

The basic set of floating-point custom instructions includes single precision (32-bit) floating-point addition, subtraction, and multiplication. Floating-point division is available as an extension to the basic instruction set. The best choice for your hardware design depends on a balance among floating-point usage, hardware resource usage, and performance.

If the target device includes on-chip multiplier blocks, the floating-point custom instructions incorporate them as needed. If there are no on-chip multiplier blocks, the floating-point custom instructions are entirely based on general-purpose logic elements.

The opcode extensions for the floating-point custom instructions are 252 through 255 (0xFC through 0xFF). These opcode extensions cannot be modified.

To add the floating-point custom instructions to the Nios II processor in Qsys, select Floating Point Hardware under Custom Instruction Modules on the Component Library tab, and click Add. In SOPC Builder, select Floating Point Hardware in the list on the Custom Instructions tab, and click Add. By default, Qsys and SOPC Builder include floating-point addition, subtraction, and multiplication, but omit the more resource intensive floating-point division. The Floating Point Hardware parameter editor appears, giving you the option to include the floating-point division hardware.

Table 4–8 lists the parameter, its possible values, and its description.

<table>
<thead>
<tr>
<th>Name</th>
<th>Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameters</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Use floating point division hardware</td>
<td>On/Off</td>
<td>Specifies inclusion of floating-point division hardware.</td>
</tr>
</tbody>
</table>
Turn on **Use floating point division hardware** to include floating-point division hardware. The floating-point division hardware requires more resources than the other instructions, so you might wish to omit it if your application does not make heavy use of floating-point division.

Click **Finish** to add the floating-point custom instructions to the Nios II processor.

For more information about the floating-point custom instructions, refer to the *Processor Architecture* chapter of the *Nios II Processor Reference Handbook*.

**Bitswap Custom Instruction**

The Nios II processor core offers a bitswap custom instruction to reduce the time spent performing bit reversal operations.

To add the bitswap custom instruction to the Nios II processor in Qsys, select **Bitswap** under **Custom Instruction Modules** on the **Component Library** tab, and click **Add**. In SOPC Builder, select **Bitswap** in the list on the **Custom Instructions** tab, and click **Add**.

The bitswap custom instruction reverses a 32-bit value in a single clock cycle. To perform the equivalent operation in software requires many mask and shift operations.

For details about integrating the bitswap custom instruction into your own algorithm, refer to the *Nios II Custom Instruction User Guide*.

**Endian Converter Custom Instruction**

An endian conversion custom instruction can reduce the time spent performing byte reversal operations. To use a custom instruction for this purpose, Altera recommends that you develop a custom instruction that meets your specific needs.

For an example that performs 16-bit, 32-bit, and 64-bit byte reversals, locate the Nios II endian converter on the Altera wiki website (www.alterawiki.com). For information about endianness in Avalon-MM interfaces, refer to the *Avalon-MM Byte Ordering Considerations* chapter of the *Embedded Design Handbook*.

In SOPC Builder, an endian converter custom instruction is available for the following specific use. The endian converter custom instruction takes a 32-bit value and reverses all four bytes in a single clock cycle. The Nios II core is a little-endian processor, so this custom instruction allows you to convert 32-bit data shared with a big-endian processor core.

To add the endian converter custom instruction to the Nios II processor in SOPC Builder, select **Endian Converter** from the list on the **Custom Instructions** tab, and click **Add**.

For information about converting the endian converter custom instruction for use in Qsys, refer to the *SOPC Builder to Qsys Migration Guidelines*.

Endian conversion custom instructions do not convert the Nios II processor core to a big-endian architecture; they only reverse the byte order of 32-bit data that you pass into the custom instruction.
Interrupt Vector Custom Instruction

In SOPC Builder, the Nios II processor core offers an interrupt vector custom instruction which reduces average- and worst-case interrupt latency. The interrupt vector custom instruction is less efficient than the EIC interface with the Altera vectored interrupt controller component, and thus is deprecated in Qsys. Altera recommends using the EIC interface.

The interrupt vector custom instruction is not compatible with the EIC interface. For the Nios II/f core, the EIC interface with the Altera vectored interrupt controller component provides superior performance.

To add the interrupt vector custom instruction to the Nios II processor in SOPC Builder, select Interrupt Vector from the list on the Custom Instructions tab, and click Add.

For information about converting the interrupt vector custom instruction for use in Qsys, refer to the SOPC Builder to Qsys Migration Guidelines.

There can only be one interrupt vector custom instruction component in a Nios II processor. If the interrupt vector custom instruction is present in the Nios II processor, the hardware abstraction layer (HAL) source detects it at compile time and generates code using the custom instruction.

The interrupt vector custom instruction improves both average- and worst-case interrupt latency by up to 20%. To achieve the lowest possible interrupt latency, consider using tightly-coupled memories so that interrupt handlers can run without cache misses.


The interrupt vector custom instruction is based on a priority encoder with one input for each interrupt connected to the Nios II processor. The cost of the interrupt vector custom instruction depends on the number of interrupts connected to the Nios II processor. The worst case is a system with 32 interrupts. In this case, the interrupt vector custom instruction consumes about 50 logic elements (LEs).

If you have a large number of interrupts connected, adding the interrupt vector custom instruction to your system might lower your fMAX.
Table 4–9 details the implementation of the interrupt vector custom instruction.

**Table 4–9. Interrupt Vector Custom Instruction**

<table>
<thead>
<tr>
<th>ALT_CI_EXCEPTION_VECTOR_N</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operation:</td>
</tr>
<tr>
<td>if (ipending == 0)</td>
</tr>
<tr>
<td>then rC ← negative value</td>
</tr>
<tr>
<td>else rC ← 8 × bit # of the least-significant 1 bit of the ipending register (ctl14)</td>
</tr>
</tbody>
</table>

| Assembler Syntax: |
| custom ALT_CI_EXCEPTION_VECTOR_N, rC, r0, r0 |

| Example: |
| custom ALT_CI_EXCEPTION_VECTOR_N, et, r0, r0 |
| blt et, r0, not_irq |

**Description:** The interrupt vector custom instruction accelerates interrupt vector dispatch. This custom instruction identifies the highest priority interrupt, generates the vector table offset, and stores this offset to rC. The instruction generates a negative offset if there is no hardware interrupt (that is, the exception is caused by a software condition, such as a trap).

**Usage:** The interrupt vector custom instruction is used exclusively by the exception handler.

**Exceptions:** None

**Instruction Type:** R

**Instruction Fields:**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>Register index of operand rC</td>
</tr>
<tr>
<td>N</td>
<td>Value of ALT_CI_EXCEPTION_VECTOR_N</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>0</td>
</tr>
<tr>
<td>30</td>
<td>0</td>
</tr>
<tr>
<td>29</td>
<td>0</td>
</tr>
<tr>
<td>28</td>
<td>C</td>
</tr>
<tr>
<td>27</td>
<td>0</td>
</tr>
<tr>
<td>26</td>
<td>0</td>
</tr>
<tr>
<td>25</td>
<td>1</td>
</tr>
<tr>
<td>24</td>
<td>N</td>
</tr>
<tr>
<td>23</td>
<td>0</td>
</tr>
<tr>
<td>22</td>
<td>0</td>
</tr>
<tr>
<td>21</td>
<td>0</td>
</tr>
<tr>
<td>20</td>
<td>0</td>
</tr>
<tr>
<td>19</td>
<td>0</td>
</tr>
<tr>
<td>18</td>
<td>0</td>
</tr>
<tr>
<td>17</td>
<td>0</td>
</tr>
<tr>
<td>16</td>
<td>0</td>
</tr>
<tr>
<td>15</td>
<td>0</td>
</tr>
<tr>
<td>14</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0x32</td>
</tr>
</tbody>
</table>

For an explanation of the instruction reference format, refer to the *Instruction Set Reference* chapter of the *Nios II Processor Reference Handbook*.

**The Quartus II IP File**

The Quartus® II IP file (.qip) is a file generated by the MegaWizard™ Plug-In Manager, Qsys, or SOPC Builder that contains information about a generated IP core. You are prompted to add this .qip file to the current project at the time of Quartus II file generation. In most cases, the .qip file contains all of the necessary assignments and information required to process the core or system in the Quartus II compiler. Generally, a single .qip file is generated for each MegaCore function and for each Qsys or SOPC Builder system. However, some complex components generate a separate .qip file, so the system .qip file references the component .qip file.
## Document Revision History

Table 4–10 lists the revision history for this document.

### Table 4–10. Document Revision History

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
</table>
| May 2011       | 11.0.0  | ■ Revised the entire chapter for the new Qsys system integration tool.  
                 ■ Replaced GUI screen shots with parameter tables.  
                 ■ Incorporated interrupt vector custom instruction information from the Processor Architecture chapter. |
| December 2010  | 10.1.0  | Maintenance release. |
| July 2010      | 10.0.0  | Maintenance release. |
| November 2009  | 9.1.0   | ■ Added external interrupt controller interface information.  
                 ■ Added shadow register set information. |
| March 2009     | 9.0.0   | Maintenance release. |
| November 2008  | 8.1.0   | ■ Added debugreq and debugack signal options to Advanced Features tab.  
                 ■ Added cpuid manual override options to Advanced Features tab. |
| May 2008       | 8.0.0   | ■ Added MMU options to Nios II Core and Advanced Features tabs.  
                 ■ Added exception handling options Advanced Features tab. |
| October 2007   | 7.2.0   | Changed title to match other Altera documentation. |
| May 2007       | 7.1.0   | ■ Revised to reflect new MegaWizard interface.  
                 ■ Added “Endian Converter Custom Instruction” on page 4–18 and “Bitswap Custom Instruction” on page 4–18.  
                 ■ Added table of contents to Introduction section.  
                 ■ Added Referenced Documents section. |
| March 2007     | 7.0.0   | Maintenance release. |
| November 2006  | 6.1.0   | ■ Add section on interrupt vector custom instruction.  
                 ■ Add section on system-dependent Nios II processor settings. |
| May 2006       | 6.0.0   | ■ Added details on floating-point custom instructions.  
                 ■ Added section on Advanced Features tab. |
| October 2005   | 5.1.0   | Maintenance release. |
| May 2005       | 5.0.0   | ■ Updates to reflect new GUI options in Nios II processor version 5.0.  
                 ■ New details in “Caches and Tightly-Coupled Memory” section. |
| September 2004 | 1.1     | ■ Updates to reflect new GUI options in Nios II processor version 1.1.  
                 ■ New details in section “Multiply and Divide Settings.” |
| May 2004       | 1.0     | Initial release. |