Nios II Processor Reference Handbook

101 Innovation Drive
San Jose, CA 95134
www.altera.com

NII5V1-7.1

Copyright © 2007 Altera Corporation. All rights reserved. Altera, The Programmable Solutions Company, the stylized Altera logo, specific device designations, and all other words and logos that are identified as trademarks and/or service marks are, unless noted otherwise, the trademarks and service marks of Altera Corporation in the U.S. and other countries. All other product or service names are the property of their respective holders. Altera products are protected under numerous U.S. and foreign patents and pending applications, maskwork rights, and copyrights. Altera warrants performance of its semiconductor products to current specifications in accordance with Altera's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Altera assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Altera Corporation. Altera customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.

Printed on recycled paper

I.S. EN ISO 9001

Contents

Chapter Revision Dates ix

About This Handbook xi

Introduction xi
Prerequisites 1–xi
How to Find Further Information xii
How to Contact Altera xii
Typographical Conventions xiii

Section I. Nios II Processor

Chapter 1. Introduction

Introduction 1–1
Nios II Processor System Basics 1–1
Getting Started with the Nios II Processor 1–2
Customizing Nios II Processor Designs 1–3
Configurable Soft-Core Processor Concepts 1–4
Configurable Soft-Core Processor 1–4
Flexible Peripheral Set and Address Map 1–5
Custom Instructions 1–5
Automated System Generation 1–6
Referenced Documents 1–6
Document Revision History 1–7

Chapter 2. Processor Architecture

Introduction 2–1
Processor Implementation 2–2
Register File 2–3
Arithmetic Logic Unit 2–4
Unimplemented Instructions 2–4
Custom Instructions 2–4
Floating Point Instructions 2–5
Reset Signals 2–6
Exception and Interrupt Controller 2–6
Exception Controller 2–6
Integral Interrupt Controller 2–6
Interrupt Vector Custom Instruction 2–7
Memory and I/O Organization 2–8

Instruction and Data Buses 2–9
Cache Memory 2–11
Tightly-Coupled Memory 2–13
Address Map 2–14
JTAG Debug Module 2–14
JTAG Target Connection 2–15
Download and Execute Software 2–15
Software Breakpoints 2–16
Hardware Breakpoints 2–16
Hardware Triggers 2–16
Trace Capture 2–17
Referenced Documents 2–19
Document Revision History 2–20

Chapter 3. Programming Model

Introduction 3–1
General-Purpose Registers 3–1
Control Registers 3–2
Operating Modes 3–4
Normal Mode 3–4
Debug Mode 3–5
Changing Modes 3–5
Exception Processing 3–5
Exception Types 3–6
Determining the Cause of Exceptions 3–8
Nested Exceptions 3–10
Returning from an Exception 3–10
Break Processing 3–11
Processing a Break 3–11
Returning from a Break 3–11
Register Usage 3–11
Memory and Peripheral Access 3–12
Addressing Modes 3–12
Cache Memory 3–12
Processor Reset State 3–13
Instruction Set Categories 3–14
Data Transfer Instructions 3–14
Arithmetic and Logical Instructions 3–16
Move Instructions 3–17
Comparison Instructions 3–17
Shift and Rotate Instructions 3–18
Program Control Instructions 3–19
Other Control Instructions 3–20
Custom Instructions 3–20
No-Operation Instruction 3–20
Potential Unimplemented Instructions 3–21
Referenced Documents 3–21

Document Revision History 3–22

Chapter 4. Implementing the Nios II Processor in SOPC Builder

Introduction 4–1
Core Nios II Page 4–2
Core Selection 4–3
Multiply and Divide Settings 4–3
Reset and Exception Vectors 4–4
Caches and Memory Interfaces Page 4–6
Instruction Master Settings 4–7
Data Master Settings 4–8
Advanced Features Page 4–9
JTAG Debug Module Page 4–11
Debug Level Settings 4–12
Break Vector 4–13
Advanced Debug Settings 4–14
Custom Instructions Page 4–14
Interrupt Vector Custom Instruction 4–16
Floating Point Hardware Custom Instruction 4–17
Endian Converter Custom Instruction 4–18
Bitswap Custom Instruction 4–19
Referenced Documents 4–19
Document Revision History 4–20

Section II. Appendices

Chapter 5. Nios II Core Implementation Details

Introduction 5–1
Device Family Support 5–3
Nios II/f Core 5–3
Overview 5–4
Register File 5–4
Arithmetic Logic Unit 5–4
Memory Access 5–6
Tightly-Coupled Memory 5–8
Execution Pipeline 5–9
Instruction Performance 5–10
Exception Handling 5–11
JTAG Debug Module 5–12
Unsupported Features 5–12
Nios II/s Core 5–12
Overview 5–12
Register File 5–13
Arithmetic Logic Unit 5–13
Memory Access 5–14

Tightly-Coupled Memory 5–15
Execution Pipeline 5–16
Instruction Performance 5–17
Exception Handling 5–18
JTAG Debug Module 5–18
Unsupported Features 5–18
Nios II/e Core 5–19
Overview 5–19
Register File 5–19
Arithmetic Logic Unit 5–19
Memory Access 5–20
Instruction Execution Stages 5–20
Instruction Performance 5–20
Exception Handling 5–21
JTAG Debug Module 5–21
Unsupported Features 5–21
Referenced Documents 5–22
Document Revision History 5–22

Chapter 6. Nios II Processor Revision History

Introduction 6–1
Nios II Versions 6–1
Architecture Revisions 6–2
Core Revisions 6–3
Nios II/f Core 6–3
Nios II/s Core 6–5
Nios II/e Core 6–6
JTAG Debug Module Revisions 6–6
Referenced Documents 6–7
Document Revision History 6–8

Chapter 7. Application Binary Interface

Data Types 7–1
Memory Alignment 7–2
Register Usage 7–2
Stacks 7–3
Frame Pointer Elimination 7–4
Call Saved Registers 7–4
Further Examples of Stacks 7–5
Function Prologs 7–6
Arguments and Return Values 7–8
Arguments 7–8
Return Values 7–8
Referenced Documents 7–9
Document Revision History 7–10

Chapter 8. Instruction Set Reference

Introduction 8–1
Word Formats 8–1
I-Type 8–1
R-Type 8–2
J-Type 8–3
Instruction Opcodes 8–4
Assembler Pseudo-instructions 8–6
Assembler Macros 8–7
Instruction Set Reference 8–8
Referenced Documents 8–103
Document Revision History 8–103

Chapter Revision Dates

The chapters in this book, *Nios II Processor Reference Handbook*, were revised on the following dates. Where chapters or groups of chapters are available separately, part numbers are listed.

Chapter 1. Introduction
Revised: *May 2007*
Part number: *NII51001-7.1.0*

Chapter 2. Processor Architecture
Revised: *May 2007*
Part number: *NII51002-7.1.0*

Chapter 3. Programming Model
Revised: *May 2007*
Part number: *NII51003-7.1.0*

Chapter 4. Implementing the Nios II Processor in SOPC Builder
Revised: *May 2007*
Part number: *NII51004-7.1.0*

Chapter 5. Nios II Core Implementation Details
Revised: *May 2007*
Part number: *NII51015-7.1.0*

Chapter 6. Nios II Processor Revision History
Revised: *May 2007*
Part number: *NII51018-7.1.0*

Chapter 7. Application Binary Interface
Revised: *May 2007*
Part number: *NII51016-7.1.0*

Chapter 8. Instruction Set Reference
Revised: *May 2007*
Part number: *NII51017-7.1.0*

About This Handbook

Introduction

This handbook is the primary reference for the Nios® II family of embedded processors. The handbook describes the Nios II processor from a high-level conceptual description to the low-level details of implementation. The chapters in this handbook define the Nios II processor architecture, the programming model, the instruction set, and more.

This handbook is part of a larger collection of documents covering the Nios II processor and its usage. See "How to Find Further Information" on page 1–xii.

Prerequisites

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 intentionally minimizes discussion of hardware implementation details of the processor system. That said, the Nios II processors are designed for Altera field programmable gate array (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.

How to Find Further Information

This handbook is one part of the complete Nios II processor documentation. The following references are also available.

- The *Nios II Software Developer's Handbook* describes the software development environment, and discusses application programming for the Nios II processor.
- The *Quartus II Handbook, Volume 5: Embedded Peripherals* discusses Altera-provided peripherals and Nios II drivers which are included with the Quartus® II software.
- The Nios II integrated development environment (IDE) provides tutorials and complete reference for using the features of the graphical user interface. The help system is available after launching the Nios II IDE.
- Altera's on-line solutions database is an internet resource that offers solutions to frequently asked questions via an easy-to-use search engine. Go to the support center on www.altera.com and click on the Find Answers link.
- Altera application notes and tutorials offer step-by-step instructions on using the Nios II processor for a specific application or purpose. These documents are often installed with Altera development kits, or can be obtained online from www.altera.com.

How to Contact Altera

For the most up-to-date information about Altera® products, refer to the following table.

Information Type	Contact *(1)*
Technical support	www.altera.com/mysupport/
Technical training	www.altera.com/training/ custrain@altera.com
Product literature	www.altera.com/literature/
Altera literature services	literature@altera.com
FTP site	ftp.altera.com

Note to table:

(1) You can also contact your local Altera sales office or sales representative.

Typographical Conventions

This document uses the typographic conventions shown below.

Visual Cue	Meaning
Bold Type with Initial Capital Letters	Command names, dialog box titles, checkbox options, and dialog box options are shown in bold, initial capital letters. Example: **Save As** dialog box.
Bold type	External timing parameters, directory names, project names, disk drive names, filenames, filename extensions, and software utility names are shown in bold type. Examples: $\mathbf{f_{MAX}}$, **\qdesigns** directory, **d:** drive, **chiptrip.gdf** file.
Italic Type with Initial Capital Letters	Document titles are shown in italic type with initial capital letters. Example: *AN 75: High-Speed Board Design.*
Italic type	Internal timing parameters and variables are shown in italic type. Examples: t_{PIA}, $n + 1$. Variable names are enclosed in angle brackets (< >) and shown in italic type. Example: *<file name>*, *<project name>***.pof** file.
Initial Capital Letters	Keyboard keys and menu names are shown with initial capital letters. Examples: Delete key, the Options menu.
"Subheading Title"	References to sections within a document and titles of on-line help topics are shown in quotation marks. Example: "Typographic Conventions."
`Courier type`	Signal and port names are shown in lowercase Courier type. Examples: `data1`, `tdi`, `input`. Active-low signals are denoted by suffix `n`, e.g., `resetn`. Anything that must be typed exactly as it appears is shown in Courier type. For example: `c:\qdesigns\tutorial\chiptrip.gdf`. Also, sections of an actual file, such as a Report File, references to parts of files (e.g., the AHDL keyword `SUBDESIGN`), as well as logic function names (e.g., `TRI`) are shown in Courier.
1., 2., 3., and a., b., c., etc.	Numbered steps are used in a list of items when the sequence of the items is important, such as the steps listed in a procedure.
■ ● •	Bullets are used in a list of items when the sequence of the items is not important.
✓	The checkmark indicates a procedure that consists of one step only.
☞	The hand points to information that requires special attention.
CAUTION	A caution calls attention to a condition or possible situation that can damage or destroy the product or the user's work.
WARNING	A warning calls attention to a condition or possible situation that can cause injury to the user.
↵	The angled arrow indicates you should press the Enter key.
	The feet direct you to more information on a particular topic.

Section I. Nios II Processor

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, Implementing the Nios II Processor in SOPC Builder

1. Introduction

NII51001-7.1.0

Introduction

This chapter is an introduction to the Nios® II embedded processor family. This chapter helps hardware and software engineers understand the similarities and differences between the Nios II processor and traditional embedded processors.

This chapter contains the following sections:

■ "Configurable Soft-Core Processor Concepts" on page 1–4

Nios II Processor System Basics

The Nios II processor is a general-purpose RISC processor core, providing:

- Full 32-bit instruction set, data path, and address space
- 32 general-purpose registers
- 32 external interrupt sources
- Single-instruction 32 × 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 integrated development environment (IDE) control
- Software development environment based on the GNU C/C++ tool chain and Eclipse IDE
- Integration with Altera®'s SignalTap® II 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. The term "Nios II processor

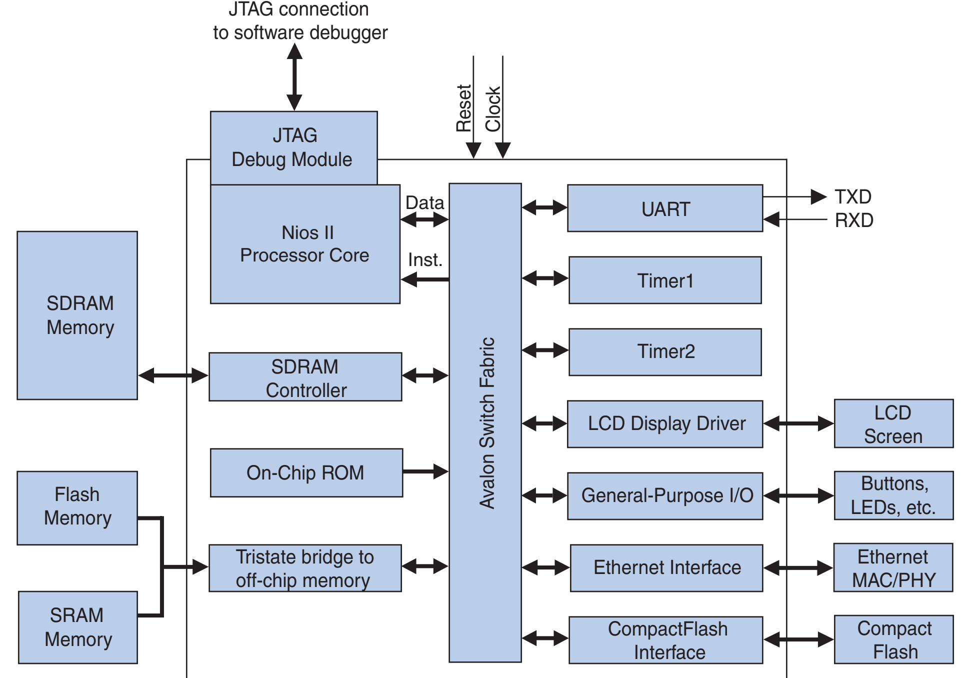
system" refers to a Nios II processor core, a set of on-chip peripherals, on-chip memory, and interfaces to off-chip memory, all implemented on a single Altera device. Like a microcontroller family, all Nios II processor systems use a consistent instruction set and programming model.

Getting Started with the Nios II Processor

Getting started with the Nios II processor is similar to any other microcontroller family. The easiest way to start designing effectively is to purchase a development kit from Altera that includes a ready-made evaluation board and all the software development tools necessary to write Nios II software.

The Nios II software development environment is called The Nios II integrated development environment (IDE). The Nios II IDE is based on the GNU C/C++ compiler and the Eclipse IDE, and provides a familiar and established environment for software development. Using the Nios II IDE, you can immediately begin developing and simulating Nios II software applications. 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 Nios II 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 as-is 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. Conversely, you can 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 siimplify 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-Core Processor 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 document.

For the most part, these concepts relate to the flexibility available to hardware designers to fine-tune system implementation. Software programmers generally are not affected by the hardware implementation details, and can write programs without awareness of the configurable nature of the Nios II processor core.

Configurable Soft-Core Processor

The Nios II processor is a configurable soft-core processor, as opposed to a fixed, off-the-shelf microcontroller. In this context, "configurable" means that you can add or remove features on a system-by-system basis to meet performance or price goals. "Soft-core" means the processor core is offered in "soft" design form (i.e., not fixed in silicon), and can be targeted to any Altera FPGA family.

Configurability does not require you 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 the system requirements, there is no need to configure the design further. In addition, software designers 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 of the soft-core nature of the Nios II processor, 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 grow as Altera and third-party vendors release new soft peripheral cores.

Custom Peripherals

You can also create custom peripherals and integrate them into 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.

Custom Instructions

Like custom peripherals, custom instructions allow you to increase system performance by augmenting the processor with custom hardware. The soft-core nature of the Nios II processor enables you to integrate custom logic into the 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 know assembly in order to use custom instructions.

Automated System Generation

Altera's SOPC Builder design tool fully automates the process of configuring processor features and generating a hardware design that you program into an FPGA. The SOPC Builder graphical user interface (GUI) enables 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 hardware description-language (HDL) design entry. SOPC Builder can also import HDL design files, providing an easy mechanism to integrate custom logic into 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, non-configurable processors.

Referenced Documents

This chapter references no other documents.

Document Revision History

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

Table 1–1. Document Revision History

Date & Document Version	Changes Made	Summary of Changes
May 2007 v7.1.0	• Added table of contents to Introduction section. • Added Referenced Documents section.	
March 2007 v7.0.0	No change from previous release.	
November 2006 v6.1.0	No change from previous release.	
May 2006 v6.0.0	• Added single precision floating point and integration with SignalTap®II logic analyzer to features list. • Updated performance to 250 DMIPS.	
October 2005 v5.1.0	No change from previous release.	
May 2005 v5.0.0	No change from previous release.	
September 2004 v1.1	Updates for Nios II 1.01 release.	
May 2004 v1.0	Initial release.	

2. Processor Architecture

NII51002-7.1.0

Introduction

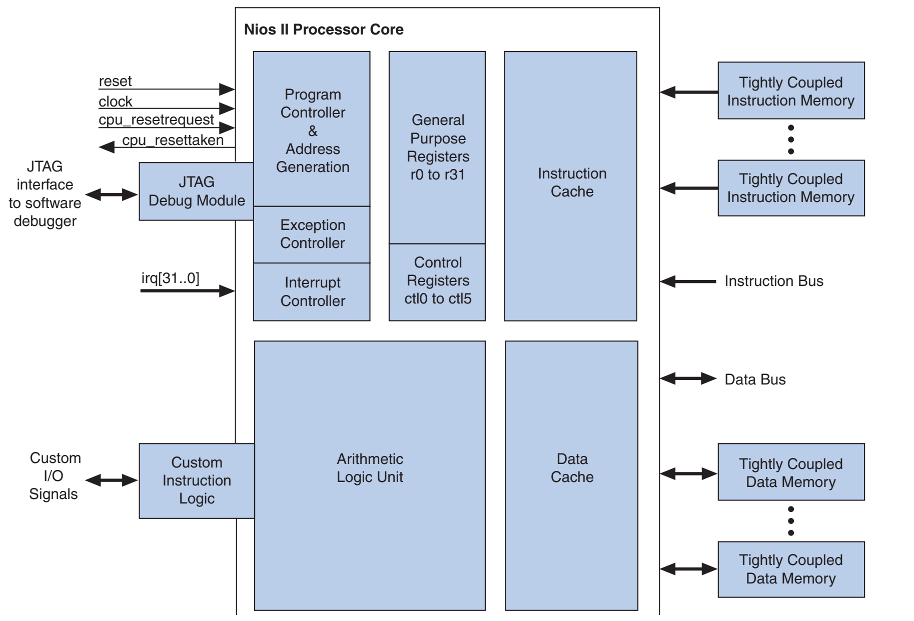
This chapter describes the hardware structure of the Nios® II processor, including a discussion of all the functional units of the Nios II architecture and the fundamentals of the Nios II processor hardware implementation. This chapter contains the following sections:

- "Processor Implementation" on page 2–2
- "Register File" on page 2–3
- "Arithmetic Logic Unit" on page 2–4
- "Reset Signals" on page 2–6
- "Exception and Interrupt Controller" on page 2–6
- "Memory and I/O Organization" on page 2–8
- "JTAG Debug Module" on page 2–14

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.

Figure 2–1 shows a block diagram of the Nios II processor core.

Figure 2–1. Nios II Processor Core Block Diagram



The Nios II architecture defines the following user-visible functional units:

- Register file
- Arithmetic logic unit
- Interface to custom instruction logic
- Exception controller
- Interrupt controller
- Instruction bus
- Data bus
- 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 *Nios II Processor Reference Handbook*. Each implementation achieves specific objectives, such as smaller core size or higher performance. This allows the Nios II architecture to adapt to the needs of 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 details of 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 of user-selectable parameters for the Nios II processor, refer to the *Implementing the Nios II Processor in SOPC Builder* 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 six 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 architecture allows for the future addition of floating point registers.

Arithmetic Logic Unit

The Nios II arithmetic logic unit (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 shown in Table 2–1:

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

Category	Details
Arithmetic	The ALU supports addition, subtraction, multiplication, and division on signed and unsigned operands.
Relational	The ALU supports the equal, not-equal, greater-than-or-equal, and less-than relational operations (==, != >=, <) on signed and unsigned operands.
Logical	The ALU supports AND, OR, NOR, and XOR logical operations.
Shift and Rotate	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.

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 perform multiplication or division operations. The following instructions are not present in all Nios II core implementations: `mul`, `muli`, `mulxss`, `mulxsu`, `mulxuu`, `div`, `divu`. In such a core, these are known as unimplemented instructions. All other instructions are implemented in hardware.

The processor generates an exception whenever it issues an unimplemented instruction, and the exception handler calls a routine that emulates the operation in software. Therefore, unimplemented instructions do not affect the programmer's view of the processor.

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 in hardware operations that are accessed and used exactly like native instructions.

For further information see the *Nios II Custom Instruction User Guide*.

Floating Point Instructions

The Nios II architecture supports single precision floating point instructions as specified by the IEEE Std 754-1985. These floating point instructions are implemented as custom instructions. Table 2–2 provides a detailed description of the conformance to IEEE 754-1985.

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

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

Notes to: Table 2–2

(1) The Nios II IDE generates a software implementation 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.

The floating point custom instructions can be added to any Nios II processor core. The Nios II software development tools recognize C code that can take advantage of the floating point instructions when they are present in the processor core.

Reset Signals

The Nios II processor core supports two reset signals.

- `reset` - This a global hardware reset signal that forces the processor core to reset immediately.
- `cpu_resetrequest` - This is an 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. The processor core asserts the `cpu_resettaken` signal for 1 cycle when the reset is complete and then periodically if cpu_resetrequest remains asserted. The processor remains in reset for as long as `cpu_resetrequest` is asserted.

  While the processor is in reset, it periodically reads from the reset address. It discards the result of the read, and remains in reset.

  The processor does not respond to `cpu_resetrequest` when the processor is under the control of the JTAG debug module, that is, when the proceesor 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.

Exception and Interrupt Controller

Exception Controller

The Nios II architecture provides a simple, non-vectored exception controller to handle all exception types. All exceptions, including hardware interrupts, cause the processor to transfer execution to a single *exception address*. The exception handler at this address determines the cause of the exception and dispatches an appropriate exception routine.

The exception address is specified at system generation time.

Integral Interrupt Controller

The Nios II architecture supports 32 external 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.

The 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 three of these 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

Interrupt Vector Custom Instruction

The Nios II processor core offers an interrupt vector custom instruction which accelerates interrupt vector dispatch. Include this custom instruction to reduce your program's interrupt latency.

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 f_{MAX}.

For guidance in adding the interrupt vector custom instruction to the Nios II processor, refer to the *Implementing the Nios II Processor in SOPC Builder* chapter of the *Nios II Software Developer's Handbook*.

Table 2–3 details the implementation of the interrupt vector custom instruction.

Table 2–3. Interrupt Vector Custom Instruction

ALT_CI_EXCEPTION_VECTOR_N

Operation:	if (`ipending` == 0) | (`estatus.PIE` == 0) then rC ← negative value else rC ← 8 × bit # of the least-significant 1 bit of the `ipending` register (`ctl4`)
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.

Instruction Type: R

Instruction Fields: C = Register index of operand rC
N = Value of ALT_CI_EXCEPTION_VECTOR_N

31–27	26–22	21–17	16	15	14	13–6	5–0
0	0	C	0	0	1	N	0x32

For an explanation of the instruction reference format, see the *Instruction Set Reference* 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-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 memory outside the Nios II core

The Nios II architecture hides the hardware details from the programmer, so programmers can develop Nios II applications without awareness of the hardware implementation.

For details that affect programming issues, see 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.

Figure 2–2. Nios II Memory and I/O Organization

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 Memory Mapped Interface Specification* 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 is little endian. 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 f_{MAX} 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. See “Cache Memory” on page 2–11 for details. The Nios II architecture supports tightly-coupled memory, which provides guaranteed low-latency access to on-chip memory. See “Tightly-Coupled Memory” on page 2–13 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 wth 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. See "Cache Memory" 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 2–13 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 Kbytes, but the size of the instruction cache is 1 Kbyte, 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 2–13 for details.

Cache Bypass Method

The Nios II architecture provides load and store I/O instructions such as `ldio` and `stio` that bypass the data cache and force an Avalon-MM data transfer to a specified address. Additional cache bypass methods might be provided, depending on the processor core implementation.

Some Nios II processor cores support a mechanism called *bit-31 cache bypass* to bypass the cache depending on the value of the most-significant bit of the address.

Refer to the *Implementing the Nios II Core Implementation Details* chapter of the *Nios II Processor Reference Handbook* for details.

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 memories 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 usually 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 partition exception handler code into a tightly-coupled memory to minimize interrupt latency. Similarly, compute-intensive digital signal processing (DSP) applications can partition 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.

Address Map

The address map for memories and peripherals in a Nios II processor system is design dependent. You specify the address map at system generation time.

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.

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:

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

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 non-maskable 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 control to a routine located at the break address. The *break address* is specified at system generation time.

Soft-core processors such as the Nios II processor offer unique debug capabilities beyond the features of traditional, fixed processors. The soft-core 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 IDE, which manages the connection to the target processor and controls the debug process.

JTAG Target Connection

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

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 provide the ability 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 provide the ability 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 2–4 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.

When a trigger condition occurs during processor execution, the JTAG debug module triggers an action, such as halting execution, or starting trace capture. Table 2–5 lists the trigger actions supported by the Nios II JTAG debug module.

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.

Table 2–4. Trigger Conditions

Condition	Bus (1)	Description
Specific address	D, I	Trigger when the bus accesses a specific address.
Specific data value	D	Trigger when a specific data value appears on the bus.
Read cycle	D	Trigger on a read bus cycle.
Write cycle	D	Trigger on a write bus cycle.
Armed	D, I	Trigger only after an armed trigger event. See "Armed Triggers" on page 2–16.
Range	D	Trigger on a range of address values, data values, or both. See "Triggering on Ranges of Values" on page 2–17.

Notes:
(1) "I" indicates instruction bus, "D" indicates data bus.

Table 2–5. Trigger Actions

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

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

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 First Silicon Solutions (FS2) or Lauterbach.

For details, see **www.fs2.com** and **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 (i.e., 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.

Referenced Documents

This chapter references the following documents:

- *Nios II Core Implementation Details* chapter of the *Nios II Processor Reference Handbook*
- *Implementing the Nios II Processor in SOPC Builder* chapter of the *Nios II Processor Reference Handbook*
- *Nios II Custom Instruction User Guide*
- *Instruction Set Reference* chapter of the *Nios II Processor Reference Handbook*
- *Programming Model* chapter of the *Nios II Processor Reference Handbook*
- *Avalon Memory Mapped Interface Specification*

Document Revision History

Table 2–6 shows the revision history for this document.

Table 2–6. Document Revision History

Date & Document Version	Changes Made	Summary of Changes
May 2007 v7.1.0	• Added table of contents to Introduction section. • Added Referenced Documents section.	
March 2007 v7.0.0	No change from previous release.	
November 2006 v6.1.0	Describe interrupt vector custom instruction.	Interrupt vector custom instruction added.
May 2006 v6.0.0	• Added description of optional `cpu_resetrequest` and `cpu_resettaken`. • Added section on single precision floating point.	
October 2005 v5.1.0	No change from previous release.	
May 2005 v5.0.0	Added tightly-coupled memory.	
December 2004 v1.2	Added new control register `ctl5`.	
September 2004 v1.1	Updates for Nios II 1.01 release.	
May 2004 v1.0	Initial release.	

3. Programming Model

NII51003-7.1.0

Introduction

This chapter describes the Nios® II programming model, covering processor features at the assembly language level. The programmer's view of the following features are discussed in detail:

- General-purpose registers, page 3–1
- Control registers, page 3–2
- Hardware-assisted debug processing, page 3–11
- Exception processing, page 3–5
- Hardware interrupts, page 3–6
- Unimplemented instructions, page 3–8
- Memory and peripheral organization, page 3–12
- Cache memory, page 3–12
- Processor reset state, page 3–13
- Instruction set categories, page 3–14
- Custom instructions, page 3–20

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

General-Purpose Registers

The Nios II architecture provides thirty-two 32-bit general-purpose registers, `r0` through `r31`. See Table 3–1 on page 2. Some registers have names recognized by the assembler. The `zero` register (`r0`) always returns the value 0, and writing to `zero` has no effect. The `ra` register (`r31`) holds the return address used by procedure calls and is implicitly accessed by `call` and `ret` instructions. C and C++ compilers use a common procedure-call convention, assigning specific meaning to registers `r1` through `r23` and `r26` through `r28`.

Table 3–1. The Nios II General Purpose Registers

Register	Name	Function	Register	Name	Function
`r0`	`zero`	0x00000000	`r16`		
`r1`	`at`	Assembler Temporary	`r17`		
`r2`		Return Value	`r18`		
`r3`		Return Value	`r19`		
`r4`		Register Arguments	`r20`		
`r5`		Register Arguments	`r21`		
`r6`		Register Arguments	`r22`		
`r7`		Register Arguments	`r23`		
`r8`		Caller-Saved Register	`r24`	`et`	Exception Temporary
`r9`		Caller-Saved Register	`r25`	`bt`	Breakpoint Temporary *(1)*
`r10`		Caller-Saved Register	`r26`	`gp`	Global Pointer
`r11`		Caller-Saved Register	`r27`	`sp`	Stack Pointer
`r12`		Caller-Saved Register	`r28`	`fp`	Frame Pointer
`r13`		Caller-Saved Register	`r29`	`ea`	Exception Return Address
`r14`		Caller-Saved Register	`r30`	`ba`	Breakpoint Return Address *(1)*
`r15`		Caller-Saved Register	`r31`	`ra`	Return Address

Notes to Table 3–1:
(1) This register is used exclusively by the JTAG debug module.

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

Control Registers

There are six 32-bit control registers, `ctl0` through `ctl5`. All control registers have names recognized by the assembler.

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.

Details of the control registers are shown in Table 3–2. For details on the relationship between the control registers and exception processing, see Figure 3–1 on page 3–7.

Table 3–2. Control Register and Bits

Register	Name	31...1	0
ctl0	status	Reserved	PIE
ctl1	estatus	Reserved	EPIE
ctl2	bstatus	Reserved	BPIE
ctl3	ienable	Interrupt-enable bits	
ctl4	ipending	Pending-interrupt bits	
ctl5	cpuid	Unique processor identifier	

status (ctl0)

The value in the `status` register controls the state of the Nios II processor. All status bits are cleared after processor reset. See "Processor Reset State" on page 3–13. One bit is defined: PIE, as shown in Table 3–3.

Table 3–3. Status Register Bits

Bit	Description
PIE bit	PIE is the processor interrupt-enable bit. When PIE is 0, external interrupts are ignored. When PIE is 1, external interrupts can be taken, depending on the value of the `ienable` register.

estatus (ctl1)

The `estatus` register holds a saved copy of the `status` register during exception processing. One bit is defined: EPIE. This is the saved values of PIE, as defined in Table 3–3.

The exception handler can examine estatus to determine the pre-exception status of the processor. When returning from an interrupt, the eret instruction causes the processor to copy estatus back to status, restoring the pre-exception value of status. See "Exception Processing" on page 3–5 for more information.

bstatus (ctl2)

The `bstatus` register holds a saved copy of the `status` register during debug break processing. One bit is defined: BPIE. This is the saved value of PIE, as defined in Table 3–3 on page 3–3.

When a break occurs, the value of the `status` register is copied into `bstatus`. Using `bstatus`, the `status` register can be restored to the value it had prior to the break. See "Debug Mode" on page 3–5 for more information.

ienable (ctl3)

The `ienable` register controls the handling of external hardware interrupts. Each bit of the `ienable` register corresponds to one of the interrupt inputs, `irq0` through `irq31`. A bit value of 1 means that the corresponding interrupt is enabled; a bit value of 0 means that the corresponding interrupt is disabled. See "Exception Processing" on page 3–5 for more information.

ipending (ctl4)

The value of the `ipending` register indicates which interrupts are pending. A value of 1 in bit *n* means that the corresponding `irq`*n* input is asserted, and that the corresponding interrupt is enabled in the `ienable` register. The effect of writing a value to the `ipending` register is undefined.

cpuid (ctl5)

The `cpuid` register holds a static value that uniquely identifies the processor in a multi-processor system. The `cpuid` value is determined at system generation time. Writing to the `cpuid` register has no effect. See "Exception Processing" on page 3–5 for more information.

Operating Modes

The Nios II processor has two operating modes:

- Normal mode
- Debug mode

The following sections define the modes and the transitions between modes.

Normal Mode

In general, system and application code execute in normal mode. The processor is in normal mode immediately after processor reset.

General-purpose registers `bt` (`r25`) and `ba` (`r30`) are not available in normal mode. Programs are not prevented from storing values in these registers, but if they do, the debug mode could overwrite the values. The `bstatus` register (`ctl2`) is also unavailable in normal mode.

Debug Mode

Software debugging tools use debug mode to implement features such as breakpoints and watch-points. System code and application code never execute in debug mode. The processor enters debug mode only after the `break` instruction or after the JTAG debug module forces a break via hardware.

In debug mode all processor functions are available and unrestricted to the software debugging tool. See "Break Processing" on page 3–11 for further information.

Changing Modes

The processor starts in normal mode after reset. It enters debug mode only as directed by software debugging tools. System code and application code have no control over when the processor enters debug mode. The processor always returns to its prior state when exiting from debug mode. See "Break Processing" on page 3–11 for further information.

Exception Processing

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

An exception causes the processor to take the following steps:

1. Copies the contents of the `status` register (`ctl0`) to `estatus` (`ctl1`) saving the processor's pre-exception status

2. Clears the PIE bit of the `status` register, disabling external processor interrupts

3. Writes the address of the instruction after the exception to the `ea` register (`r29`)

4. Transfers execution to the address of the *exception handler* that determines the cause of the interrupt

The address of the exception handler is specified at system generation time. At run-time this address is fixed, and software cannot modify it. Programmers do not directly access the exception handler address, and can write programs without awareness of the address.

The exception handler is a routine that determines the cause of each exception, and then dispatches an appropriate *exception routine* to respond to the interrupt.

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

Exception Types

Nios II exceptions fall into the following categories:

- Hardware interrupt
- Software trap
- Unimplemented instruction
- Other

The following sections describe each exception type in detail.

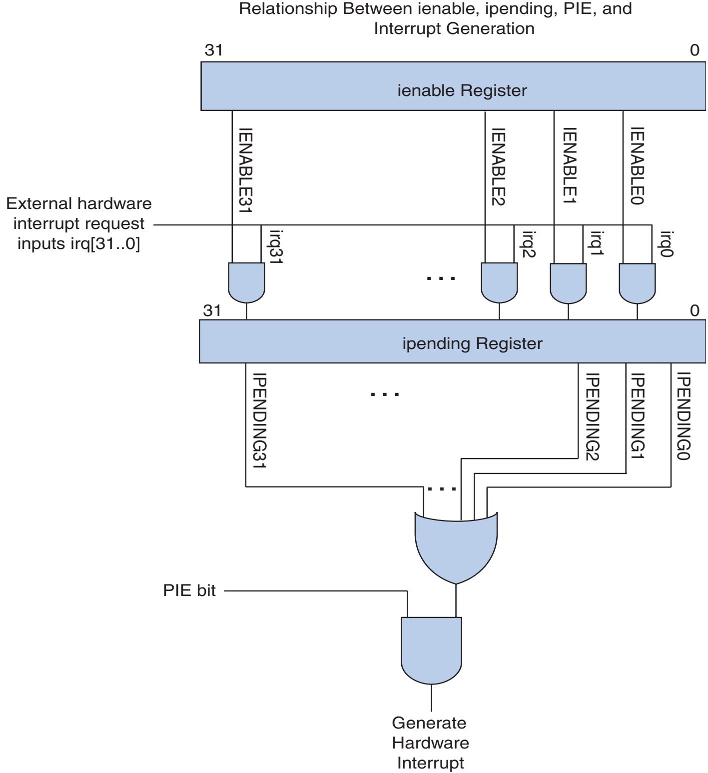
Hardware Interrupt

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

- The PIE bit of the `status` register (`ctl0`) is 1
- An interrupt-request input, `irq`*n*, is asserted
- The corresponding bit *n* of the `ienable` register (`ctl3`) is 1.

Upon hardware interrupt the PIE bit is set to 0, disabling further interrupts. The value of the `ipending` register (`ctl4`) 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 3–1 shows the relationship between `ipending`, `ienable`, PIE, and the generation of an interrupt.

Figure 3–1. Relationship Between ienable, ipending, PIE and Hardware Interrupts



A software exception routine determines which of the pending interrupts has the highest priority, and then transfers control to the appropriate *interrupt service routine (ISR)*. The ISR must stop 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 must also save `estatus` (`ctl1`) and `ea` (`r29`) before re-enabling PIE.

Interrupts can be re-enabled by writing 1 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. See "Nested Exceptions" on page 3–10 for more information.

Software Trap

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

Unimplemented Instruction

When the processor issues a valid instruction that is not implemented in hardware, an unimplemented instruction exception is generated. The exception handler determines which instruction generated the exception. If the instruction is not implemented in hardware, control is passed to an exception routine that emulates the operation in software. See "Potential Unimplemented Instructions" on page 3–21 for more information.

"Unimplemented instruction" does not mean "invalid instruction." Processor behavior for undefined, i.e., invalid, instruction words is dependent on the Nios II core. For most Nios II core implementations, executing an invalid instruction produces an undefined result. See the *Nios II Core Implementation Details* chapter of the *Nios II Processor Reference Handbook* for details.

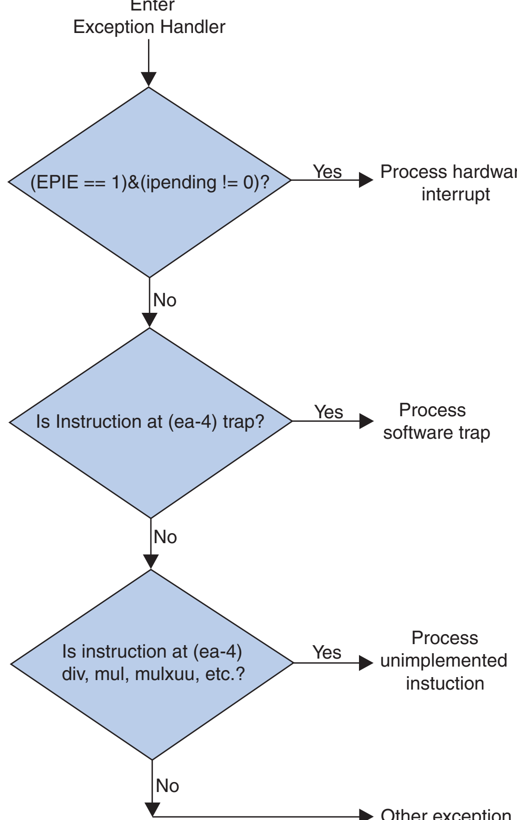
Other Exceptions

The previous sections describe all of the exception types defined by the Nios II architecture at the time of publishing. However, some processor implementations might generate exceptions that do not fall into the above categories. For example, a future implementation might provide a memory management unit (MMU) that generates access violation exceptions. Therefore, a robust exception handler should provide a safe response (such as issuing a warning) in the event that it cannot exactly identify the cause of an exception.

Determining the Cause of Exceptions

The exception handler must determine the cause of each exception and then transfer control to an appropriate exception routine. Figure 3–2 shows an example of the process used to determine the exception source.

Figure 3–2. Process to Determine the Cause of an Exception



If the EPIE bit of the `estatus` register (`ctl1`) is 1 and the value of the `ipending` register (`ctl4`) is non-zero, the exception was caused by an external hardware interrupt. Otherwise, the exception might be caused by a software trap or an unimplemented instruction. To distinguish between software traps and unimplemented instructions, read the instruction at address `ea`–4 (the Nios II data master must have access to the code memory to read this address). If the instruction is `trap`, the exception is a software trap. If the instruction at address `ea-4` is one of the instructions that can be implemented in software, the exception was caused by an unimplemented instruction. See "Potential Unimplemented Instructions" on page 3–21 for details. If none of the above conditions apply, the exception type is unrecognized, and the exception handler should report the condition.

Nested Exceptions

Exception routines must take special precautions before:

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

Before allowing any of these actions, the exception routine must save `estatus` (`ctl1`) and `ea` (r29), so that they can be restored properly before returning.

Returning from an Exception

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

When executing the `eret` instruction, the processor:

1. Copies the contents of `estatus` (`ctl1`) to `status` (`ctl0`)

2. Transfers program execution to the address in the `ea` register (r29)

Return Address

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

When returning from software trap and unimplemented instruction exceptions, execution must resume from the instruction following the software trap or unimplemented instruction. 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.

Break Processing

A break is a transfer of control away from a program's normal flow of execution caused by a `break` instruction or the JTAG debug module. Software debugging tools can take control of the Nios II processor via the JTAG debug module. Only debugging tools control the processor when executing in debug mode; application and system code never execute in this mode.

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

Processing a Break

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

- The processor issues the `break` instruction
- The JTAG debug module asserts a hardware break

A break causes the processor to take the following steps:

1. Stores the contents of the `status` register (`ctl0`) to `bstatus` (`ctl2`)

2. Clears the PIE bit of the `status` register, disabling external processor interrupts

3. Writes the address of the instruction following the break to the `ba` register (`r30`).

4. Transfers execution to the address of the *break handler*. The address of the break handler is specified at system generation time.

Returning from a Break

After performing break processing, the debugging tool releases control of the processor by executing a `bret` instruction. The `bret` instruction restores `status` and returns program execution to the address in `ba`.

Register Usage

The break handler can use `bt` (`r25`) to help save additional registers. Aside from `bt`, all other registers are guaranteed to be returned to their pre-break state after returning from the break-processing routine.

Memory and Peripheral Access

Nios II addresses are 32 bits, allowing access up to a 4 gigabyte address space. However, many Nios II core implementations restrict addresses to 31 bits or fewer.

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

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

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

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

Addressing Modes

The Nios II architecture supports the following addressing modes:

- Register addressing
- Displacement addressing
- Immediate addressing
- Register indirect addressing
- Absolute addressing

In register addressing, all operands are registers, and results are stored back to a register. In displacement addressing, the address is calculated as the sum of a register and a signed, 16-bit immediate value. In immediate addressing, the operand is a constant within the instruction itself. Register indirect addressing uses displacement addressing, but the displacement is the constant 0. Limited-range absolute addressing is achieved by using displacement addressing with register `r0`, whose value is always 0x00.

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.

Some Nios II processor cores support a mechanism called bit-31 cache bypass to bypass the cache depending on the value of the most-significant bit of the address. The address space of these processor implementations is 2 GBytes, and the high bit of the address controls the caching of data memory accesses.

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

Code written for a processor core with cache memory behaves correctly on a processor core without cache memory. The reverse is not true. Therefore, for a program to work properly on all 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, see 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. See "Processor Reset State" on page 3–13 for details.

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

Processor Reset State

After reset, the Nios II processor:

1. Clears the `status` register to 0x0.
2. Invalidates the instruction-cache line associated with the *reset address*, the address of the reset routine.
3. Begins executing from the reset address.

Clearing `status` (`ctl0`) disables hardware interrupts. Invalidating the reset cache line guarantees that instruction fetches for reset code comes from uncached memory. The reset address is specified at system generation time.

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

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

- General-purpose registers, except for `zero` (`r0`) which is permanently zero.
- Control registers, except for `status` (`ctl0`) which is reset to 0x0.
- Instruction and data memory.
- Cache memory, except for the instruction-cache line associated with the reset address.
- Peripherals. Refer to the appropriate peripheral data sheet or specification for reset conditions.
- Custom instruction logic. Refer to the custom instruction specification for reset conditions.

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 3–4 describes the wide (32-bit) load and store instructions.

Table 3–4. Wide Data Transfer Instructions

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

The data transfer instructions in Table 3–5 support byte and half-word transfers.

Table 3–5. Narrow Data Transfer Instructions

Instruction	Description

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

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

Arithmetic and Logical Instructions

Logical instructions support `and`, `or`, `xor`, and `nor` operations. Arithmetic instructions support addition, subtraction, multiplication, and division operations. See Table 3–6.

Table 3–6. Arithmetic and Logical Instructions

Instruction	Description

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

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

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

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

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

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

Move Instructions

These instructions provide move operations to copy the value of a register or an immediate value to another register. See Table 3–7.

Table 3–7. Move Instructions

Instruction	Description

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

Comparison Instructions

The Nios II architecture supports a number of comparison instructions. All of these compare two registers or a register and an immediate value, and write either 1 (if true) or 0 to the result register. These instructions perform all the equality and relational operators of the C programming language. See Table 3–8.

Table 3–8. Comparison Instructions (Part 1 of 2)

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

Table 3–8. Comparison Instructions (Part 2 of 2)

Instruction	Description
`cmpltu`	unsigned <

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

Shift and Rotate Instructions

The following instructions provide shift and rotate operations. The number of bits to rotate or shift can be specified in a register or an immediate value. See Table 3–9.

Table 3–9. Shift and Rotate Instructions

Instruction	Description

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

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

Program Control Instructions

The Nios II architecture supports the unconditional jump and call instructions listed in Table 3–10. These instructions do not have delay slots.

Table 3–10. Unconditional Jump and Call Instructions

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

The conditional-branch instructions compare register values directly, and branch if the expression is true. See Table 3–11. The conditional branches support the equality and relational comparisons of the C programming language:

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

The conditional-branch instructions do not have delay slots.

Table 3–11. Conditional-Branch Instructions

Instruction	Description

`bge`
`bgeu`
`bgt`
`bgtu`
`ble`
`bleu`
`blt`
`bltu`
`beq`
`bne`
	These instructions provide relative branches that compare two register values and branch if the expression is true. See “Comparison Instructions” on page 3–17 for a description of the relational operations implemented.

Other Control Instructions

Table 3–12 shows other control instructions.

Table 3–12. Other Control Instructions

Instruction	Description
trap eret	The trap and eret instructions generate and return from exceptions. These instructions are similar to the call/ret pair, but are used for exceptions. trap saves the status register in the estatus register, saves the return address in the ea register, and then transfers execution to the exception handler. eret returns from exception processing by restoring status from estatus, and executing the instruction specified by the address in ea.
break bret	The break and bret instructions generate and return from breaks. break and bret are used exclusively by software debugging tools. Programmers never use these instructions in application code.
rdctl wrctl	These instructions read and write control registers, such as the status register. The value is read from or stored to a general-purpose register.
flushd flushi initd initi	These instructions are used to manage the data and instruction cache memories.
flushp	This instruction flushes all pre-fetched instructions from the pipeline. This is necessary before jumping to recently-modified instruction memory.
sync	This instruction ensures that all previously-issued operations have completed before allowing execution of subsequent load and store operations.

Custom Instructions

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

For further details, see the “Custom Instructions” section of the *Processor Architecture* chapter of the *Nios II Processor Reference Handbook* and the *Nios II Custom Instruction User Guide*.

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

No-Operation Instruction

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

Potential Unimplemented Instructions

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

- `mul`
- `muli`
- `mulxss`
- `mulxsu`
- `mulxuu`
- `div`
- `divu`

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. See "Unimplemented Instruction " on page 3–8 for details regarding unimplemented instruction processing.

Referenced Documents

This chapter references the following documents:

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

Document Revision History

Table 3–13 shows the revision history for this document.

Table 3–13. Document Revision History

Date & Document Version	Changes Made	Summary of Changes
May 2007 v7.1.0	• Added table of contents to Introduction section. • Added Referenced Documents section.	
March 2007 v7.0.0	No change from previous release.	
November 2006 v6.1.0	No change from previous release.	
May 2006 v6.0.0	No change from previous release.	
October 2005 v5.1.0	No change from previous release.	
May 2005 v5.0.0	No change from previous release.	
September 2004 v1.1	• Added details for new control register `ctl5`. • Updated details of debug mode and break processing to reflect new behavior of the `break` instruction.	
May 2004 v1.0	Initial release.	

4. Implementing the Nios II Processor in SOPC Builder

NII51004-7.1.0

Introduction

This chapter describes the Nios® II Processor MegaWizard interface in SOPC Builder. This chapter contains the following sections:

- "Core Nios II Page" on page 4–2
- "Caches and Memory Interfaces Page" on page 4–6
- "Advanced Features Page" on page 4–9
- "JTAG Debug Module Page" on page 4–11
- "Custom Instructions Page" on page 4–14

The Nios II Processor MegaWizard interface allows you to specify the processor features for a particular Nios II hardware system. This chapter covers only the features of the Nios II processor that you can configure with the Nios II Processor MegaWizard. It is not a user guide for creating complete Nios II processor systems.

To get started using SOPC Builder to design custom Nios II systems, refer to the *Nios II Hardware Development Tutorial*. Nios II development kits also provide a number of ready-made example hardware designs that demonstrate several different configurations of the Nios II processor.

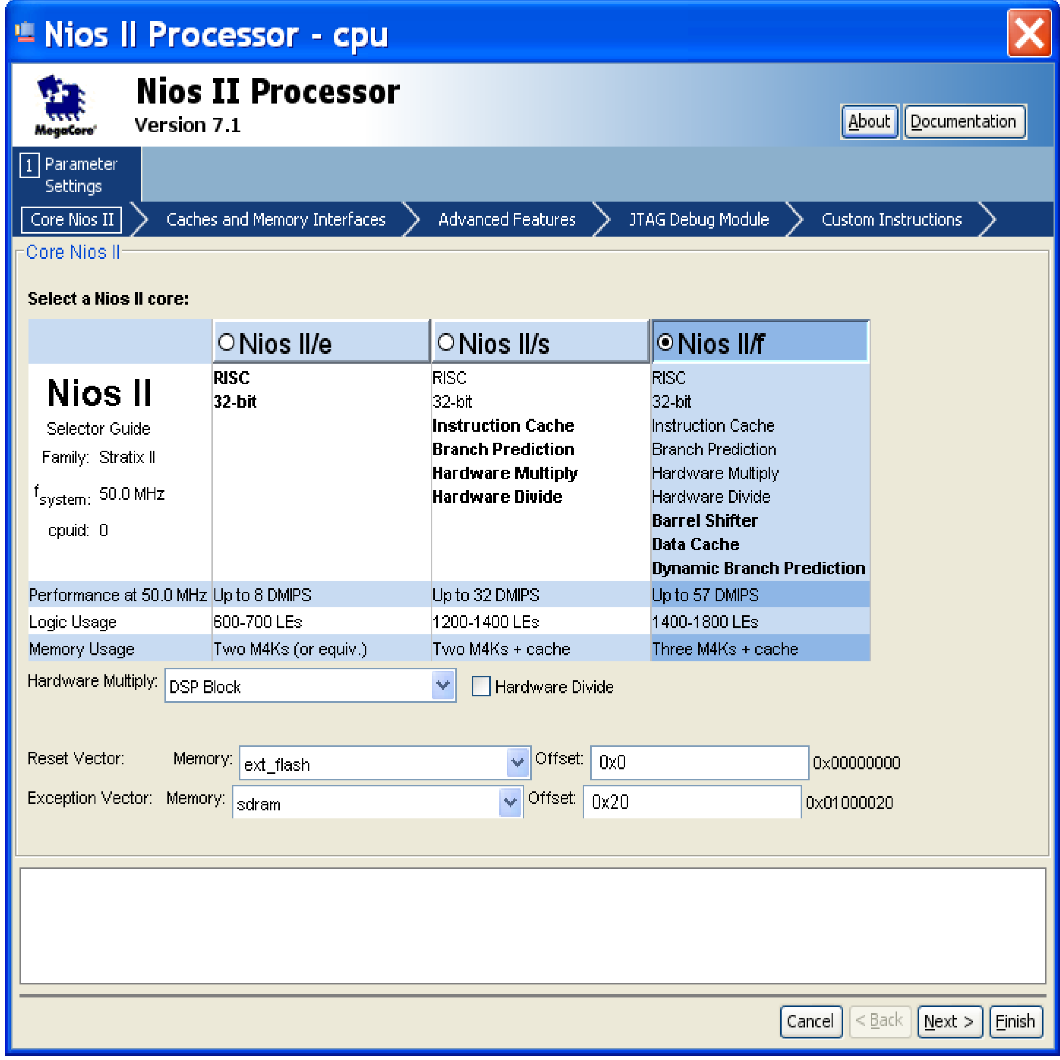
The Nios II Processor MegaWizard interface has several pages. The following sections describe the settings available on each page.

Due to evolution and improvement of the Nios II Processor MegaWizard interface, the figures in this chapter might not match the exact screens that appear in SOPC Builder.

Core Nios II Page

The **Core Nios II** page presents the main settings for configuring the Nios II processor. Figure 4–1 shows an example of the **Core Nios II** page.

Figure 4–1. Core Nios II Page in the Nios II Processor MegaWizard



The following sections describe the configuration settings available.

Core Selection

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

Currently, Altera® offers three 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.

As shown in Figure 4–1, the **Core Nios II** page displays a "selector guide" table that lists the basic properties of each core.

For complete details of each core, see 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 different hardware multiply and divide options. You can choose the best option to balance embedded multiplier usage, logic element (LE) usage, and performance.

The **Hardware Multiply** setting provides the following options:

- **DSP Block** - Include DSP block multipliers in the arithmetic logic unit (ALU). This option is only present when targeting devices that have DSP block multipliers.
- **Embedded Multipliers** - Include embedded multipliers in the ALU. This option is only present when targeting 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.
- **None** - This option conserves logic resources by eliminating multiply hardware. Multiply operations are implemented in software.

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 details on the performance effects of the **Hardware Multiply** and **Hardware Divide** options, see the *Nios II Core Implementation Details* chapter of the *Nios II Processor Reference Handbook*.

Reset and Exception Vectors

The Nios II processor has settings which cannot be configured until other system components are in place. These settings include:

- Reset Vector
- Exception Vector

The following sections describe each system-dependent setting found on the **Core Nios II** page.

Reset Vector

You can select the memory module where the reset code (boot loader) resides, and the location of the reset vector (reset address).

The **Memory** allows you to select the reset vector memory module, which includes all memory modules mastered by the Nios II processor. In a typical system, you select a nonvolatile memory module for the reset code.

Offset allows you to specify the location of the reset vector relative to the memory module's base address. SOPC Builder calculates the physical address of the reset vector when you modify the memory module, the offset, or the memory module's base address.

Exception Vector

You can select the memory module where the exception vector (exception address) resides, and the location of the exception vector.

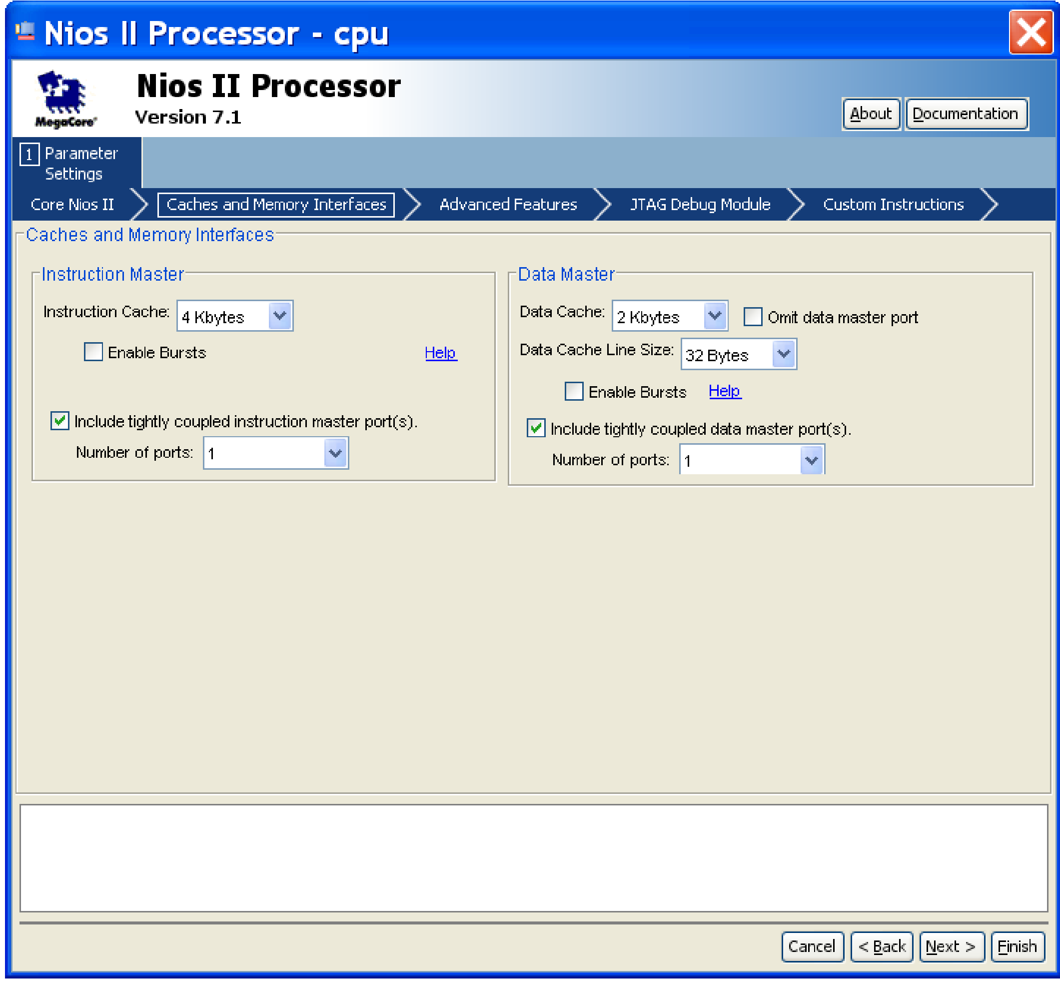
The **Memory** list allows you to select the exception vector memory module, which includes all memory modules mastered by the Nios II processor. In a typical system, you select a low-latency memory module for the exception code.

Offset allows you to specify the location of the exception vector relative to the memory module's base address. SOPC Builder calculates the physical address of the exception vector when you modify the memory module, the offset, or the memory module's base address.

Caches and Memory Interfaces Page

The **Caches and Memory Interfaces** page allows you to configure the cache and tightly-coupled memory usage for the instruction and data master ports. Figure 4–2 shows an example of the **Caches and Memory Interfaces** page.

Figure 4–2. Caches and Memory Interfaces Page in the Nios II Processor MegaWizard



The following sections describe the configuration settings available.

Instruction Master Settings

The **Instruction Master** settings 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.

- **Enable Bursts** - 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 f_{MAX}.

  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.

- **Include tightly coupled instruction master port(s)** - When turned on, the Nios II processor includes tightly-coupled memory ports. You can specify one to four ports with the **Number of ports** setting. Tightly-coupled memory ports appear on the connection panel of the Nios II processor in the SOPC Builder **System Contents** tab. You must connect each port to exactly one memory component in the system.

Data Master Settings

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

- **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, 16, or 32 bytes.
  - **Omit data master port** - If you set **Data Cache** to **None**, you can optionally turn on **Omit data master port** to remove the Avalon-MM data master port from the Nios II processor. In this case, you must include a tightly-coupled data memory.

  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.

  Enable Bursts - 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 f_{MAX}.

  Bursting is only enabled for data 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.

- **Include tightly coupled data master port(s)** - When turned on, the Nios II processor includes tightly-coupled memory ports. You can specify one to four ports with the **Number of ports** setting. Tightly-coupled memory ports appear on the connection panel of the Nios II processor in the SOPC Builder **System Contents** tab. You must connect each port to exactly one memory component in the system.

Advanced Features Page

The **Advanced Features** page allows you to enable specialized features of the Nios II processor. It contains one option: **Include cpu_resetrequest and cpu_resettaken signals**. This option adds processor-only reset request signals to the Nios II processor. These signals let another device individually reset the Nios II processor without resetting the entire SOPC Builder system. The signals are exported to the top level of your SOPC Builder system.

Figure 4–3 on page 4–10 shows the **Advanced Features** page.

Figure 4–3. Advanced Features Page in the Nios II Processor MegaWizard

For further details on the processor-only reset request signals, refer to the *Processor Architecture* chapter of the *Nios II Processor Reference Handbook*.

JTAG Debug Module Page

The **JTAG Debug Module** page presents settings for configuring the JTAG debug module on the Nios II processor. You can select the debug features appropriate for your target application.

Soft-core processors such as the Nios II processor offer unique debug capabilities beyond the features of traditional-fixed processors. The soft-core 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 4–1 describes the debug features available to you for debugging your system.

Table 4–1. Debug Configuration Features

Feature	Description
JTAG Target Connection	Connects to the processor through the standard JTAG pins on the Altera FPGA. This provides the basic capabilities to start and stop the processor, and examine/edit registers and memory.
Download Software	Downloads executable code to the processor's memory via the JTAG connection.
Software Breakpoints	Sets a breakpoint on instructions residing in RAM
Hardware Breakpoints	Sets a breakpoint on instructions residing in nonvolatile memory, such as flash memory.
Data Triggers	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.
Instruction Trace	Captures the sequence of instructions executing on the processor in real time.
Data Trace	Captures the addresses and data associated with read and write operations executed by the processor in real time.
On-Chip Trace	Stores trace data in on-chip memory.
Off-Chip Trace	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 First Silicon Solutions (FS2) or Lauterbach.

The following sections describe the configuration settings available.

Debug Level Settings

There are five debug levels in the **JTAG Debug Module** page as shown in Figure 4–4.

Figure 4–4. JTAG Debug Module Page in the Nios II Processor MegaWizard

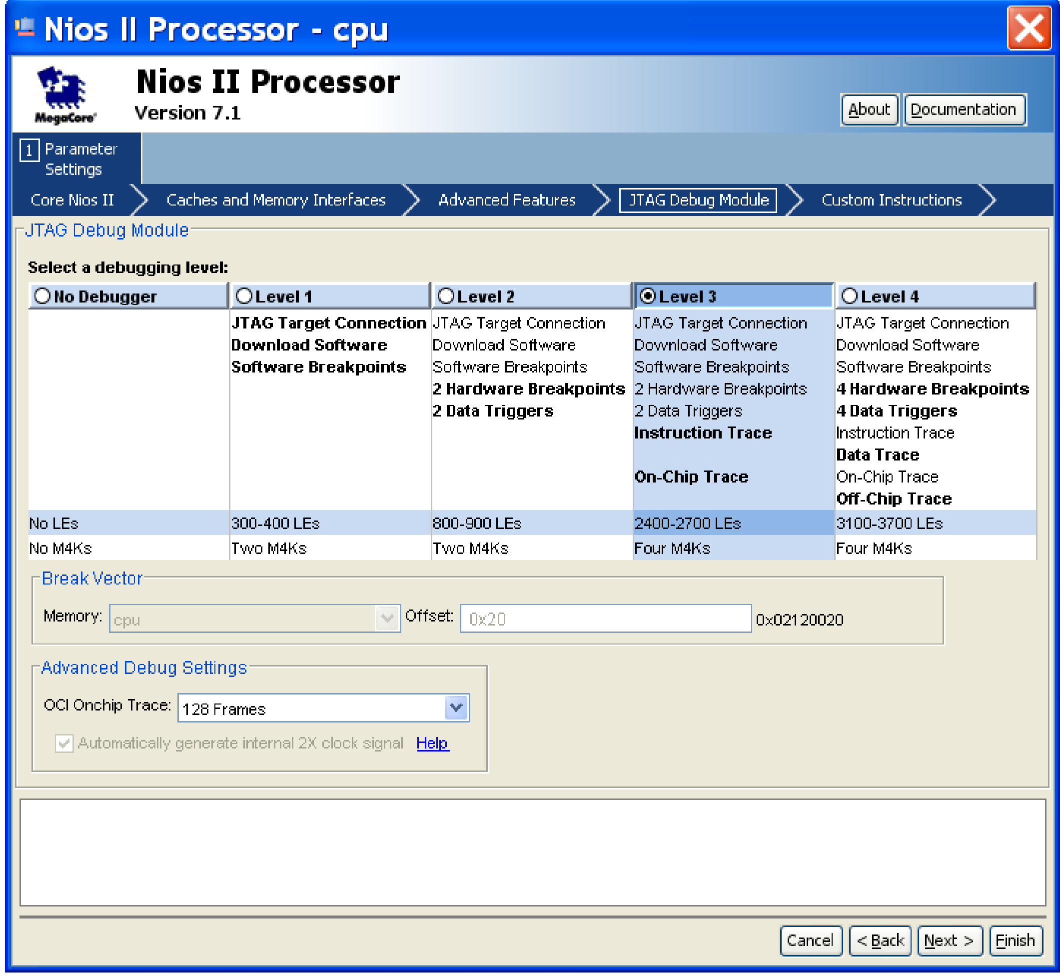


Table 4–2 on page 4–13 is a detailed list of the characteristics of each debug level. Different levels consume different amounts of on-chip resources. Certain Nios II cores have restricted debug options, and certain options require debug tools provided by First Silicon Solutions (FS2) or Lauterbach.

For details on the debug features available from FS2, visit **www.fs2.com**, and from Lauterbach, see **www.lauterbach.com**.

Table 4–2. JTAG Debug Module Levels

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

Notes to Table 4–2:
(1) Level 4 requires the purchase of a software upgrade from FS2 or Lauterbach.
(2) Not including the dedicated JTAG pins on the Altera FPGA.
(3) An additional license from FS2 is required to use more than 16 frames.
(4) Off-chip trace requires the purchase of additional hardware from FS2 or Lauterbach.

Break Vector

If the Nios II processor contains a JTAG debug module, SOPC Builder determines a break vector (break location). **Memory** is always the processor core you are configuring. **Offset** is fixed at `0x20`. SOPC Builder calculates the physical address of the break vector from the memory module's base address and the offset.

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.

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

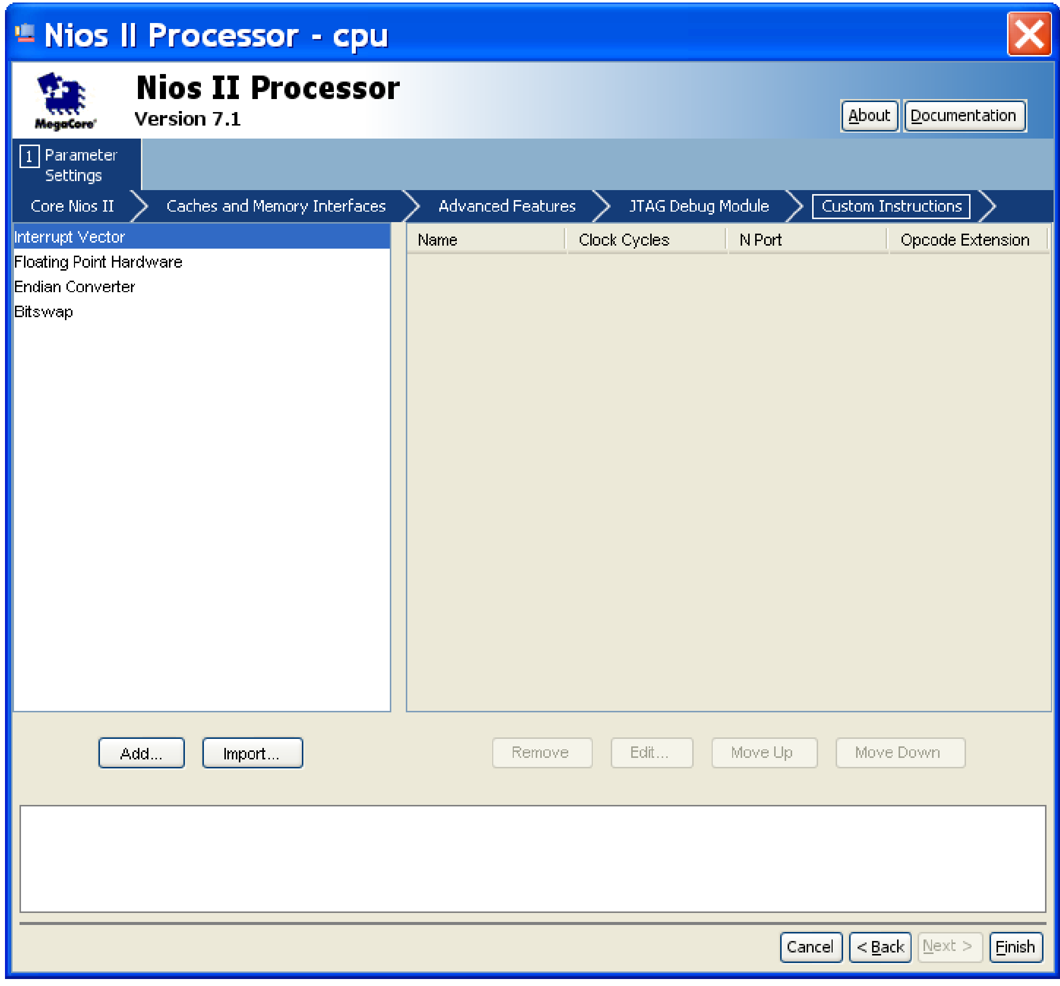
For further details on trace frames, refer to the *Processor Architecture* chapter of the *Nios II Processor Reference Handbook*.

Custom Instructions Page

The **Custom Instructions** page allows you to connect custom instruction logic to the Nios II arithmetic logic unit (ALU). You can achieve significant performance improvements, often on the order of 10x to 100x,

by implementing performance-critical operations in hardware using custom-instruction logic. Figure 4–5 shows an example of the **Custom Instructions** page.

Figure 4–5. Custom Instructions Page in the Nios II Processor MegaWizard



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.

To display custom instructions in the table of active components on the SOPC Builder **System Contents** tab, click **Filter** in the lower-right of the **System Contents** tab, and turn on **Nios Custom Instruction**.

To create your own custom instruction using the component editor, click **Import**. After finishing in the component editor, click **Refresh Component List** on the File menu to add the new instruction to the list at the left side of the **Custom Instructions** page.

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, see the *Nios II Custom Instruction User Guide*.

The following sections describe the default custom instructions Altera provides.

Interrupt Vector Custom Instruction

The Nios II processor offers an interrupt vector custom instruction which reduces average and worst case interrupt latency.

To add the interrupt vector custom instruction to the Nios II processor, select **Interrupt Vector** from the list, and click **Add**.

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.

For details of the interrupt vector custom instruction implementation, see the *Exception and Interrupt Controller* section in the *Processor Architecture* chapter of the *Nios II Processor Reference Handbook*. For guidance with tightly-coupled memories, see the *Tightly-Coupled Memory* section in the *Processor Architecture* chapter of the *Nios II Processor Reference Handbook*.

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, select **Floating Point Hardware** from the list, and click **Add**. By default, SOPC Builder includes floating-point addition, subtraction, and multiplication, but omits the more resource intensive floating-point division. The **Nios II Floating Point Hardware** dialog box, shown in Figure 4–6, appears, giving you the option to include the floating-point division hardware.

Figure 4–6. Nios II Floating Point Hardware Dialog Box

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 details on floating-point instruction usage in the Nios II Embedded Design Suite (EDS), see the *Using Nios II Floating-Point Custom Instructions* tutorial.

Endian Converter Custom Instruction

The Nios II processor core offers an endian converter custom instruction to reduce the time spent performing byte reversal operations.

To add the endian converter custom instruction to the Nios II processor, select **Endian Converter** from the list, and click **Add**.

The endian converter custom instruction takes a 32 bit value and converts the endianness in a single clock cycle. The Nios II processor core supports little endian so this custom instruction allows you to convert data shared with a big endian processor core. It is important to note that this custom instruction does not convert the Nios II processor core to big endian architecture, it only converts big endian data to little endian and visa versa.

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

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, select **Bitswap** from the list, 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 integrating the bitswap custom instruction into your own algorithm, see the *Nios II Custom Instruction User Guide*.

Referenced Documents

This chapter references the following documents:

- *Nios II Hardware Development Tutorial*
- *Nios II Core Implementation Details* chapter of the *Nios II Processor Reference Handbook*
- *Processor Architecture* chapter of the *Nios II Processor Reference Handbook*
- *Nios II Custom Instruction User Guide*
- *Using Nios II Floating-Point Custom Instructions*

Document Revision History

Table 4–3 shows the revision history for this document.

Table 4–3. Document Revision History

Date & Document Version	Changes Made	Summary of Changes
May 2007 v7.1.0	• Revised to reflect new MegaWizard interface. • Added “Endian Converter Custom Instruction” on page 4–18 and “Bitswap Custom Instruction” on page 4–19. • Added table of contents to Introduction section. • Added Referenced Documents section.	
March 2007 v7.0.0	No change from previous release.	
November 2006 v6.1.0	• Add section on interrupt vector custom instruction. • Add section on system-dependent Nios II processor settings.	
May 2006 v6.0.0	• Added details on floating point custom instructions. • Added section on Advanced Features tab.	
October 2005 v5.1.0	No change from previous release.	
May 2005 v5.0.0	• Updates to reflect new GUI options in Nios II processor version 5.0. • New details in “Caches and Tightly-Coupled Memory” section.	
September 2004 v1.1	• Updates to reflect new GUI options in Nios II processor version 1.1. • New details in section “Multiply and Divide Settings.”	
May 2004 v1.0	Initial release.	

Section II. Appendices

This section provides additional information about the Nios® II processor.

This section includes the following chapters:

- Chapter 5, Nios II Core Implementation Details
- Chapter 6, Nios II Processor Revision History
- Chapter 7, Application Binary Interface
- Chapter 8, Instruction Set Reference

5. Nios II Core Implementation Details

NII51015-7.1.0

Introduction

This document describes all of the Nios® II processor core implementations available at the time of publishing. This document describes only implementation-specific features of each processor core. All cores support the Nios II instruction set architecture.

For more information regarding the Nios II instruction set architecture, refer to the *Instruction Set Reference* chapter of the *Nios II Processor Reference Handbook*.

For common core information and details on a specific core, see the appropriate section:

- "Device Family Support" on page 5–3
- "Nios II/f Core" on page 5–3
- "Nios II/s Core" on page 5–12
- "Nios II/e Core" on page 5–19

Table 5–1 compares the objectives and features of each Nios II processor core. The table is designed to help system designers choose the core that best suits their target application.

Table 5–1. Nios II Processor Cores (Part 1 of 2)

Feature		Core		
		Nios II/e	Nios II/s	Nios II/f
Objective		Minimal core size	Small core size	Fast execution speed
Performance	DMIPS/MHz *(1)*	0.15	0.74	1.16
	Max. DMIPS *(2)*	31	127	218
	Max. f_{MAX} *(2)*	200 MHz	165 MHz	185 MHz
Area		< 700 LEs; < 350 ALMs	< 1400 LEs; < 700 ALMs	< 1800 LEs; < 900 ALMs
Pipeline		1 Stage	5 Stages	6 Stages
External Address Space		2 Gbytes	2 GBytes	2 GBytes

Table 5–1. Nios II Processor Cores (Part 2 of 2)

Feature		Core		
		Nios II/e	Nios II/s	Nios II/f
Instruction Bus	Cache	–	512 bytes to 64 kbytes	512 bytes to 64 kbytes
	Pipelined Memory Access	–	Yes	Yes
	Branch Prediction	–	Static	Dynamic
	Tightly-Coupled Memory	–	Optional	Optional
Data Bus	Cache	–	–	512 bytes to 64 Kbytes
	Pipelined Memory Access	–	–	–
	Cache Bypass Methods	–	–	I/O instructions; bit-31 cache bypass
	Tightly-Coupled Memory	–	–	Optional
Arithmetic Logic Unit	Hardware Multiply	–	3-Cycle *(3)*	1-Cycle *(3)*
	Hardware Divide	–	Optional	Optional
	Shifter	1 Cycle-per-bit	3-Cycle Shift *(3)*	1-Cycle Barrel Shifter *(3)*
JTAG Debug Module	JTAG interface, run control, software breakpoints	Optional	Optional	Optional
	Hardware Breakpoints	–	Optional	Optional
	Off-Chip Trace Buffer	–	Optional	Optional
Exception Handling	Exception Types	Software trap, unimplemented instruction, hardware interrupt	Software trap, unimplemented instruction, hardware interrupt	Software trap, unimplemented instruction, hardware interrupt
	Integrated Interrupt Controller	Yes	Yes	Yes
User Mode Support		No; Permanently in supervisor mode	No; Permanently in supervisor mode	No; Permanently in supervisor mode
Custom Instruction Support		Yes	Yes	Yes

Notes to Table 5–1:

(1) DMIPS performance for the Nios II/s and Nios II/f cores depends on the hardware multiply option.
(2) Using the fastest hardware multiply option, and targeting a Stratix II FPGA in the fastest speed grade.
(3) Multiply and shift performance depends on which hardware multiply option is used. If no hardware multiply option is used, multiply operations are emulated in software, and shift operations require one cycle per bit. For details, see the arithmetic logic unit description for each core.

Device Family Support

All Nios II cores provide the same support for target Altera device families. Nios II cores provide either full or preliminary device family support, as described below:

- *Full support* means the Nios II cores meet all functional and timing requirements for the device family and may be used in production designs
- *Preliminary support* means the Nios II cores meet all functional requirements, but may still be undergoing timing analysis for the device family; they may be used in production designs with caution.

Table 5–2 shows the level of support offered to each of the Altera device families by the Nios II cores.

Table 5–2. Device Family Support

Device Family	Support
Arria™ GX	Preliminary
Stratix® III	Preliminary
Stratix II	Full
Stratix II GX	Full
Stratix GX	Full
Stratix	Full
Hardcopy® II	Full
HardCopy	Full
Cyclone™ III	Preliminary
Cyclone™ II	Full
Cyclone	Full
Other device families	No support

Nios II/f Core

The Nios II/f "fast" core is designed for high execution performance. Performance is gained at the expense of core size, making the Nios II/f core approximately 25% larger than the Nios II/s core. Altera designed the Nios II/f core with the following design goals in mind:

- Maximize the instructions-per-cycle execution efficiency
- Maximize f_{MAX} performance of the processor core

The resulting core is optimal for performance-critical applications, as well as for applications with large amounts of code and/or data, such as systems running a full-featured operating system.

Overview

The Nios II/f core:

- Has separate instruction and data caches
- Can access up to 2 GBytes of external address space
- Supports optional tightly-coupled memory for instructions and data
- Employs a 6-stage pipeline to achieve maximum DMIPS/MHz
- Performs dynamic branch prediction
- Provides hardware multiply, divide, and shift options to improve arithmetic performance
- Supports the addition of custom instructions
- Supports the JTAG debug module
- Supports optional JTAG debug module enhancements, including hardware breakpoints and real-time trace

The following sections discuss the noteworthy details of the Nios II/f core implementation. This document does not discuss low-level design issues or implementation details that do not affect Nios II hardware or software designers.

Register File

At system generation time, the `cpuid` control register (`clt5`) is assigned a value that is guaranteed to be unique for each processor in the system.

Arithmetic Logic Unit

The Nios II/f core provides several arithmetic logic unit (ALU) options to improve the performance of multiply, divide, and shift operations.

Multiply and Divide Performance

The Nios II/f core provides the following hardware multiplier options:

- *No hardware multiply* — Does not include multiply hardware. In this case, multiply operations are emulated in software.
- *Use embedded multipliers* — Includes dedicated embedded multipliers available on the target device. This option is available only on Altera FPGAs that have embedded multipliers, such as the DSP blocks in Stratix II FPGAs.
- *Use LE-based multipliers* — Includes hardware multipliers built from logic element (LE) resources.

The Nios II/f core also provides a hardware divide option that includes LE-based divide circuitry in the ALU.

Including an ALU option improves the performance of one or more arithmetic instructions.

The performance of the embedded multipliers differ, depending on the target FPGA family.

Table 5–3 lists the details of the hardware multiply and divide options.

Table 5–3. Hardware Multiply and Divide Details for the Nios II/f Core

ALU Option	Hardware Details	Cycles per Instruction	Result Latency Cycles	Supported Instructions
No hardware multiply or divide	Multiply and divide instructions generate an exception	–	–	None
LE-based multiplier	ALU includes 32 x 4-bit multiplier	11	+2	`mul`, `muli`
Embedded multiplier on Stratix, Stratix II and Stratix III families	ALU includes 32 x 32-bit multiplier	1	+2	`mul`, `muli`, `mulxss`, `mulxsu`, `mulxuu`
Embedded multiplier on Cyclone II and Cyclone III families	ALU includes 32 x 16-bit multiplier	5	+2	`mul`, `muli`
Hardware divide	ALU includes multicycle divide circuit	4 – 66	+2	`div`, `divu`

The cycles per instruction value determines the maximum rate at which the ALU can dispatch instructions and produce each result. The latency value determines when the result becomes available. If there is no data dependency between the results and operands for back-to-back instructions, then the latency does not affect throughput. However, if an instruction depends on the result of an earlier instruction, then the processor stalls through any result latency cycles until the result is ready.

In the following code example, a multiply operation (with 1 instruction cycle and 2 result latency cycles) is followed immediately by an add operation that uses the result of the multiply. On the Nios II/f core, the `addi` instruction, like most ALU instructions, executes in a single cycle. However, in this code example, execution of the `addi` instruction is delayed by two additional cycles until the multiply operation completes.

```
mul r1, r2, r3          ; r1 = r2 * r3
addi r1, r1, 100        ; r1 = r1 + 100 (Depends on result of mul)
```

In contrast, the following code does not stall the processor.

```
mul r1, r2, r3          ; r1 = r2 * r3
or r5, r5, r6           ; No dependency on previous results
or r7, r7, r8           ; No dependency on previous results
addi r1, r1, 100        ; r1 = r1 + 100 (Depends on result of mul)
```

Shift and Rotate Performance

The performance of shift operations depends on the hardware multiply option. When a hardware multiplier is present, the ALU achieves shift and rotate operations in one or two clock cycles. Otherwise, the ALU includes dedicated shift circuitry that achieves one-bit-per-cycle shift and rotate performance. Refer to Table 5–5 on page 5–11 for details.

Memory Access

The Nios II/f core provides both instruction and data caches. The cache size for each is user-definable, between 512 bytes and 64 Kbytes. The Nios II/f core supports the bit-31 cache bypass method for accessing I/O on the data master port. Addresses are 31 bits wide to accommodate the bit-31 cache bypass method.

Instruction and Data Master Ports

The instruction master port is a pipelined Avalon®-MM master port. If the core includes data cache with a line size greater than four bytes, then the data master port is a pipelined Avalon-MM master port. Otherwise, the data master port is not pipelined.

The instruction and data master ports on the Nios II/f core are optional. A master port can be excluded, as long as the core includes at least one tightly-coupled memory to take the place of the missing master port.

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.

Support for pipelined Avalon-MM transfers minimizes the impact of synchronous memory with pipeline latency. The pipelined instruction and data master ports can issue successive read requests before prior requests complete.

Instruction Cache

The instruction cache memory has the following characteristics:

- Direct-mapped cache implementation
- 32 bytes (8 words) per cache line

- The instruction master port reads an entire cache line at a time from memory, and issues one read per clock cycle.
- Critical word first

The instruction byte address is divided into the following fields:

		. . . 5	4 3 2	1	0
tag	line		offset	0	0

The sizes of the line and tag fields depend on the size of the cache memory, but the offset field is always three bits (i.e., an 8-word line). The maximum instruction byte address size is 31 bits.

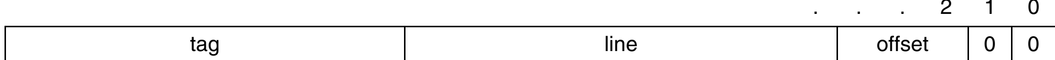
The instruction cache is optional. However, excluding instruction cache from the Nios II/f core requires that the core include at least one tightly-coupled instruction memory.

Data Cache

The data cache memory has the following characteristics:

- Direct-mapped cache implementation
- Configurable line size of 4, 16, or 32 bytes
- The data master port reads an entire cache line at a time from memory, and issues one read per clock cycle.
- Write-back
- Write-allocate (i.e., store instructions that miss allocate the line for that address)

The data byte address is divided into the following fields:



The size of the line and tag fields depend on the size of the cache memory. The size of the offset field depends on the line size. Line sizes of 4, 16, and 32 bytes have offset widths of 0, 2, and 3 bits respectively. The maximum data byte address size is 31 bits.

The data cache is optional. If the data cache is excluded from the core, the data master port can also be excluded.

Cache Bypass

The normal method for bypassing the data cache is to use I/O load and store instructions that bypass the cache. In addition, the Nios II/f core also implements the bit-31 cache bypass method on the data master port. This method 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.

Mixing Cached and Noncached Accesses

Mixing cached and noncached accesses to the same cache line can result in invalid data reads. For example, the following sequence of events causes cache incoherency.

1. The Nios II core writes data to cache, creating a dirty data cache line.

2. The Nios II core reads data from the same address, but bypasses the cache.

Software should not mix both cached and uncached accesses to the same cache line. If it is necessary to mix cached and uncached data accesses, flush the corresponding line of the data cache after completing the cached accesses and before performing the uncached accesses.

Tightly-Coupled Memory

The Nios II/f core provides optional tightly-coupled memory interfaces for both instructions and data. A Nios II/f core can use up to four each of instruction and data tightly-coupled memories. When a tightly-coupled memory interface is enabled, the Nios II core includes an additional memory interface master port. Each tightly-coupled memory interface must connect directly to exactly one memory slave port.

When tightly-coupled memory is present, the Nios II core decodes addresses internally to determine if requested instructions or data reside in tightly-coupled memory. If the address resides in tightly-coupled memory, the Nios II core fetches the instruction or data through the tightly-coupled memory interface. Software accesses tightly-coupled memory with the usual load and store instructions, such as `ldw` or `ldwio`.

Accessing tightly-coupled memory bypasses cache memory. The processor core functions as if cache were not present for the address span of the tightly-coupled memory. Instructions for managing cache, such as `initd` and `flushd`, do not affect the tightly-coupled memory, even if the instruction specifies an address in tightly-coupled memory.

Execution Pipeline

This section provides an overview of the pipeline behavior for the benefit of performance-critical applications. Designers can use this information to minimize unnecessary processor stalling. Most application programmers never need to analyze the performance of individual instructions.

The Nios II/f core employs a 6-stage pipeline. The pipeline stages are listed in Table 5–4.

Table 5–4. Implementation Pipeline Stages for Nios II/f Core

Stage Letter	Stage Name
F	Fetch
D	Decode
E	Execute
M	Memory
A	Align
W	Writeback

Up to one instruction is dispatched and/or retired per cycle. Instructions are dispatched and retired in-order. Dynamic branch prediction is implemented using a 2-bit branch history table. The pipeline stalls for the following conditions:

- Multi-cycle instructions
- Avalon-MM instruction master port read accesses
- Avalon-MM data master port read/write accesses
- Data dependencies on long latency instructions (e.g., load, multiply, shift).

Pipeline Stalls

The pipeline is set up so that if a stage stalls, no new values enter that stage or any earlier stages. No “catching up” of pipeline stages is allowed, even if a pipeline stage is empty.

Only the A-stage and D-stage are allowed to create stalls.

The A-stage stall occurs if any of the following conditions occurs:

- An A-stage memory instruction is waiting for Avalon-MM data master requests to complete. Typically this happens when a load or store misses in the data cache, or a flushd instruction needs to write back a dirty line.
- An A-stage shift/rotate instruction is still performing its operation. This only occurs with the multi-cycle shift circuitry (i.e., when the hardware multiplier is not available).
- An A-stage divide instruction is still performing its operation. This only occurs when the optional divide circuitry is available.
- An A-stage multi-cycle custom instruction is asserting its stall signal. This only occurs if the design includes multi-cycle custom instructions.

The D-stage stall occurs if the following condition occurs and no M-stage pipeline flush is active:

An instruction is trying to use the result of a late result instruction too early. The late result instructions are loads, shifts, rotates, rdctl, multiplies (if hardware multiply is supported), divides (if hardware divide is supported), and multi-cycle custom instructions (if present).

Branch Prediction

The Nios II/f core performs dynamic branch prediction to minimize the cycle penalty associated with taken branches.

Instruction Performance

All instructions take one or more cycles to execute. Some instructions have other penalties associated with their execution. Late result instructions have two cycles placed between them and an instruction that uses their result. Instructions that flush the pipeline cause up to three instructions after them to be cancelled. This creates a three-cycle penalty and an execution time of four cycles. Instructions that require Avalon-MM transfers are stalled until any required Avalon-MM transfers (up to one write and one read) are completed.

Execution performance for all instructions is shown in Table 5–5.

Table 5–5. Instruction Execution Performance for Nios II/f Core

Instruction	Cycles	Penalties
Normal ALU instructions (e.g., `add`, `cmplt`)	1	
Combinatorial custom instructions	1	
Multi-cycle custom instructions	1	Late result
Branch (correctly predicted, taken)	2	
Branch (correctly predicted, not taken)	1	
Branch (mis-predicted)	4	Pipeline flush
`trap`, `break`, `eret`, `bret`, `flushp`, `wrctl`, and unimplemented instructions	4	Pipeline flush
`call`	2	
`jmp`, `ret`, `callr`	3	
`rdctl`	1	Late result
`load` (without Avalon-MM transfer)	1	Late result
`load` (with Avalon-MM transfer)	> 1	Late result
`store`, `flushd` (without Avalon-MM transfer)	1	
`store`, `flushd` (with Avalon-MM transfer)	> 1	
`initd`	1	
`flushi`, `initi`	4	
Multiply	*(1)*	Late result
Divide	*(1)*	Late result
Shift/rotate (with hardware multiply using embedded multipliers)	1	Late result
Shift/rotate (with hardware multiply using LE-based multipliers)	2	Late result
Shift/rotate (without hardware multiply present)	1 - 32	Late result
All other instructions	1	

Note to Table 5–5:
(1) Depends on the hardware multiply or divide option. See Table 5–3 on page 5 for details.

Exception Handling

The Nios II/f core supports the following exception types:

- Hardware interrupt
- Software trap
- Unimplemented instruction

JTAG Debug Module

The Nios II/f core supports the JTAG debug module to provide a JTAG interface to software debugging tools. The Nios II/f core supports an optional enhanced interface that allows real-time trace data to be routed out of the processor and stored in an external debug probe.

Unsupported Features

The Nios II/f core does not handle the execution of instructions with undefined opcodes. If the processor issues an instruction word with an undefined opcode, the resulting behavior is undefined.

Nios II/s Core

The Nios II/s "standard" core is designed for small core size. On-chip logic and memory resources are conserved at the expense of execution performance. The Nios II/s core uses approximately 20% less logic than the Nios II/f core, but execution performance also drops by roughly 40%. Altera designed the Nios II/s core with the following design goals in mind:

- Do not cripple performance for the sake of size.
- Remove hardware features that have the highest ratio of resource usage to performance impact.

The resulting core is optimal for cost-sensitive, medium-performance applications. This includes applications with large amounts of code and/or data, such as systems running an operating system where performance is not the highest priority.

Overview

The Nios II/s core:

- Has instruction cache, but no data cache
- Can access up to 2 Gbytes of external address space
- Supports optional tightly-coupled memory for instructions
- Employs a 5-stage pipeline
- Performs static branch prediction
- Provides hardware multiply, divide, and shift options to improve arithmetic performance
- Supports the addition of custom instructions
- Supports the JTAG debug module
- Supports optional JTAG debug module enhancements, including hardware breakpoints and real-time trace

The following sections discuss the noteworthy details of the Nios II/s core implementation. This document does not discuss low-level design issues, or implementation details that do not affect Nios II hardware or software designers.

Register File

At system generation time, the `cpuid` control register (`clt5`) is assigned a value that is guaranteed to be unique for each processor in the system.

Arithmetic Logic Unit

The Nios II/s core provides several ALU options to improve the performance of multiply, divide, and shift operations.

Multiply and Divide Performance

The Nios II/s core provides the following hardware multiplier options:

- *No hardware multiply* – Does not include multiply hardware. In this case, multiply operations are emulated in software.
- *Use embedded multipliers* – Includes dedicated embedded multipliers available on the target device. This option is available only on Altera FPGAs that have embedded multipliers, such as the DSP blocks in Stratix II FPGAs.
- *Use LE-based multipliers* – Includes hardware multipliers built from logic element (LE) resources.

The Nios II/s core also provides a hardware divide option that includes LE-based divide circuitry in the ALU.

Including an ALU option improves the performance of one or more arithmetic instructions.

The performance of the embedded multipliers differ, depending on the target FPGA family.

Table 5–6 lists the details of the hardware multiply and divide options.

Table 5–6. Hardware Multiply and Divide Details for the Nios II/s Core

ALU Option	Hardware Details	Cycles per instruction	Supported Instructions
No hardware multiply or divide	Multiply and divide instructions generate an exception	–	None
LE-based multiplier	ALU includes 32 x 4-bit multiplier	11	`mul`, `muli`
Embedded multiplier on Stratix, Stratix II and Stratix III families	ALU includes 32 x 32-bit multiplier	3	`mul`, `muli`, `mulxss`, `mulxsu`, `mulxuu`
Embedded multiplier on Cyclone II and Cyclone III families	ALU includes 32 x 16-bit multiplier	5	`mul`, `muli`
Hardware divide	ALU includes multicycle divide circuit	4 – 66	`div`, `divu`

Shift and Rotate Performance

The performance of shift operations depends on the hardware multiply option. When a hardware multiplier is present, the ALU achieves shift and rotate operations in three or four clock cycles. Otherwise, the ALU includes dedicated shift circuitry that achieves one-bit-per-cycle shift and rotate performance. Refer to Table 5–8 on page 5–17 for details.

Memory Access

The Nios II/s core provides instruction cache, but no data cache. The instruction cache size is user-definable, between 512 bytes and 64 Kbytes. The Nios II/s core can address up to 2 Gbyte of external memory. The Nios II/s core does not support bit-31 data cache bypass. The most-significant bit of addresses is ignored.

Instruction and Data Master Ports

The instruction port on the Nios II/s core is optional. The instruction master port can be excluded, as long as the core includes at least one tightly-coupled instruction memory. 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. The pipelined instruction master port can issue successive read requests before prior requests complete.

The data master port on the Nios II/s core is always present.

Instruction Cache

The instruction cache for the Nios II/s core is nearly identical to the instruction cache in the Nios II/f core. The instruction cache memory has the following characteristics:

- Direct-mapped cache implementation
- The instruction master port reads an entire cache line at a time from memory, and issues one read per clock cycle.
- Critical word first

The instruction byte address is divided into the following fields:

	. . . 5	4 3 2	1	0
tag	line	offset	0	0

The size of the line and tag fields depend on the size of the cache memory, but the offset field is always three bits (i.e., an 8-word line). The maximum instruction byte address size is 31 bits.

The instruction cache is optional. However, excluding instruction cache from the Nios II/s core requires that the core include at least one tightly-coupled instruction memory.

Tightly-Coupled Memory

The Nios II/s core provides optional tightly-coupled memory interfaces for instructions. A Nios II/s core can use up to four tightly-coupled instruction memories. When a tightly-coupled memory interface is enabled, the Nios II core includes an additional memory interface master port. Each tightly-coupled memory interface must connect directly to exactly one memory slave port.

When tightly-coupled memory is present, the Nios II core decodes addresses internally to determine if requested instructions reside in tightly-coupled memory. If the address resides in tightly-coupled memory, the Nios II core fetches the instruction through the tightly-coupled memory interface. Software does not require awareness of whether code resides in tightly-coupled memory or not.

Accessing tightly-coupled memory bypasses cache memory. The processor core functions as if cache were not present for the address span of the tightly-coupled memory. Instructions for managing cache, such as `initi` and `flushi`, do not affect the tightly-coupled memory, even if the instruction specifies an address in tightly-coupled memory.

Execution Pipeline

This section provides an overview of the pipeline behavior for the benefit of performance-critical applications. Designers can use this information to minimize unnecessary processor stalling. Most application programmers never need to analyze the performance of individual instructions, and live happy lives without ever studying Table 5–7.

The Nios II/s core employs a 5-stage pipeline. The pipeline stages are listed in Table 5–7.

Table 5–7. Implementation Pipeline Stages for Nios II/s Core

Stage Letter	Stage Name
F	Fetch
D	Decode
E	Execute
M	Memory
W	Writeback

Up to one instruction is dispatched and/or retired per cycle. Instructions are dispatched and retired in-order. Static branch prediction is implemented using the branch offset direction; a negative offset (backward branch) is predicted as taken, and a positive offset (forward branch) is predicted as not-taken. The pipeline stalls for the following conditions:

- Multi-cycle instructions (e.g., shift/rotate without hardware multiply)
- Avalon-MM instruction master port read accesses
- Avalon-MM data master port read/write accesses
- Data dependencies on long latency instructions (e.g., load, multiply, shift operations)

Pipeline Stalls

The pipeline is set up so that if a stage stalls, no new values enter that stage or any earlier stages. No "catching up" of pipeline stages is allowed, even if a pipeline stage is empty.

Only the M-stage is allowed to create stalls.

The M-stage stall occurs if any of the following conditions occurs:

- An M-stage load/store instruction is waiting for Avalon-MM data master transfer to complete.
- An M-stage shift/rotate instruction is still performing its operation when using the multi-cycle shift circuitry (i.e., when the hardware multiplier is not available).
- An M-stage shift/rotate/multiply instruction is still performing its operation when using the hardware multiplier (which takes three cycles).
- An M-stage multi-cycle custom instruction is asserting its stall signal. This only occurs if the design includes multi-cycle custom instructions.

Branch Prediction

The Nios II/s core performs static branch prediction to minimize the cycle penalty associated with taken branches.

Instruction Performance

All instructions take one or more cycles to execute. Some instructions have other penalties associated with their execution. Instructions that flush the pipeline cause up to three instructions after them to be cancelled. This creates a three-cycle penalty and an execution time of four cycles. Instructions that require an Avalon-MM transfer are stalled until the transfer completes.

Execution performance for all instructions is shown in Table 5–8.

Table 5–8. Instruction Execution Performance for Nios II/s Core (Part 1 of 2)

Instruction	Cycles	Penalties
Normal ALU instructions (e.g., `add`, `cmplt`)	1	
Combinatorial custom instructions	1	
Multi-cycle custom instructions	1	
Branch (correctly predicted taken)	2	

Table 5–8. Instruction Execution Performance for Nios II/s Core (Part 2 of 2)

Instruction	Cycles	Penalties
Branch (correctly predicted not taken)	1	
Branch (mispredicted)	4	Pipeline flush
`trap`, `break`, `eret`, `bret`, `flushp`, `wrctl`, unimplemented	4	Pipeline flush
`jmp`, `ret`, `call`, `callr`	4	Pipeline flush
`rdctl`	1	
`load`, `store`	> 1	
`flushi`, `initi`	4	
Multiply	*(1)*	
Divide	*(1)*	
Shift/rotate (with hardware multiply using embedded multipliers)	3	
Shift/rotate (with hardware multiply using LE-based multipliers)	4	
Shift/rotate (without hardware multiply present)	1 to 32	
All other instructions	1	

Note to Table 5–8:

(1) Depends on the hardware multiply or divide option. See Table 5–6 on page 14 for details.

Exception Handling

The Nios II/s core supports the following exception types:

- Hardware interrupt
- Software trap
- Unimplemented instruction

JTAG Debug Module

The Nios II/s core supports the JTAG debug module to provide a JTAG interface to software debugging tools. The Nios II/s core supports an optional enhanced interface that allows real-time trace data to be routed out of the processor and stored in an external debug probe.

Unsupported Features

The Nios II/s core does not handle the execution of instructions with undefined opcodes. If the processor issues an instruction word with an undefined opcode, the resulting behavior is undefined.

Nios II/e Core

The Nios II/e "economy" core is designed to achieve the smallest possible core size. Altera designed the Nios II/e core with a singular design goal: Reduce resource utilization any way possible, while still maintaining compatibility with the Nios II instruction set architecture. Hardware resources are conserved at the expense of execution performance. The Nios II/e core is roughly half the size of the Nios II/s core, but the execution performance is substantially lower.

The resulting core is optimal for cost-sensitive applications, as well as applications that require simple control logic.

Overview

The Nios II/e core:

- Executes at most one instruction per six clock cycles
- Can access up to 2 Gbytes of external address space
- Supports the addition of custom instructions
- Supports the JTAG debug module
- Does not provide hardware support for potential unimplemented instructions
- Has no instruction cache or data cache
- Does not perform branch prediction

The following sections discuss the noteworthy details of the Nios II/e core implementation. This document does not discuss low-level design issues, or implementation details that do not affect Nios II hardware or software designers.

Register File

At system generation time, the `cpuid` control register (`clt5`) is assigned a value that is guaranteed to be unique for each processor in the system.

Arithmetic Logic Unit

The Nios II/e core does not provide hardware support for any of the potential unimplemented instructions. All unimplemented instructions are emulated in software.

The Nios II/e core employs dedicated shift circuitry to perform shift and rotate operations. The dedicated shift circuitry achieves one-bit-per-cycle shift and rotate operations.

Memory Access

The Nios II/e core does not provide instruction cache or data cache. All memory and peripheral accesses generate an Avalon-MM transfer. The Nios II/e core can address up to 2 Gbytes of external memory. The core does not support bit-31 data cache bypass. However, the most-significant bit of addresses is ignored to maintain consistency with Nios II core implementations that do support bit-31 cache bypass method.

Instruction Execution Stages

This section provides an overview of the pipeline behavior as a means of estimating assembly execution time. Most application programmers never need to analyze the performance of individual instructions.

Instruction Performance

The Nios II/e core dispatches a single instruction at a time, and the processor waits for an instruction to complete before fetching and dispatching the next instruction. Because each instruction completes before the next instruction is dispatched, branch prediction is not necessary. This greatly simplifies the consideration of processor stalls. Maximum performance is one instruction per six clock cycles. To achieve six cycles, the Avalon-MM instruction master port must fetch an instruction in one clock cycle. A stall on the Avalon-MM instruction master port directly extends the execution time of the instruction.

Execution performance for all instructions is shown in Table 5–9.

Table 5–9. Instruction Execution Performance for Nios II/e Core

Instruction	Cycles
Normal ALU instructions (e.g., `add`, `cmplt`)	6
`branch`, `jmp`, `ret`, `call`, `callr`	6
`trap, break, eret, bret, flushp, wrctl, rdctl,` unimplemented	6
`load word`	6 + Duration of Avalon-MM read transfer
`load halfword`	9 + Duration of Avalon-MM read transfer
`load byte`	10 + Duration of Avalon-MM read transfer
`store`	6 + Duration of Avalon-MM write transfer
`Shift, rotate`	7 to 38
All other instructions	6
Combinatorial custom instructions	6
Multi-cycle custom instructions	≥6

Exception Handling

The Nios II/e core supports the following exception types:

- Hardware interrupt
- Software traps
- Unimplemented instruction

JTAG Debug Module

The Nios II/e core supports the JTAG debug module to provide a JTAG interface to software debugging tools. The JTAG debug module on the Nios II/e core does not support hardware breakpoints or trace.

Unsupported Features

The Nios II/e core does not handle the execution of instructions with undefined opcodes. If the processor issues an instruction word with an undefined opcode, the resulting behavior is undefined.

Referenced Documents

This chapter references the following documents:

- *Instruction Set Reference* chapter of the *Nios II Processor Reference Handbook*

Document Revision History

Table 5–10 shows the revision history for this document.

Table 5–10. Document Revision History

Date & Document Version	Changes Made	Summary of Changes
May 2007 v7.1.0	● Added table of contents to Introduction section. ● Added Referenced Documents section.	
March 2007 v7.0.0	Add preliminary Cyclone III device family support	Cyclone III device family
November 2006 v6.1.0	Add preliminary Stratix III device family support	Stratix III device family
May 2006 v6.0.0	Performance for `flushi` and `initi` instructions changes from 1 to 4 cycles for Nios II/s and Nios II/f cores.	
October 2005 v5.1.0	No change from previous release.	
May 2005 v5.0.0	Updates to Nios II/f and Nios II/s cores. Added tightly-coupled memory and new data cache options. Corrected cycle counts for shift/rotate operations.	
December 2004 v1.2	Updates to Multiply and Divide Performance section for Nios II/f and Nios II/s cores.	
September 2004 v1.1	Updates for Nios II 1.01 release.	
May 2004 v1.0	Initial release.	

6. Nios II Processor Revision History

NII51018-7.1.0

Introduction

Each release of the Nios® II Embedded Design Suite (EDS) introduces improvements to the Nios II processor, the software development tools, or both. This document catalogs the history of revisions to the Nios II processor; it does not track revisions to development tools, such as the Nios II IDE. This chapter contains the following sections:

- "Nios II Versions" on page 6–1
- "Architecture Revisions" on page 6–2
- "Core Revisions" on page 6–3
- "JTAG Debug Module Revisions" on page 6–6

Improvements to the Nios II processor might affect:

- *Features of the Nios II architecture* – An example of an architecture revision is adding instructions to support floating-point arithmetic.

- *Implementation of a specific Nios II core* – An example of a core revision is increasing the maximum possible size of the data cache memory for the Nios II/f core.

- *Features of the JTAG debug module* – An example of a JTAG debug module revision is adding an additional trigger input to the JTAG debug module, allowing it to halt processor execution on a new type of trigger event.

Altera implements Nios II revisions such that code written for an existing Nios II core also works on future revisions of the same core.

Nios II Versions

The number for any version of the Nios II processor is determined by the version of the Nios II EDS. For example, in the Nios II EDS version 6.0, all Nios II cores are also version 6.0.

Table 6–1 lists the version numbers of all releases of the Nios II processor.

Table 6–1. Nios II Processor Revision History

Version	Release Date	Notes
7.1	May 2007	No changes.
7.0	March 2007	No changes.
6.1	November 2006	No changes.
6.0	May 2006	The name Nios II Development Kit describing the software development tools changed to Nios II Embedded Design Suite.
5.1 SP1	January 2006	Bug fix for Nios II/f core.
5.1	October 2005	No changes.
5.0	May 2005	• Changed version nomenclature. Altera® now aligns the Nios II processor version with Altera's Quartus II® software version. • Memory structure enhancements: (1) Added tightly-coupled memory. (2) Made data cache line size configurable. (3) Made cache optional in Nios II/f and Nios II/s cores. • Support for HardCopy® devices.
1.1	December 2004	• Minor enhancements to the architecture: Added `cpuid` control register, and updated the `break` instruction. • Increased user control of multiply and shift hardware in the arithmetic logic unit (ALU) for Nios II/s and Nios II/f cores. • Minor bug fixes.
1.01	September 2004	• Minor bug fixes.
1.0	May2004	Initial release of the Nios processor.

Architecture Revisions

Architecture revisions augment the fundamental capabilities of the Nios II architecture, and affect all Nios II cores. A change in the architecture mandates a revision to all Nios II cores to accommodate the new architectural enhancement. For example, when Altera adds a new

instruction to the instruction set, Altera consequently must update all Nios II cores to recognize the new instruction. Table 6–2 lists revisions to the Nios II architecture.

Table 6–2. Nios II Architecture Revisions

Version	Release Date	Notes
7.0	March 2007	No changes.
6.1	November 2006	No changes.
6.0	May 2006	Added optional `cpu_resetrequest` and and `cpu_resettaken` signals to all processor cores.
5.1	October 2005	No changes.
5.0	May 2005	Added the `flushda` instruction.
1.1	December 2004	● Added `cpuid` control register. ● Updated `break` instruction specification to accept an immediate argument for use by debugging tools.
1.01	September 2004	No changes.
1.0	May 2004	Initial release of the Nios II processor architecture.

Core Revisions

Core revisions introduce changes to an existing Nios II core. Core revisions most commonly fix identified bugs, or add support for an architecture revision. Not every Nios II core is revised with every release of the Nios II architecture.

Nios II/f Core

Table 6–3 lists revisions to the Nios II/f core.

Table 6–3. Nios II/f Core Revisions

Version	Release Date	Notes
7.1	May 2007	No changes.
7.0	March 2007	No changes.
6.1	November 2006	No changes.
6.0	May 2006	Cycle count for `flushi` and `initi` instructions changes from 1 to 4 cycles. (SPR 201456)
5.1 SP1	January 2006	Bug Fix: Back-to-back store instructions can cause memory corruption to the stored data. If the first store is not to the last word of a cache line and the second store is to the last word of the line, memory corruption occurs. (SPR 201895)

Table 6–3. Nios II/f Core Revisions

Version	Release Date	Notes
5.1	October 2005	No changes.
5.0	May 2005	● Added optional tightly-coupled memory ports. Designers can add zero to four tightly-coupled instruction master ports, and zero to four tightly-coupled data master ports. ● Made the data cache line size configurable. Designers can configure the data cache with the following line sizes: 4, 16, or 32 bytes. Previously, the data cache line size was fixed at 4 bytes. ● Made instruction and data caches optional (previously, cache memories were always present). If the instruction cache is not present, the Nios II core does not have an instruction master port, and must use a tightly-coupled instruction memory. ● Full support for HardCopy devices (previous versions required a work around to support HardCopy devices).
1.1	December 2004	● Added user-configurable options affecting multiply and shift operations. Now designers can choose one of three options: (1) Use embedded multiplier resources available in the target device family (previously available). (2) Use logic elements to implement multiply and shift hardware (new option). (3) Omit multiply hardware. Shift operations take one cycle per bit shifted; multiply operations are emulated in software (new option). ● Added `cpuid` control register. ● Bug Fix: Interrupts that were disabled by `wrctl ienable` remained enabled for one clock cycle following the `wrctl` instruction. Now the instruction following such a `wrctl` cannot be interrupted. (SPR 164828)
1.01	September 2004	● Bug Fixes: (1) When a store to memory is followed immediately in the pipeline by a load from the same memory location, and the memory location is held in d-cache, the load may return invalid data.This situation can occur in C code compiled with optimization off (-O0). (SPR 158904) (2) The SOPC Builder top-level system module included an extra, unnecessary output port for systems with very small address spaces. (SPR 155871)
1.0	May 2004	Initial release of the Nios II/f core.

Nios II/s Core

Table 6–4 lists revisions to the Nios II/s core.

Table 6–4. Nios II/s Core Revisions

Version	Release Date	Notes
7.1	May 2007	No changes.
7.0	March 2007	No changes.
6.1	November 2006	No changes.
6.0	May 2006	• Cycle count for `flushi` and `initi` instructions changes from 1 to 4 cycles. (SPR 201456)
5.1	October 2005	No changes.
5.0	May 2005	• Added optional tightly-coupled memory ports. Designers can add zero to four tightly-coupled instruction master ports. • Made instruction cache optional (previously instruction cache was always present). If the instruction cache is not present, the Nios II core does not have an instruction master port, and must use a tightly-coupled instruction memory. • Full support for HardCopy devices (previous versions required a work around to support HardCopy devices).
1.1	December 2004	• Added user-configurable options affecting multiply and shift operations. Now designers can choose one of three options: (1) Use embedded multiplier resources available in the target device family (previously available). (2) Use logic elements to implement multiply and shift hardware (new option). (3) Omit multiply hardware. Shift operations take one cycle per bit shifted; multiply operations are emulated in software (new option). • Added user-configurable option to include divide hardware in the ALU. Previously this option was available for only the Nios II/f core. • Added `cpuid` control register.
1.01	September 2004	• Bug Fix: The SOPC Builder top-level system module included an extra, unnecessary output port for systems with very small address spaces. (SPR 155871)
1.0	May 2004	Initial release of the Nios II/s core.

Nios II/e Core

Table 6–5 lists revisions to the Nios II/e core.

Table 6–5. Nios II/e Core Revisions

Version	Release Date	Notes
7.1	May 2007	No changes.
7.0	March 2007	No changes.
6.1	November 2006	No changes.
6.0	May 2006	No changes.
5.1	October 2005	No changes.
5.0	May 2005	● Full support for HardCopy devices (previous versions required a work around to support HardCopy devices).
1.1	December 2004	Added `cpuid` control register.
1.01	September 2004	● Bug Fix: The SOPC Builder top-level system module included an extra, unnecessary output port for systems with very small address spaces. (SPR 155871)
1.0	May 2004	Initial release of the Nios II/e core.

JTAG Debug Module Revisions

JTAG debug module revisions augment the debug capabilities of the Nios II processor, or fix bugs isolated within the JTAG debug module logic.

Table 6–6 lists revisions to the JTAG debug module.

Table 6–6. JTAG Debug Module Revisions

Version	Release Date	Notes
7.1	May 2007	No changes.
7.0	March 2007	No changes.
6.1	November 2006	No changes.
6.0	May 2006	No changes.
5.1	October 2005	No changes.
5.0	May 2005	Full support for HardCopy devices (previous versions of the JTAG debug module did not support HardCopy devices).
1.1	December 2004	Bug fix: When using the Nios II/s and Nios II/f cores, hardware breakpoints may have falsely triggered when placed on the instruction sequentially following a `jmp`, `trap`, or any branch instruction. (SPR 158805)
1.01	September 2004	● Feature enhancements: (1) Added the ability to trigger based on the instruction address. Uses include triggering trace control (trace on/off), sequential triggers (see below), and trigger in/out signal generation. (2) Enhanced trace collection such that collection can be stopped when the trace buffer is full without halting the Nios II processor. (3) Armed triggers – Enhanced trigger logic to support two levels of triggers, or "armed triggers"; enabling the use of "Event A then event B" trigger definitions. ● Bug fixes: (1) On the Nios II/s core, trace data sometimes recorded incorrect addresses during interrupt processing. (SPR 158033) (2) Under certain circumstances, captured trace data appeared to start earlier or later than the desired trigger location. (SPR 154467) (3) During debug, the processor would hang if a hardware breakpoint and an interrupt occurred simultaneously. (SPR 154097)
1.0	May 2004	Initial release of the JTAG debug module.

Referenced Documents

This chapter references no other documents.

Document Revision History

Table 6–7 shows the revision history for this document.

Table 6–7. Document Revision History

Date & Document Version	Changes Made	Summary of Changes
May 2007 v7.1.0	• Updated tables to reflect no changes to cores. • Added table of contents to Introduction section. • Added Referenced Documents section.	
March 2007 v7.0.0	Updated tables to reflect no changes to cores.	
November 2006 v6.1.0	Updated tables to reflect no changes to cores.	
May 2006 v6.0.0	Updates for Nios II cores version 6.0.	
October 2005 v5.1.0	Updates for Nios II cores version 5.1.	
May 2005 v5.0.0	Updates for Nios II cores version 5.0.	
December 2004 v1.1	Updates for Nios II cores version 1.1.	
September 2004 v1.0	Initial release.	

7. Application Binary Interface

NII51016-7.1.0

This section describes the Application Binary Interface (ABI) for the Nios® II processor. The ABI describes:

- How data is arranged in memory
- Behavior and structure of the stack
- Function calling conventions

This chapter contains the following sections:

- "Data Types" on page 7–1
- "Memory Alignment" on page 7–2
- "Register Usage" on page 7–2
- "Stacks" on page 7–3
- "Arguments and Return Values" on page 7–8

Data Types

Table 7–1 shows the size and representation of the C/C++ data types for the Nios II processor.

Table 7–1. Representation of Data Types

Type	Size (Bytes)	Representation
char, signed char	1	2s complement (ASCII)
unsigned char	1	binary (ASCII)
short, signed short	2	2s complement
unsigned short	2	binary
int, signed int	4	2s complement
unsigned int	4	binary
long, signed long	4	2s complement
unsigned long	4	binary
float	4	IEEE
double	8	IEEE
pointer	4	binary
long long	8	2s complement
unsigned long long	8	binary

Memory Alignment

Contents in memory are aligned as follows:

- A function must be aligned to a minimum of 32-bit boundary.
- The minimum alignment of a data element is its natural size. A data element larger than 32-bits need only be aligned to a 32-bit boundary.
- Structures, unions, and strings must be aligned to a minimum of 32 bits.
- Bit-fields inside structures are always 32-bit aligned.

Register Usage

The ABI adds additional usage conventions to the Nios II register file defined in the *Programming Model* chapter of the *Nios II Processor Reference Handbook*. The ABI uses the registers as shown in Table 7–2.

Table 7–2. Nios II ABI Register Usage (Part 1 of 2)

Register	Name	Used by Compiler	Callee Saved (1)	Normal Usage
`r0`	`zero`	✓		0x00000000
`r1`	`at`			Assembler Temporary
`r2`		✓		Return Value (Least-significant 32 bits)
`r3`		✓		Return Value (Most-significant 32 bits)
`r4`		✓		Register Arguments (First 32 bits)
`r5`		✓		Register Arguments (Second 32 bits)
`r6`		✓		Register Arguments (Third 32 bits)
`r7`		✓		Register Arguments (Fourth 32 bits)
`r8`		✓		Caller-Saved General-Purpose Registers
`r9`		✓		
`r10`		✓		
`r11`		✓		
`r12`		✓		
`r13`		✓		
`r14`		✓		
`r15`		✓		

Table 7–2. Nios II ABI Register Usage (Part 2 of 2)

Register	Name	Used by Compiler	Callee Saved (1)	Normal Usage
r16		✓	✓	Callee-Saved General-Purpose Registers
r17		✓	✓	
r18		✓	✓	
r19		✓	✓	
r20		✓	✓	
r21		✓	✓	
r22		✓	✓	
r23		✓	✓	
r24	et			Exception Temporary
r25	bt			Break Temporary
r26	gp	✓		Global Pointer
r27	sp	✓		Stack Pointer
r28	fp	✓		Frame Pointer (2)
r29	ea			Exception Return Address
r30	ba			Break Return Address
r31	ra	✓		Return Address

Notes to Table 7–2:
(1) A function may use one of these registers if it saves it first. The function must restore the register's original value before exiting.
(2) If the frame pointer is not used, the register is available as a temporary. See "Frame Pointer Elimination" on page 7–4.

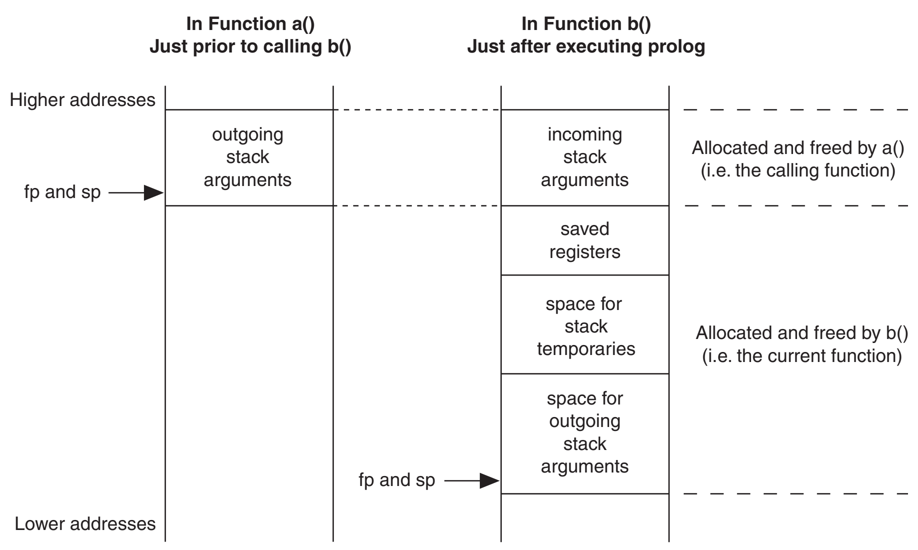
The endianess of values greater than 8-bits is little endian. The upper 8 bits of a value are stored at the higher byte address.

Stacks

The stack grows downward (i.e. towards lower addresses). The Stack Pointer points to the last used slot. The frame grows upwards, which means that the Frame Pointer points to the bottom of the frame.

Figure 7–1 shows an example of the structure of a current frame. In this case, function `a()` calls function `b()`, and the stack is shown before the call and after the prolog in the called function has completed.

Figure 7–1. Stack Pointer, Frame Pointer and the Current Frame



Each section of the current frame is aligned to a 32-bit boundary. The ABI requires the stack pointer be 32-bit aligned at all times.

Frame Pointer Elimination

Because, in the normal case, the frame pointer is the same as the stack pointer, the information in the frame pointer is redundant. Therefore, to achieve most optimal code, eliminating the frame pointer is desirable. However, when the frame pointer is eliminated, because debuggers have issues locating the stack properly, debugging without a frame pointer is difficult to do. When the frame pointer is eliminated, register `fp` becomes available as a temporary.

Call Saved Registers

Implementation note: the compiler is responsible for saving registers that need to be saved in a function. If there are any such registers, they are saved on the stack in this order from high addresses: `ra`, `fp`, `r2`, `r3`, `r4`, `r5`, `r6`, `r7`, `r8`, `r9`, `r10`, `r11`, `r12`, `r13`, `r14`, `r15`, `r16`, `r17`, `r18`, `r19`, `r20`, `r21`, `r22`, `r23`, `r24`, `r25`, `gp`, and `sp`. Stack space is not allocated for registers that are not saved.

Further Examples of Stacks

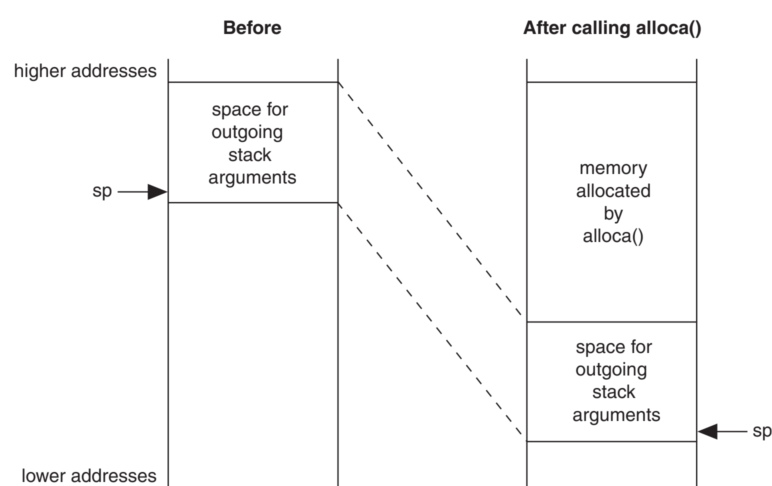
There are a number of special cases for stack layout, which are described in this section.

Stack Frame for a Function With alloca()

Figure 7–2 depicts what the frame looks like after `alloca()` is called. The space allocated by alloca() replaces the outgoing arguments and the outgoing arguments get new space allocated at the bottom of the frame.

Implementation note: the Nios II C/C++ compiler maintains a frame pointer for any function that calls `alloca()`, even if –fomit-frame-pointer is specifed.

Figure 7–2. Stack Frame after Calling alloca()

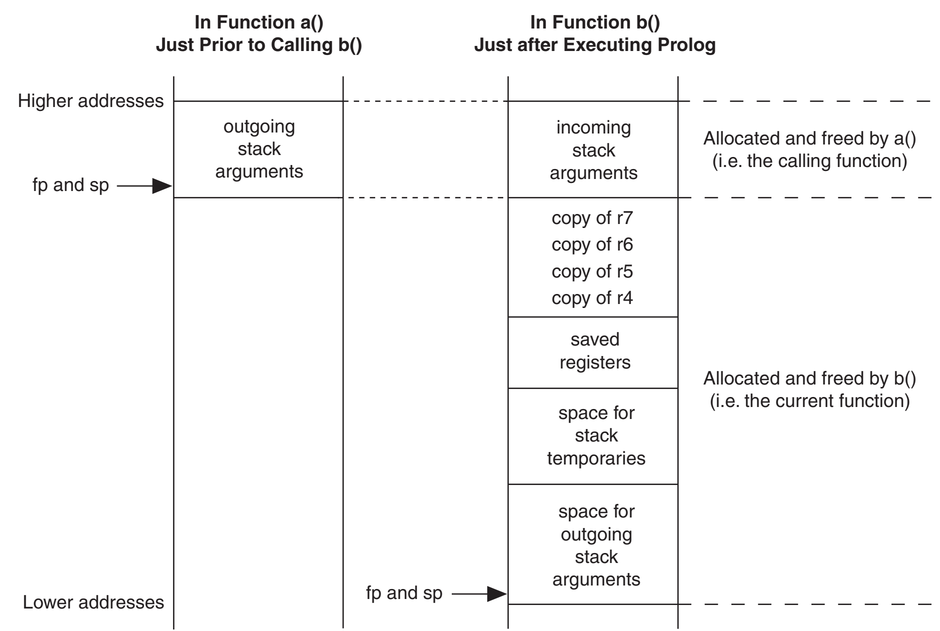


Stack Frame for a Function with Variable Arguments

Functions that take variable arguments still have their first 16-bytes of arguments arriving in registers `r4` through `r7`, just like other functions.

Implementation note: In order for varargs to work, functions that take variable arguments will allocate 16 extra bytes of storage on the stack. They will copy to the stack the first 16-bytes of their arguments from registers `r4` through `r7` as shown in Figure 7–3.

Figure 7–3. Stack Frame Using Variable Arguments



Stack Frame for a Function with Structures Passed By Value

Functions that take struct value arguments still have their first 16-bytes of arguments arriving in registers `r4` through `r7`, just like other functions.

Implementation note: if part of a structure is passed via registers, the function may need to copy the register contents back to the stack. This is similar to the variable arguments case as shown in Figure 7–3.

Function Prologs

The Nios II C/C++ compiler generates function prologs that allocate the stack frame of a function for storage of stack temporaries and outgoing arguments. In addition, each prolog is responsible for saving any state of its calling function for variables marked callee-saved by the ABI. The callee-saved register are listed in Table 7–2 on page 7–2. A function prolog is required to save a callee saved register only if the function will be using the register.

Debuggers can use the knowledge of how the function prologs work to disassemble the instructions to reconstruct state when doing a back trace. Preferably, debuggers can use information stored in the DWARF2 debugging information to find out what a prolog has done.

The instructions found in a Nios II function prolog perform the following tasks:

- Adjust the SP (to allocate the frame)
- Store registers to the frame.
- Assign the SP to the FP

Example 7–1 shows an example of a function prolog.

Example 7–1. A function prolog

```
/* Adjust the stack pointer */
addisp, sp, -120/* make a 120 byte frame */

/* Store registers to the frame */
stw ra, 116(sp)/* store the return address */
stw fp, 112(sp)/* store the frame pointer*/
stw r16, 108(sp)/* store callee-saved register */
stw r17, 104(sp) /* store callee-saved register */

/* Set the new frame pointer */
mov fp, sp
```

Prolog Variations

The following variations can occur in a prolog:

- If the function's frame size is greater than 32,767 bytes, extra temporary registers will be used in the calculation of the new SP as well as for the offsets of where to store callee-saved registers. This is due to the maximum size of immediate values allowed by the Nios II processor.
- If the frame pointer is not in use, the move of the SP to FP will not happen.
- If variable arguments are used, there will be extra instructions to store the argument registers to the stack.
- If the function is a leaf function, the return address will not be saved.
- If optimizations are on, especially instruction scheduling, the order of the instructions may change and may become interlaced with instructions located after the prolog.

Arguments and Return Values

This section discusses the details of passing arguments to functions and returning values from functions.

Arguments

The first 16-bytes to a function are passed in registers `r4` through `r7`. The arguments are passed as if a structure containing the types of the arguments was constructed, and the first 16-bytes of the structure are located in `r4` through `r7`.

A simple example:

```
int function (int a, int b);
```

The equivalent structure representing the arguments is:

```
struct { int a; int b; };
```

The first 16-bytes of the struct are assigned to `r4` through `r7`. Therefore `r4` is assigned the value of *a* and `r5` the value of *b*.

The first 16-bytes to a function taking variable arguments are passed the same way as a function not taking variable arguments. The called function must clean-up the stack as necessary to support the variable arguments. See "Stack Frame for a Function with Variable Arguments" on page 7–5.

Return Values

Return values of types up to 8-bytes are returned in `r2` and `r3`. For return values greater than 8-bytes, the caller must allocate memory for the result and must pass the address of the result memory as a hidden zero argument.

The hidden zero argument is best explained through an example.

Example 7–2. Example: function* `a()` *calls function* `b()`*, which returns a struct.

```
/* b() computes a structure-type result and returns it
*/
STRUCT b(int i, int j)
{
    ...
    return result;
}

void a(...)
{
    ...
    value = b(i, j);
}
```

In this example, as long as the result type is no larger than 8 bytes, `b()` will return its result in `r2` and `r3`.

If the return type is larger than 8 bytes, the Nios II C/C++ compiler treats this program as if `a()` had passed a pointer to `b()`. The example below shows how the Nios II C/C++ compiler sees the code above.

Example 7–3. void b(STRUCT *p_result, int i, int j)

```
{
    ...
    *p_result = result;
}

void a(...)
{
    STRUCT value;
    ...
    b(*value, i, j);
}
```

Referenced Documents

This chapter references no other documents.

Document Revision History

Table 7–3 shows the revision history for this document.

Table 7–3. Document Revision History

Date & Document Version	Changes Made	Summary of Changes
May 2007 v7.1.0	• Added table of contents to Introduction section. • Added Referenced Documents section.	
March 2007 v7.0.0	No change from previous release.	
November 2006 v6.1.0	No change from previous release.	
May 2006 v6.0.0	No change from previous release.	
October 2005 v5.1.0	No change from previous release.	
May 2005 v5.0.0	No change from previous release.	
May 2004 v1.0	Initial release.	

8. Instruction Set Reference

NII51017-7.1.0

Introduction

This section introduces the Nios® II instruction-word format and provides a detailed reference of the Nios II instruction set. This chapter contains the following sections:

- "Word Formats" on page 8–1
- "Instruction Opcodes" on page 8–4
- "Assembler Pseudo-instructions" on page 8–6
- "Assembler Macros" on page 8–7
- "Instruction Set Reference" on page 8–8

Word Formats

There are three types of Nios II instruction word format: I-type, R-type, and J-type.

I-Type

The defining characteristic of the I-type instruction-word format is that it contains an immediate value embedded within the instruction word. I-type instructions words contain:

- A 6-bit opcode field OP
- Two 5-bit register fields A and B
- A 16 bit immediate data field IMM16

In most cases, fields A and IMM16 specify the source operands, and field B specifies the destination register. IMM16 is considered signed except for logical operations and unsigned comparisons.

I-type instructions include arithmetic and logical operations such as `addi` and `andi`; branch operations; load and store operations; and cache-management operations.

The I-type instruction format is:

31–27	26–22	21–6	5–0
A	B	IMM16	OP

R-Type

The defining characteristic of the R-type instruction-word format is that all arguments and results are specified as registers. R-type instructions contain:

- A 6-bit opcode field OP
- Three 5-bit register fields A, B, and C
- An 11-bit opcode-extension field OPX

In most cases, fields A and B specify the source operands, and field C specifies the destination register. Some R-Type instructions embed a small immediate value in the low-order bits of OPX.

R-type instructions include arithmetic and logical operations such as `add` and `nor`; comparison operations such as `cmpeq` and `cmplt`; the `custom` instruction; and other operations that need only register operands.

The R-type instruction format is:

31–27	26–22	21–17	16–6	5–0
A	B	C	OPX	OP

J-Type

J-type instructions contain:

- A 6-bit opcode field
- A 26-bit immediate data field

The only J-type instruction is `call`.

The J-type instruction format is:

31–6	5–0
IMMED26	OP

Instruction Opcodes

The OP field in the Nios II instruction word specifies the major class of an opcode as shown in Table 8–1 and Table 8–2. Most values of OP are encodings for I-type instructions. One encoding, OP = 0x00, is the J-type instruction `call`. Another encoding, OP = 0x3a, is used for all R-type instructions, in which case, the OPX field differentiates the instructions. All unused encodings of OP and OPX are reserved.

Table 8–1. OP Encodings

OP	Instruction	OP	Instruction	OP	Instruction	OP	Instruction
0x00	call	0x10	cmplti	0x20	cmpeqi	0x30	cmpltui
0x01		0x11		0x21		0x31	
0x02		0x12		0x22		0x32	custom
0x03	ldbu	0x13		0x23	ldbuio	0x33	initd
0x04	addi	0x14	ori	0x24	muli	0x34	orhi
0x05	stb	0x15	stw	0x25	stbio	0x35	stwio
0x06	br	0x16	blt	0x26	beq	0x36	bltu
0x07	ldb	0x17	ldw	0x27	ldbio	0x37	ldwio
0x08	cmpgei	0x18	cmpnei	0x28	cmpgeui	0x38	
0x09		0x19		0x29		0x39	
0x0A		0x1A		0x2A		0x3A	R-Type
0x0B	ldhu	0x1B	flushda	0x2B	ldhuio	0x3B	flushd
0x0C	andi	0x1C	xori	0x2C	andhi	0x3C	xorhi
0x0D	sth	0x1D		0x2D	sthio	0x3D	
0x0E	bge	0x1E	bne	0x2E	bgeu	0x3E	
0x0F	ldh	0x1F		0x2F	ldhio	0x3F	

Table 8–2. OPX Encodings for R-Type Instructions

OPX	Instruction	OPX	Instruction	OPX	Instruction	OPX	Instruction
0x00		0x10	cmplt	0x20	cmpeq	0x30	cmpltu
0x01	eret	0x11		0x21		0x31	add
0x02	roli	0x12	slli	0x22		0x32	
0x03	rol	0x13	sll	0x23		0x33	
0x04	flushp	0x14		0x24	divu	0x34	break
0x05	ret	0x15		0x25	div	0x35	
0x06	nor	0x16	or	0x26	rdctl	0x36	sync
0x07	mulxuu	0x17	mulxsu	0x27	mul	0x37	
0x08	cmpge	0x18	cmpne	0x28	cmpgeu	0x38	
0x09	bret	0x19		0x29	initi	0x39	sub
0x0A		0x1A	srli	0x2A		0x3A	srai
0x0B	ror	0x1B	srl	0x2B		0x3B	sra
0x0C	flushi	0x1C	nextpc	0x2C		0x3C	
0x0D	jmp	0x1D	callr	0x2D	trap	0x3D	
0x0E	and	0x1E	xor	0x2E	wrctl	0x3E	
0x0F		0x1F	mulxss	0x2F		0x3F	

Assembler Pseudo-instructions

Table 8–3 lists pseudoinstructions available in Nios II assembly language. Pseudoinstructions are used in assembly source code like regular assembly instructions. Each pseudoinstruction is implemented at the machine level using an equivalent instruction. The `movia` pseudoinstruction is the only exception, being implemented with two instructions. Most pseudoinstructions do not appear in disassembly views of machine code.

Table 8–3. Assembler Pseudoinstructions

Pseudoinstruction	Equivalent Instruction
`bgt rA, rB, label`	`blt rB, rA, label`
`bgtu rA, rB, label`	`bltu rB, rA, label`
`ble rA, rB, label`	`bge rB, rA, label`
`bleu rA, rB, label`	`bgeu rB, rA, label`
`cmpgt rC, rA, rB`	`cmplt rC, rB, rA`
`cmpgti rB, rA, IMMED`	`cmpgei rB, rA, (IMMED+1)`
`cmpgtu rC, rA, rB`	`cmpltu rC, rB, rA`
`cmpgtui rB, rA, IMMED`	`cmpgeui rB, rA, (IMMED+1)`
`cmple rC, rA, rB`	`cmpge rC, rB, rA`
`cmplei rB, rA, IMMED`	`cmplti rB, rA, (IMMED+1)`
`cmpleu rC, rA, rB`	`cmpgeu rC, rB, rA`
`cmpleui rB, rA, IMMED`	`cmpltui rB, rA, (IMMED+1)`
`mov rC, rA`	`add rC, rA, r0`
`movhi rB, IMMED`	`orhi rB, r0, IMMED`
`movi rB, IMMED`	`addi, rB, r0, IMMED`
`movia rB, label`	`orhi rB, r0, %hiadj(label)` `addi, rB, r0, %lo(label)`
`movui rB, IMMED`	`ori rB, r0, IMMED`
`nop`	`add r0, r0, r0`
`subi, rB, rA, IMMED`	`addi rB, rA, IMMED`

Assembler Macros

The Nios II assembler provides macros to extract halfwords from labels and from 32-bit immediate values. Table 8–4 lists the available macros. These macros return 16-bit signed values or 16-bit unsigned values depending on where they are used. When used with an instruction that requires a 16-bit signed immediate value, these macros return a value ranging from –32768 to 32767. When used with an instruction that requires a 16-bit unsigned immediate value, these macros return a value ranging from 0 to 65535.

Table 8–4. Assembler Macros

Macro	Description	Operation
`%lo(immed32)`	Extract bits [15..0] of immed32	immed32 & 0xffff
`%hi(immed32)`	Extract bits [31..16] of immed32	(immed32 >> 16) & 0xffff
`%hiadj(immed32)`	Extract bits [31..16] and adds bit 15 of immed32	((immed32 >> 16) & 0xffff) + ((immed32 >> 15) & 0x1)
`%gprel(immed32)`	Replace the immed32 address with an offset from the global pointer (1)	immed32 –_gp

Note to Table 8–4:

(1) See the *Application Binary Interface* chapter of the *Nios II Processor Reference Handbook* for more information about global pointers.

Instruction Set Reference

The following pages list all Nios II instruction mnemonics in alphabetical order. Table 8–5 shows the notation conventions used to describe instruction operation.

Table 8–5. Notation Conventions

Notation	Meaning
X ← Y	X is written with Y
PC ← X	The program counter (PC) is written with address X; the instruction at X will be the next instruction to execute
PC	The address of the assembly instruction in question
rA, rB, rC	One of the 32-bit general-purpose registers
IMM*n*	An *n*-bit immediate value, embedded in the instruction word
IMMED	An immediate value
X_n	The nth bit of X, where $n = 0$ is the LSB
$X_{n..m}$	Consecutive bits *n* through *m* of X
0xNNMM	Hexadecimal notation
X : Y	Bitwise concatenation For example, (0x12 : 0x34) = 0x1234
σ(X)	The value of X after being sign-extended into a full register-sized signed integer
X >> *n*	The value X after being right-shifted *n* bit positions
X << *n*	The value X after being left-shifted *n* bit positions
X & Y	Bitwise logical AND
X | Y	Bitwise logical OR
X ^ Y	Bitwise logical XOR
~X	Bitwise logical NOT (one's complement)
Mem8[X]	The byte located in data memory at byte-address X
Mem16[X]	The halfword located in data memory at byte-address X
Mem32[X]	The word located in data memory at byte-address X
label	An address label specified in the assembly file
(signed) rX	The value of rX treated as a signed number
(unsigned) rX	The value of rX, treated as an unsigned number

add

Operation: rC ← rA + rB

Assembler Syntax: `add rC, rA, rB`

Example: `add r6, r7, r8`

Description: Calculates the sum of rA and rB. Stores the result in rC. Used for both signed and unsigned addition.

Usage:

Carry Detection (unsigned operands):

Following an `add` operation, a carry out of the MSB can be detected by checking whether the unsigned sum is less than one of the unsigned operands. The carry bit can be written to a register, or a conditional branch can be taken based on the carry condition. Both cases are shown below.

```
add rC, rA, rB          ; The original add operation
cmpltu rD, rC, rA       ; rD is written with the carry bit

add rC, rA, rB          ; The original add operation
bltu rC, rA, label      ; Branch if carry was generated
```

Overflow Detection (signed operands):

An overflow is detected when two positives are added and the sum is negative, or when two negatives are added and the sum is positive. The overflow condition can control a conditional branch, as shown below.

```
add rC, rA, rB          ; The original add operation
xor rD, rC, rA          ; Compare signs of sum and rA
xor rE, rC, rB          ; Compare signs of sum and rB
and rD, rD, rE          ; Combine comparisons
blt rD, r0,label        ; Branch if overflow occurred
```

Instruction Type: R

Instruction Fields:
A = Register index of operand rA
B = Register index of operand rB
C = Register index of operand rC

31–27	26–22	21–17	16–11	10–6	5–0
A	B	C	0x31	0	0x3a

addi

add immediate

Operation: rB ← rA + σ (IMM16)

Assembler Syntax: `addi rB, rA, IMM16`

Example: `addi r6, r7, -100`

Description: Sign-extends the 16-bit immediate value and adds it to the value of rA. Stores the sum in rB.

Usage:

Carry Detection (unsigned operands):

Following an `addi` operation, a carry out of the MSB can be detected by checking whether the unsigned sum is less than one of the unsigned operands. The carry bit can be written to a register, or a conditional branch can be taken based on the carry condition. Both cases are shown below.

```
addi rB, rA, IMM16      ; The original add operation
cmpltu rD, rB, rA       ; rD is written with the carry bit

addi rB, rA, IMM16      ; The original add operation
bltu rB, rA, label      ; Branch if carry was generated
```

Overflow Detection (signed operands):

An overflow is detected when two positives are added and the sum is negative, or when two negatives are added and the sum is positive. The overflow condition can control a conditional branch, as shown below.

```
addi rB, rA, IMM16      ; The original add operation
xor rC, rB, rA          ; Compare signs of sum and rA
xorhi rD, rB, IMM16     ; Compare signs of sum and IMM16
and rC, rC, rD          ; Combine comparisons
blt rC, r0,label        ; Branch if overflow occurred
```

Instruction Type: I

Instruction Fields:

A = Register index of operand rA
B = Register index of operand rB
IMM16 = 16-bit signed immediate value

31–27	26–22	21–6	5–0
A	B	IMM16	0x04

and

bitwise logical and

Operation: rC ← rA & rB

Assembler Syntax: `and rC, rA, rB`

Example: `and r6, r7, r8`

Description: Calculates the bitwise logical AND of rA and rB and stores the result in rC.

Instruction Type: R

Instruction Fields:
A = Register index of operand rA
B = Register index of operand rB
C = Register index of operand rC

31–27	26–22	21–17	16–11	10–6	5–0
A	B	C	0x0e	0	0x3a

andhi

bitwise logical and immediate into high halfword

Operation: rB ← rA & (IMM16 : 0x0000)

Assembler Syntax: `andhi rB, rA, IMM16`

Example: `andhi r6, r7, 100`

Description: Calculates the bitwise logical AND of rA and (IMM16 : 0x0000) and stores the result in rB.

Instruction Type: I

Instruction Fields:
A = Register index of operand rA
B = Register index of operand rB
IMM16 = 16-bit unsigned immediate value

31–27	26–22	21–6	5–0
A	B	IMM16	0x2c

andi

bitwise logical and immediate

Operation: rB ← rA & (0x0000 : IMM16)

Assembler Syntax: `andi rB, rA, IMM16`

Example: `andi r6, r7, 100`

Description: Calculates the bitwise logical AND of rA and (0x0000 : IMM16) and stores the result in rB.

Instruction Type: I

Instruction Fields:
A = Register index of operand rA
B = Register index of operand rB
IMM16 = 16-bit unsigned immediate value

<table>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="5">A</td><td colspan="5">B</td><td colspan="16">IMM16</td><td colspan="6">0x0c</td></tr>
</table>

beq

branch if equal

Operation: if (rA == rB)
then PC ← PC + 4 + σ (IMM16)
else PC ← PC + 4

Assembler Syntax: `beq rA, rB, label`

Example: `beq r6, r7, label`

Description: If rA == rB, then `beq` transfers program control to the instruction at label. In the instruction encoding, the offset given by IMM16 is treated as a signed number of bytes relative to the instruction immediately following `beq`. The two least-significant bits of IMM16 are always zero, because instruction addresses must be word-aligned.

Instruction Type: I

Instruction Fields: A = Register index of operand rA
B = Register index of operand rB
IMM16 = 16-bit signed immediate value

31–27	26–22	21–6	5–0
A	B	IMM16	0x26

bge

branch if greater than or equal signed

Operation: if ((signed) rA >= (signed) rB)
then PC ← PC + 4 + σ (IMM16)
else PC ← PC + 4

Assembler Syntax: `bge rA, rB, label`

Example: `bge r6, r7, top_of_loop`

Description: If (signed) rA >= (signed) rB, then `bge` transfers program control to the instruction at label. In the instruction encoding, the offset given by IMM16 is treated as a signed number of bytes relative to the instruction immediately following `bge`. The two least-significant bits of IMM16 are always zero, because instruction addresses must be word-aligned.

Instruction Type: I

Instruction Fields: A = Register index of operand rA
B = Register index of operand rB
IMM16 = 16-bit signed immediate value

31–27	26–22	21–6	5–0
A	B	IMM16	0x0e

bgeu

branch if greater than or equal unsigned

Operation: if ((unsigned) rA >= (unsigned) rB)
then PC ← PC + 4 + σ (IMM16)
else PC ← PC + 4

Assembler Syntax: `bgeu rA, rB, label`

Example: `bgeu r6, r7, top_of_loop`

Description: If (unsigned) rA >= (unsigned) rB, then `bgeu` transfers program control to the instruction at label. In the instruction encoding, the offset given by IMM16 is treated as a signed number of bytes relative to the instruction immediately following `bgeu`. The two least-significant bits of IMM16 are always zero, because instruction addresses must be word-aligned.

Instruction Type: I

Instruction Fields: A = Register index of operand rA
B = Register index of operand rB
IMM16 = 16-bit signed immediate value

31–27	26–22	21–6	5–0
A	B	IMM16	0x2e

bgt
branch if greater than signed

Operation: if ((signed) rA > (signed) rB)
then PC ← label
else PC ← PC + 4

Assembler Syntax: `bgt rA, rB, label`

Example: `bgt r6, r7, top_of_loop`

Description: If (signed) rA > (signed) rB, then `bgt` transfers program control to the instruction at label.

Pseudoinstruction: `bgt` is implemented with the `blt` instruction by swapping the register operands.

bgtu

branch if greater than unsigned

Operation: if ((unsigned) rA > (unsigned) rB)
then PC ← label
else PC ← PC + 4

Assembler Syntax: `bgtu rA, rB, label`

Example: `bgtu r6, r7, top_of_loop`

Description: If (unsigned) rA > (unsigned) rB, then `bgtu` transfers program control to the instruction at label.

Pseudoinstruction: `bgtu` is implemented with the `bltu` instruction by swapping the register operands.

ble

branch if less than or equal signed

Operation: if ((signed) rA <= (signed) rB)
then PC ← label
else PC ← PC + 4

Assembler Syntax: `ble rA, rB, label`

Example: `ble r6, r7, top_of_loop`

Description: If (signed) rA <= (signed) rB, then `ble` transfers program control to the instruction at label.

Pseudoinstruction: `ble` is implemented with the `bge` instruction by swapping the register operands.

bleu

branch if less than or equal to unsigned

Operation:	if ((unsigned) rA <= (unsigned) rB) then PC ← label else PC ← PC + 4
Assembler Syntax:	`bleu rA, rB, label`
Example:	`bleu r6, r7, top_of_loop`
Description:	If (unsigned) rA <= (unsigned) rB, then `bleu` transfers program counter to the instruction at label.
Pseudoinstruction:	`bleu` is implemented with the `bgeu` instruction by swapping the register operands.

blt
branch if less than signed

Operation: if ((signed) rA < (signed) rB)
then PC ← PC + 4 + σ (IMM16)
else PC ← PC + 4

Assembler Syntax: `blt rA, rB, label`

Example: `blt r6, r7, top_of_loop`

Description: If (signed) rA < (signed) rB, then `blt` transfers program control to the instruction at label. In the instruction encoding, the offset given by IMM16 is treated as a signed number of bytes relative to the instruction immediately following `blt`. The two least-significant bits of IMM16 are always zero, because instruction addresses must be word-aligned.

Instruction Type: I

Instruction Fields: A = Register index of operand rA
B = Register index of operand rB
IMM16 = 16-bit signed immediate value

31–27	26–22	21–6	5–0
A	B	IMM16	0x16

bltu

branch if less than unsigned

Operation: if ((unsigned) rA < (unsigned) rB)
then PC ← PC + 4 + σ (IMM16)
else PC ← PC + 4

Assembler Syntax: `bltu rA, rB, label`

Example: `bltu r6, r7, top_of_loop`

Description: If (unsigned) rA < (unsigned) rB, then `bltu` transfers program control to the instruction at label. In the instruction encoding, the offset given by IMM16 is treated as a signed number of bytes relative to the instruction immediately following `bltu`. The two least-significant bits of IMM16 are always zero, because instruction addresses must be word-aligned.

Instruction Type: I

Instruction Fields: A = Register index of operand rA
B = Register index of operand rB
MM16 = 16-bit signed immediate value

31–27	26–22	21–6	5–0
A	B	IMM16	0x36

bne

branch if not equal

Operation: if (rA != rB)
then PC ← PC + 4 + σ (IMM16)
else PC ← PC + 4

Assembler Syntax: `bne rA, rB, label`

Example: `bne r6, r7, top_of_loop`

Description: If rA != rB, then `bne` transfers program control to the instruction at label. In the instruction encoding, the offset given by IMM16 is treated as a signed number of bytes relative to the instruction immediately following `bne`.The two least-significant bits of IMM16 are always zero, because instruction addresses must be word-aligned.

Instruction Type: I

Instruction Fields: A = Register index of operand rA
B = Register index of operand rB
IMM16 = 16-bit signed immediate value

31–27	26–22	21–6	5–0
A	B	IMM16	0x1e

br
unconditional branch

Operation: PC ← PC + 4 + σ (IMM16)

Assembler Syntax: `br label`

Example: `br top_of_loop`

Description: Transfers program control to the instruction at label. In the instruction encoding, the offset given by IMM16 is treated as a signed number of bytes relative to the instruction immediately following `br`. The two least-significant bits of IMM16 are always zero, because instruction addresses must be word-aligned.

Instruction Type: I

Instruction Fields: IMM16 = 16-bit signed immediate value

31–27	26–22	21–6	5–0
0	0	IMM16	0x06

break

debugging breakpoint

Operation:

bstatus ← status
PIE ← 0
U ← 0
ba ← PC + 4
PC ← break handler address

Assembler Syntax:

```
break
break imm5
```

Example:

```
break
```

Description:

Breaks program execution and transfers control to the debugger break-processing routine. Saves the address of the next instruction in register `ba` and saves the contents of the `status` register in `bstatus`. Disables interrupts, then transfers execution to the break handler.

The 5-bit immediate field `imm5` is ignored by the processor, but it can be used by the debugger.

`break` with no argument is the same as `break 0`.

Usage:

`break` is used by debuggers exclusively. Only debuggers should place `break` in a user program, operating system, or exception handler. The address of the break handler is specified at system generation time.

Some debuggers support `break` and `break 0` instructions in source code. These debuggers treat the `break` instruction as a normal breakpoint.

Instruction Type: R

Instruction Fields: IMM5 = Type of breakpoint

31–27	26–22	21–17	16–11	10–6	5–0
0	0	0x1e	0x34	IMM5	0x3a

bret
breakpoint return

Operation: `status` ← `bstatus`
PC ← `ba`

Assembler Syntax: `bret`

Example: `bret`

Description: Copies the value of `bstatus` into the `status` register, then transfers execution to the address in `ba`.

Usage: `bret` is used by debuggers exclusively and should not appear in user programs, operating systems, or exception handlers.

Instruction Type: R

Instruction Fields: None

31–27	26–22	21–17	16–11	10–6	5–0
0x1e	0	0	0x09	0	0x3a

call

call subroutine

Operation: $ra \leftarrow PC + 4$

$PC \leftarrow (PC_{31..28} : IMM26 \times 4)$

Assembler Syntax: `call label`

Example: `call write_char`

Description: Saves the address of the next instruction in register `ra`, and transfers execution to the instruction at address ($PC_{31..28}$: IMM26 × 4).

Usage: `call` can transfer execution anywhere within the 256 MB range determined by $PC_{31..28}$. The linker does not automatically handle cases in which the address is out of this range.

Instruction Type: J

Instruction Fields: IMM26 = 26-bit unsigned immediate value

31–6	5–0
IMM26	0

callr

call subroutine in register

Operation: `ra` ← PC + 4
PC ← rA

Assembler Syntax: `callr rA`

Example: `callr r6`

Description: Saves the address of the next instruction in the return-address register, and transfers execution to the address contained in register rA.

Usage: `callr` is used to dereference C-language function pointers.

Instruction Type: R

Instruction Fields: A = Register index of operand rA

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
A	0	0x1f	0x1d	0	0x3a

cmpeq

compare equal

Operation: if (rA == rB)
then rC ← 1
else rC ← 0

Assembler Syntax: `cmpeq rC, rA, rB`

Example: `cmpeq r6, r7, r8`

Description: If rA == rB, then stores 1 to rC; otherwise, stores 0 to rC.

Usage: `cmpeq` performs the == operation of the C programming language. Also, `cmpeq` can be used to implement the C logical-negation operator "!".

```
cmpeq rC, rA, r0          ; Implements rC = !rA
```

Instruction Type: R

Instruction Fields: A = Register index of operand rA
B = Register index of operand rB
C = Register index of operand rC

<table>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="5">A</td><td colspan="5">B</td><td colspan="5">C</td><td colspan="6">0x20</td><td colspan="5">0</td><td colspan="6">0x3a</td></tr>
</table>

cmpeqi

compare equal immediate

Operation: if (rA σ (IMM16))
then rB ← 1
else rB ← 0

Assembler Syntax: `cmpeqi rB, rA, IMM16`

Example: `cmpeqi r6, r7, 100`

Description: Sign-extends the 16-bit immediate value IMM16 to 32 bits and compares it to the value of rA. If rA == σ (IMM16), `cmpeqi` stores 1 to rB; otherwise stores 0 to rB.

Usage: `cmpeqi` performs the == operation of the C programming language.

Instruction Type: I

Instruction Fields: A = Register index of operand rA
B = Register index of operand rB
IMM16 = 16-bit signed immediate value

31–27	26–22	21–6	5–0
A	B	IMM16	0x20

cmpge

compare greater than or equal signed

Operation: if ((signed) rA >= (signed) rB)
then rC ← 1
else rC ← 0

Assembler Syntax: `cmpge rC, rA, rB`

Example: `cmpge r6, r7, r8`

Description: If rA >= rB, then stores 1 to rC; otherwise stores 0 to rC.

Usage: `cmpge` performs the signed >= operation of the C programming language.

Instruction Type: R

Instruction Fields: A = Register index of operand rA
B = Register index of operand rB
C = Register index of operand rC

31–27	26–22	21–17	16–11	10–6	5–0
A	B	C	0x08	0	0x3a

cmpgei

compare greater than or equal signed immediate

Operation: if ((signed) rA >= (signed) σ (IMM16))
then rB ← 1
else rB ← 0

Assembler Syntax: `cmpgei rB, rA, IMM16`

Example: `cmpgei r6, r7, 100`

Description: Sign-extends the 16-bit immediate value IMM16 to 32 bits and compares it to the value of rA. If rA >= σ(IMM16), then `cmpgei` stores 1 to rB; otherwise stores 0 to rB.

Usage: `cmpgei` performs the signed >= operation of the C programming language.

Instruction Type: R

Instruction Fields: A = Register index of operand rA
B = Register index of operand rB
IMM16 = 16-bit signed immediate value

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
A	B	IMM16	0x08

cmpgeu

compare greater than or equal unsigned

Operation: if ((unsigned) rA >= (unsigned) rB)
then rC ← 1
else rC ← 0

Assembler Syntax: `cmpgeu rC, rA, rB`

Example: `cmpgeu r6, r7, r8`

Description: If rA >= rB, then stores 1 to rC; otherwise stores 0 to rC.

Usage: `cmpgeu` performs the unsigned >= operation of the C programming language.

Instruction Type: R

Instruction Fields: A = Register index of operand rA
B = Register index of operand rB
C = Register index of operand rC

31–27	26–22	21–17	16–11	10–6	5–0
A	B	C	0x28	0	0x3a

cmpgeui

compare greater than or equal unsigned immediate

Operation: if ((unsigned) rA >= (unsigned) (0x0000 : IMM16))
then rB ← 1
else rB ← 0

Assembler Syntax: `cmpgeui rB, rA, IMM16`

Example: `cmpgeui r6, r7, 100`

Description: Zero-extends the 16-bit immediate value IMM16 to 32 bits and compares it to the value of rA. If rA >= (0x0000 : IMM16), then `cmpgeui` stores 1 to rB; otherwise stores 0 to rB.

Usage: `cmpgeui` performs the unsigned >= operation of the C programming language.

Instruction Type: I

Instruction Fields: A = Register index of operand rA
B = Register index of operand rB
IMM16 = 16-bit unsigned immediate value

31–27	26–22	21–6	5–0
A	B	IMM16	0x28

cmpgt

compare greater than signed

Operation: if ((signed) rA > (signed) rB)
then rC ← 1
else rC ← 0

Assembler Syntax: `cmpgt rC, rA, rB`

Example: `cmpgt r6, r7, r8`

Description: If rA > rB, then stores 1 to rC; otherwise stores 0 to rC.

Usage: `cmpgt` performs the signed > operation of the C programming language.

Pseudoinstruction: `cmpgt` is implemented with the `cmplt` instruction by swapping its rA and rB operands.

cmpgti

compare greater than signed immediate

Operation: if ((signed) rA > (signed) IMMED)
then rB ← 1
else rB ← 0

Assembler Syntax: `cmpgti rB, rA, IMMED`

Example: `cmpgti r6, r7, 100`

Description: Sign-extends the immediate value IMMED to 32 bits and compares it to the value of rA. If rA > σ(IMMED), then `cmpgti` stores 1 to rB; otherwise stores 0 to rB.

Usage: `cmpgti` performs the signed > operation of the C programming language. The maximum allowed value of IMMED is 32766. The minimum allowed value is –32769.

Pseudoinstruction: `cmpgti` is implemented using a `cmpgei` instruction with an immediate value IMMED + 1.

cmpgtu

compare greater than unsigned

Operation: if ((unsigned) rA > (unsigned) rB)
then rC ← 1
else rC ← 0

Assembler Syntax: `cmpgtu rC, rA, rB`

Example: `cmpgtu r6, r7, r8`

Description: If rA > rB, then stores 1 to rC; otherwise stores 0 to rC.

Usage: `cmpgtu` performs the unsigned > operation of the C programming language.

Pseudoinstruction: `cmpgtu` is implemented with the `cmpltu` instruction by swapping its rA and rB operands.

cmpgtui
compare greater than unsigned immediate

Operation:	if ((unsigned) rA > (unsigned) IMMED) then rB ← 1 else rB ← 0
Assembler Syntax:	`cmpgtui rB, rA, IMMED`
Example:	`cmpgtui r6, r7, 100`
Description:	Zero-extends the immediate value IMMED to 32 bits and compares it to the value of rA. If rA > IMMED, then `cmpgtui` stores 1 to rB; otherwise stores 0 to rB.
Usage:	`cmpgtui` performs the unsigned > operation of the C programming language. The maximum allowed value of IMMED is 65534. The minimum allowed value is 0.
Pseudoinstruction:	`cmpgtui` is implemented using a `cmpgeui` instruction with an immediate value IMMED + 1.

cmple

compare less than or equal signed

Operation: if ((signed) rA <= (signed) rB)
then rC ← 1
else rC ← 0

Assembler Syntax: `cmple rC, rA, rB`

Example: `cmple r6, r7, r8`

Description: If rA <= rB, then stores 1 to rC; otherwise stores 0 to rC.

Usage: `cmple` performs the signed <= operation of the C programming language.

Pseudoinstruction: `cmple` is implemented with the `cmpge` instruction by swapping its rA and rB operands.

cmplei

compare less than or equal signed immediate

Operation: if ((signed) rA < (signed) IMMED)
then rB ← 1
else rB ← 0

Assembler Syntax: `cmplei rB, rA, IMMED`

Example: `cmplei r6, r7, 100`

Description: Sign-extends the immediate value IMMED to 32 bits and compares it to the value of rA. If rA <= σ(IMMED), then `cmplei` stores 1 to rB; otherwise stores 0 to rB.

Usage: `cmplei` performs the signed <= operation of the C programming language. The maximum allowed value of IMMED is 32766. The minimum allowed value is –32769.

Pseudoinstruction: `cmplei` is implemented using a `cmplti` instruction with an immediate value IMMED + 1.

cmpleu

compare less than or equal unsigned

Operation: if ((unsigned) rA < (unsigned) rB)
then rC ← 1
else rC ← 0

Assembler Syntax: `cmpleu rC, rA, rB`

Example: `cmpleu r6, r7, r8`

Description: If rA <= rB, then stores 1 to rC; otherwise stores 0 to rC.

Usage: `cmpleu` performs the unsigned <= operation of the C programming language.

Pseudoinstruction: `cmpleu` is implemented with the `cmpgeu` instruction by swapping its rA and rB operands.

cmpleui

compare less than or equal unsigned immediate

Operation: if ((unsigned) rA <= (unsigned) IMMED)
then rB ← 1
else rB ← 0

Assembler Syntax: `cmpleui rB, rA, IMMED`

Example: `cmpleui r6, r7, 100`

Description: Zero-extends the immediate value IMMED to 32 bits and compares it to the value of rA. If rA <= IMMED, then `cmpleui` stores 1 to rB; otherwise stores 0 to rB.

Usage: `cmpleui` performs the unsigned <= operation of the C programming language. The maximum allowed value of IMMED is 65534. The minimum allowed value is 0.

Pseudoinstruction: `cmpleui` is implemented using a `cmpltui` instruction with an immediate value IMMED + 1.

cmplt

compare less than signed

Operation: if ((signed) rA < (signed) rB)
then rC ← 1
else rC ← 0

Assembler Syntax: `cmplt rC, rA, rB`

Example: `cmplt r6, r7, r8`

Description: If rA < rB, then stores 1 to rC; otherwise stores 0 to rC.

Usage: `cmplt` performs the signed < operation of the C programming language.

Instruction Type: R

Instruction Fields: A = Register index of operand rA
B = Register index of operand rB
C = Register index of operand rC

<table>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="5">A</td><td colspan="5">B</td><td colspan="5">C</td><td colspan="6">0x10</td><td colspan="5">0</td><td colspan="6">0x3a</td></tr>
</table>

cmplti

compare less than signed immediate

Operation: if ((signed) rA < (signed) σ (IMM16))
then rB ← 1
else rB ← 0

Assembler Syntax: `cmplti rB, rA, IMM16`

Example: `cmplti r6, r7, 100`

Description: Sign-extends the 16-bit immediate value IMM16 to 32 bits and compares it to the value of rA. If rA < σ (IMM16), then `cmplti` stores 1 to rB; otherwise stores 0 to rB.

Usage: `cmplti` performs the signed < operation of the C programming language.

Instruction Type: I

Instruction Fields: A = Register index of operand rA
B = Register index of operand rB
IMM16 = 16-bit signed immediate value

31–27	26–22	21–6	5–0
A	B	IMM16	0x10

cmpltu

compare less than unsigned

Operation: if ((unsigned) rA < (unsigned) rB)
then rC ← 1
else rC ← 0

Assembler Syntax: `cmpltu rC, rA, rB`

Example: `cmpltu r6, r7, r8`

Description: If rA < rB, then stores 1 to rC; otherwise stores 0 to rC.

Usage: `cmpltu` performs the unsigned < operation of the C programming language.

Instruction Type: R

Instruction Fields: A = Register index of operand rA
B = Register index of operand rB
C = Register index of operand rC

<table>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="5">A</td><td colspan="5">B</td><td colspan="5">C</td><td colspan="6">0x30</td><td colspan="5">0</td><td colspan="6">0x3a</td></tr>
</table>

cmpltui

compare less than unsigned immediate

Operation: if ((unsigned) rA < (unsigned) (0x0000 : IMM16))
then rB ← 1
else rB ← 0

Assembler Syntax: `cmpltui rB, rA, IMM16`

Example: `cmpltui r6, r7, 100`

Description: Zero-extends the 16-bit immediate value IMM16 to 32 bits and compares it to the value of rA. If rA < (0x0000 : IMM16), then `cmpltui` stores 1 to rB; otherwise stores 0 to rB.

Usage: `cmpltui` performs the unsigned < operation of the C programming language.

Instruction Type: I

Instruction Fields: A = Register index of operand rA
B = Register index of operand rB
IMM16 = 16-bit unsigned immediate value

31–27	26–22	21–6	5–0
A	B	IMM16	0x30

cmpne
compare not equal

Operation: if (rA != rB)
then rC ← 1
else rC ← 0

Assembler Syntax: `cmpne rC, rA, rB`

Example: `cmpne r6, r7, r8`

Description: If rA != rB, then stores 1 to rC; otherwise stores 0 to rC.

Usage: `cmpne` performs the != operation of the C programming language.

Instruction Type: R

Instruction Fields: A = Register index of operand rA
B = Register index of operand rB
C = Register index of operand rC

31–27	26–22	21–17	16–11	10–6	5–0
A	B	C	0x18	0	0x3a

cmpnei

compare not equal immediate

Operation: if (rA != σ (IMM16))
then rB ← 1
else rB ← 0

Assembler Syntax: `cmpnei rB, rA, IMM16`

Example: `cmpnei r6, r7, 100`

Description: Sign-extends the 16-bit immediate value IMM16 to 32 bits and compares it to the value of rA. If rA != σ (IMM16), then `cmpnei` stores 1 to rB; otherwise stores 0 to rB.

Usage: `cmpnei` performs the != operation of the C programming language.

Instruction Type: I

Instruction Fields: A = Register index of operand rA
B = Register index of operand rB
IMM16 = 16-bit signed immediate value

31–27	26–22	21–6	5–0
A	B	IMM16	0x18

custom

custom instruction

Operation: if c == 1
then rC ← f_M(rA, rB, A, B, C)
else Ø ← f_N(rA, rB, A, B, C)

Assembler Syntax:

```
custom N, xC, xA, xB
```

Where xA means either general purpose register rA, or custom register cA.

Example:

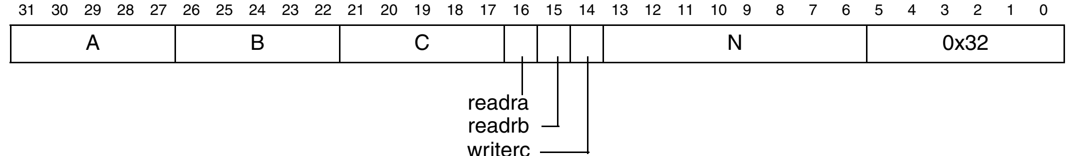
```
custom 0, c6, r7, r8
```

Description: The `custom` opcode provides access to up to 256 custom instructions allowed by the Nios II architecture. The function implemented by a custom instruction is user-defined and is specified at system generation time. The 8-bit immediate N field specifies which custom instruction to use. Custom instructions can use up to two parameters, xA and xB, and can optionally write the result to a register xC.

Usage: To access a custom register inside the custom instruction logic, clear the bit readra, readrb, or writerc that corresponds to the register field. In assembler syntax, the notation cN refers to register N in the custom register file and causes the assembler to clear the c bit of the opcode. For example, `custom 0, c3, r5, r0` performs custom instruction 0, operating on general-purpose registers r5 and r0, and stores the result in custom register 3.

Instruction Type: R

Instruction Fields:
A = Register index of operand A
B = Register index of operand B
C = Register index of operand C
N = 8-bit number that selects instruction
readra = 1 if instruction uses rA, 0 otherwise
readrb = 1 if instruction uses rB, 0 otherwise
writerc = 1 if instruction provides result for rC, 0 otherwise



div
divide

Operation: rC ← rA ÷ rB

Assembler Syntax: `div rC, rA, rB`

Example: `div r6, r7, r8`

Description: Treating rA and rB as signed integers, this instruction divides rA by rB and then stores the integer portion of the resulting quotient to rC. After attempted division by zero, the value of rC is undefined. There is no divide-by-zero exception. After dividing –2147483648 by –1, the value of rC is undefined (the number +2147483648 is not representable in 32 bits). There is no overflow exception.

Nios II processors that do not implement the `div` instruction cause an unimplemented-instruction exception.

Usage: Remainder of Division:

If the result of the division is defined, then the remainder can be computed in rD using the following instruction sequence:

```
div rC, rA, rB    ;  The original div operation
mul rD, rC, rB
sub rD, rA, rD    ;  rD = remainder
```

Instruction Type: R

Instruction Fields: A = Register index of operand rA
B = Register index of operand rB
C = Register index of operand rC

31–27	26–22	21–17	16–11	10–6	5–0
A	B	C	0x25	0	0x3a

divu

divide unsigned

Operation: rC ← rA ÷ rB

Assembler Syntax: `divu rC, rA, rB`

Example: `divu r6, r7, r8`

Description: Treating rA and rB as unsigned integers, this instruction divides rA by rB and then stores the integer portion of the resulting quotient to rC. After attempted division by zero, the value of rC is undefined. There is no divide-by-zero exception.

Nios II processors that do not implement the `divu` instruction cause an unimplemented-instruction exception.

Usage: Remainder of Division:

If the result of the division is defined, then the remainder can be computed in rD using the following instruction sequence:

```
divu rC, rA, rB   ;  The original divu operation
mul rD, rC, rB
sub rD, rA, rD    ;  rD = remainder
```

Instruction Type: R

Instruction Fields: A = Register index of operand rA
B = Register index of operand rB
C = Register index of operand rC

<table>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="5">A</td><td colspan="5">B</td><td colspan="5">C</td><td colspan="6">0x24</td><td colspan="5">0</td><td colspan="6">0x3a</td></tr>
</table>

eret

exception return

Operation: `status` ← `estatus`
PC ← `ea`

Assembler Syntax: `eret`

Example: `eret`

Description: Copies the value of `estatus` into the `status` register, and transfers execution to the address in `ea`.

Usage: Use `eret` to return from traps, external interrupts, and other exception-handling routines. Note that before returning from hardware interrupt exceptions, the exception handler must adjust the `ea` register.

Instruction Type: R

Instruction Fields: None

<table>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="5">0x1d</td><td colspan="5">0</td><td colspan="5">0</td><td colspan="6">0x01</td><td colspan="5">0</td><td colspan="6">0x3a</td></tr>
</table>

flushd

flush data cache line

Operation: Flushes the data cache line associated with address rA + σ (IMM16).

Assembler Syntax: `flushd IMM16(rA)`

Example: `flushd -100(r6)`

Description: If the Nios II processor implements a direct mapped data cache, `flushd` flushes the cache line that is mapped to the specified address, regardless whether the addressed data is currently cached. This entails the following steps:

- Computes the effective address specified by the sum of rA and the signed 16-bit immediate value
- Identifies the data cache line associated with the computed effective address. `flushd` ignores the cache line tag, which means that it flushes the cache line regardless whether the specified data location is currently cached
- If the line is dirty, writes the line back to memory
- Clears the valid bit for the line

A cache line is dirty when one or more words of the cache line have been modified by the processor, but are not yet written to memory.

If the Nios II processor core does not have a data cache, the `flushd` instruction performs no operation.

Usage: `flushd` flushes the cache line even if the addressed memory location is not in the cache. By contrast, the `flushda` instruction does nothing if the addressed memory location is not in the cache.

For more information on data cache, see the *Cache and Tightly-Coupled Memory* chapter of the *Nios II Software Developer's Handbook*.

Instruction Type: I

Instruction Fields: A = Register index of operand rA
IMM16 = 16-bit signed immediate value

31–27	26–22	21–6	5–0
A	0	IMM16	0x3b

flushda
flush data cache address

Operation: Flushes the data cache line currently cacheing address rA + σ (IMM16)

Assembler Syntax: `flushda IMM16(rA)`

Example: `flushda -100(r6)`

Description: If the addressed data is currently cached, `flushda` flushes the cache line mapped to that address. This entails the following steps:

- Computes the effective address specified by the sum of rA and the signed 16-bit immediate value
- Identifies the data cache line associated with the computed effective address.
- Compares the cache line tag with the effective address. If they do not match, the effective address is not cached, and the instruction does nothing.
- If the tag matches, and the data cache contains dirty data, writes the dirty cache line back to memory.
- Clears the valid bit for the line

A cache line is dirty when one or more words of the cache line have been modified by the processor, but are not yet written to memory.

If the Nios II processor core does not have a data cache, the `flushda` instruction performs no operation.

Usage: `flushda` flushes the cache line only if the addressed memory location is currently cached. By contrast, the `flushd` instruction flushes the cache line even if the addressed memory location is not cached.

For more information on the Nios II data cache, see the *Cache and Tightly-Coupled Memory* chapter of the *Nios II Software Developer's Handbook*.

Instruction Type: I

Instruction Fields: A = Register index of operand rA
IMM16 = 16-bit signed immediate value

31–27	26–22	21–6	5–0
A	0	IMM16	0x1b

flushi

flush instruction cache line

Operation: Flushes the instruction-cache line associated with address rA.

Assembler Syntax: `flushi rA`

Example: `flushi r6`

Description: Ignoring the tag, `flushi` identifies the instruction-cache line associated with the byte address in rA, and invalidates that line.

If the Nios II processor core does not have an instruction cache, the `flushi` instruction performs no operation.

For more information on data cache, see the *Cache and Tightly-Coupled Memory* chapter of the *Nios II Software Developer's Handbook*.

Instruction Type: R

Instruction Fields: A = Register index of operand rA

31–27	26–22	21–17	16–11	10–6	5–0
A	0	0	0x0c	0	0x3a

flushp

flush pipeline

Operation: Flushes the processor pipeline of any pre-fetched instructions.

Assembler Syntax: `flushp`

Example: `flushp`

Description: Ensures that any instructions pre-fetched after the `flushp` instruction are removed from the pipeline.

Usage: Use `flushp` before transferring control to newly updated instruction memory.

Instruction Type: R

Instruction Fields: None

<table>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="5">0</td><td colspan="5">0</td><td colspan="5">0</td><td colspan="6">0x04</td><td colspan="5">0</td><td colspan="6">0x3a</td></tr>
</table>

initd

initialize data cache line

Operation: Initializes the data cache line associated with address rA + σ (IMM16).

Assembler Syntax: `initd IMM16(rA)`

Example: `initd 0(r6)`

Description: `initd` computes the effective address specified by the sum of rA and the signed 16-bit immediate value. Ignoring the tag, `initd` indentifies the data cache line associated with the effective address, and then `initd` invalidates that line.

If the Nios II processor core does not have a data cache, the `initd` instruction performs no operation.

Usage: The instruction is used to initialize the processor's data cache. After processor reset and before accessing data memory, use `initd` to invalidate each line of the data cache.

For more information on data cache, see the *Cache and Tightly-Coupled Memory* chapter of the *Nios II Software Developer's Handbook*.

Instruction Type: I

Instruction Fields: A = Register index of operand rA

IMM16 = 16-bit signed immediate value

31–27	26–22	21–6	5–0
A	0	IMM16	0x33

initi

initialize instruction cache line

Operation: Initializes the instruction-cache line associated with address rA.

Assembler Syntax: `initi rA`

Example: `initi r6`

Description: Ignoring the tag, `initi` identifies the instruction-cache line associated with the byte address in `ra`, and `initi` invalidates that line.

If the Nios II processor core does not have an instruction cache, the `initi` instruction performs no operation.

Usage: This instruction is used to initialize the processor's instruction cache. Immediately after processor reset, use `initi` to invalidate each line of the instruction cache.

For more information on instruction cache, see the *Cache and Tightly-Coupled Memory* chapter of the *Nios II Software Developer's Handbook*.

Instruction Type: R

Instruction Fields: A = Register index of operand rA

31–27	26–22	21–17	16–11	10–6	5–0
A	0	0	0x29	0	0x3a

jmp
computed jump

Operation: PC ← rA

Assembler Syntax: `jmp rA`

Example: `jmp r12`

Description: Transfers execution to the address contained in register rA.

Usage: It is illegal to jump to the address contained in register r31. To return from subroutines called by `call` or `callr`, use `ret` instead of `jmp`.

Instruction Type: R

Instruction Fields: A = Register index of operand rA

<table>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="5">A</td><td colspan="5">0</td><td colspan="5">0</td><td colspan="6">0x0d</td><td colspan="5">0</td><td colspan="6">0x3a</td></tr>
</table>

ldb / ldbio

load byte from memory or I/O peripheral

Operation: rB ← σ (Mem8[rA + σ (IMM16)])

Assembler Syntax:

```
ldb rB, byte_offset(rA)
ldbio rB, byte_offset(rA)
```

Example:

```
ldb r6, 100(r5)
```

Description: Computes the effective byte address specified by the sum of rA and the instruction's signed 16-bit immediate value. Loads register rB with the desired memory byte, sign extending the 8-bit value to 32 bits. In Nios II processor cores with a data cache, this instruction may retrieve the desired data from the cache instead of from memory.

Usage: Use the `ldbio` instruction for peripheral I/O. In processors with a data cache, `ldbio` bypasses the cache and is guaranteed to generate an Avalon-MM data transfer. In processors without a data cache, `ldbio` acts like `ldb`.

For more information on data cache, see the *Cache and Tightly-Coupled Memory* chapter of the *Nios II Software Developer's Handbook*.

Instruction Type: I

Instruction Fields:
A = Register index of operand rA
B = Register index of operand rB
IMM16 = 16-bit signed immediate value

31–27	26–22	21–6	5–0
A	B	IMM16	0x07

Instruction format for `ldb`

31–27	26–22	21–6	5–0
A	B	IMM16	0x27

Instruction format for `ldbio`

ldbu / ldbuio

load unsigned byte from memory or I/O peripheral

Operation: rB ← 0x000000 : Mem8[rA + σ (IMM16)]

Assembler Syntax:
```
ldbu rB, byte_offset(rA)
ldbuio rB, byte_offset(rA)
```

Example:
```
ldbu r6, 100(r5)
```

Description: Computes the effective byte address specified by the sum of rA and the instruction's signed 16-bit immediate value. Loads register rB with the desired memory byte, zero extending the 8-bit value to 32 bits.

Usage: In processors with a data cache, this instruction may retrieve the desired data from the cache instead of from memory. Use the `ldbuio` instruction for peripheral I/O. In processors with a data cache, `ldbuio` bypasses the cache and is guaranteed to generate an Avalon-MM data transfer. In processors without a data cache, `ldbuio` acts like `ldbu`.

For more information on data cache, see the *Cache and Tightly-Coupled Memory* chapter of the *Nios II Software Developer's Handbook*.

Instruction Type: I

Instruction Fields:
A = Register index of operand rA
B = Register index of operand rB
IMM16 = 16-bit signed immediate value

31–27	26–22	21–6	5–0
A	B	IMM16	0x03

Instruction format for `ldbu`

31–27	26–22	21–6	5–0
A	B	IMM16	0x23

Instruction format for `ldbuio`

ldh / ldhio

load halfword from memory or I/O peripheral

Operation: rB ← σ (Mem16[rA + σ (IMM16)])

Assembler Syntax:
```
ldh rB, byte_offset(rA)
ldhio rB, byte_offset(rA)
```

Example:
```
ldh r6, 100(r5)
```

Description: Computes the effective byte address specified by the sum of rA and the instruction's signed 16-bit immediate value. Loads register rB with the memory halfword located at the effective byte address, sign extending the 16-bit value to 32 bits. The effective byte address must be halfword aligned. If the byte address is not a multiple of 2, the operation is undefined.

Usage: In processors with a data cache, this instruction may retrieve the desired data from the cache instead of from memory. Use the `ldhio` instruction for peripheral I/O. In processors with a data cache, `ldhio` bypasses the cache and is guaranteed to generate an Avalon-MM data transfer. In processors without a data cache, `ldhio` acts like `ldh`.

For more information on data cache, see the *Cache and Tightly-Coupled Memory* chapter of the *Nios II Software Developer's Handbook*.

Instruction Type: I

Instruction Fields:
A = Register index of operand rA
B = Register index of operand rB
IMM16 = 16-bit signed immediate value

31–27	26–22	21–6	5–0
A	B	IMM16	0x0f

Instruction format for `ldh`

31–27	26–22	21–6	5–0
A	B	IMM16	0x2f

Instruction format for `ldhio`

ldhu / ldhuio

load unsigned halfword from memory or I/O peripheral

Operation: rB $\leftarrow$ 0x0000 : Mem16[rA + σ (IMM16)]

Assembler Syntax:
```
ldhu rB, byte_offset(rA)
ldhuio rB, byte_offset(rA)
```

Example:
```
ldhu r6, 100(r5)
```

Description: Computes the effective byte address specified by the sum of rA and the instruction's signed 16-bit immediate value. Loads register rB with the memory halfword located at the effective byte address, zero extending the 16-bit value to 32 bits. The effective byte address must be halfword aligned. If the byte address is not a multiple of 2, the operation is undefined.

Usage: In processors with a data cache, this instruction may retrieve the desired data from the cache instead of from memory. Use the `ldhuio` instruction for peripheral I/O. In processors with a data cache, `ldhuio` bypasses the cache and is guaranteed to generate an Avalon-MM data transfer. In processors without a data cache, `ldhuio` acts like `ldhu`.

For more information on data cache, see the *Cache and Tightly-Coupled Memory* chapter of the *Nios II Software Developer's Handbook*.

Instruction Type: I

Instruction Fields:
A = Register index of operand rA
B = Register index of operand rB
IMM16 = 16-bit signed immediate value

31–27	26–22	21–6	5–0
A	B	IMM16	0x0b

Instruction format for `ldhu`

31–27	26–22	21–6	5–0
A	B	IMM16	0x2b

Instruction format for `ldhuio`

ldw / ldwio

load 32-bit word from memory or I/O peripheral

Operation: rB ← Mem32[rA + σ (IMM14)]

Assembler Syntax:
```
ldw rB, byte_offset(rA)
ldwio rB, byte_offset(rA)
```

Example:
```
ldw r6, 100(r5)
```

Description: Computes the effective byte address specified by the sum of rA and the instruction's signed 16-bit immediate value. Loads register rB with the memory word located at the effective byte address. The effective byte address must be word aligned. If the byte address is not a multiple of 4, the operation is undefined.

Usage: In processors with a data cache, this instruction may retrieve the desired data from the cache instead of from memory. Use the `ldwio` instruction for peripheral I/O. In processors with a data cache, `ldwio` bypasses the cache and memory. Use the `ldwio` instruction for peripheral I/O. In processors with a data cache, `ldwio` bypasses the cache and is guaranteed to generate an Avalon-MM data transfer. In processors without a data cache, `ldwio` acts like `ldw`.

For more information on data cache, see the *Cache and Tightly-Coupled Memory* chapter of the *Nios II Software Developer's Handbook*.

Instruction Type: I

Instruction Fields:
A = Register index of operand rA
B = Register index of operand rB
IMM16 = 16-bit signed immediate value

31–27	26–22	21–6	5–0
A	B	IMM16	0x17

Instruction format for `ldw`

31–27	26–22	21–6	5–0
A	B	IMM16	0x37

Instruction format for `ldwio`

mov
move register to register

Operation: rC ← rA

Assembler Syntax: `mov rC, rA`

Example: `mov r6, r7`

Description: Moves the contents of rA to rC.

Pseudoinstruction: `mov` is implemented as `add rC, rA, r0`.

movhi

move immediate into high halfword

Operation: rB ← (IMMED : 0x0000)

Assembler Syntax: `movhi rB, IMMED`

Example: `movhi r6, 0x8000`

Description: Writes the immediate value IMMED into the high halfword of rB, and clears the lower halfword of rB to 0x0000.

Usage: The maximum allowed value of IMMED is 65535. The minimum allowed value is 0. To load a 32-bit constant into a register, first load the upper 16 bits using a `movhi` pseudoinstruction. The %hi() macro can be used to extract the upper 16 bits of a constant or a label. Then, load the lower 16 bits with an `ori` instruction. The %lo() macro can be used to extract the lower 16 bits of a constant or label as shown below.

```
movhi rB, %hi(value)
ori rB, rB, %lo(value)
```

An alternative method to load a 32-bit constant into a register uses the %hiadj() macro and the `addi` instruction as shown below.

```
movhi rB, %hiadj(value)
addi rB, rB, %lo(value)
```

Pseudoinstruction: `movhi` is implemented as `orhi rB, r0, IMMED`.

movi

move signed immediate into word

Operation: rB ← σ (IMMED)

Assembler Syntax: `movi rB, IMMED`

Example: `movi r6, -30`

Description: Sign-extends the immediate value IMMED to 32 bits and writes it to rB.

Usage: The maximum allowed value of IMMED is 32767. The minimum allowed value is –32768. To load a 32-bit constant into a register, see the `movhi` instruction.

Pseudoinstruction: `movi` is implemented as `addi rB, r0, IMMED`.

movia

move immediate address into word

Operation: rB ← label

Assembler Syntax: `movia rB, label`

Example: `movia r6, function_address`

Description: Writes the address of label to rB.

Pseudoinstruction: `movia` is implemented as:

```
orhi rB, r0, %hiadj(label)
addi rB, rB, %lo(label)
```

movui
move unsigned immediate into word

Operation: rB ← (0x0000 : IMMED)

Assembler Syntax: `movui rB, IMMED`

Example: `movui r6, 100`

Description: Zero-extends the immediate value IMMED to 32 bits and writes it to rB.

Usage: The maximum allowed value of IMMED is 65535. The minimum allowed value is 0. To load a 32-bit constant into a register, see the `movhi` instruction.

Pseudoinstruction: `movui` is implemented as `ori rB, r0, IMMED`.

mul
multiply

Operation: rC ← $(rA \times rB)_{31..0}$

Assembler Syntax: `mul rC, rA, rB`

Example: `mul r6, r7, r8`

Description: Multiplies rA times rB and stores the 32 low-order bits of the product to rC. The result is the same whether the operands are treated as signed or unsigned integers.

Nios II processors that do not implement the `mul` instruction cause an unimplemented-instruction exception.

Usage:

Carry Detection (unsigned operands):

Before or after the multiply operation, the carry out of the MSB of rC can be detected using the following instruction sequence:

```
mul rC, rA, rB       ; The mul operation (optional)
mulxuu rD, rA, rB    ; rD is non-zero if carry occurred
cmpne rD, rD, r0     ; rD is 1 if carry occurred, 0 if not
```

The `mulxuu` instruction writes a non-zero value into rD if the multiplication of unsigned numbers will generate a carry (unsigned overflow). If a 0/1 result is desired, follow the `mulxuu` with the `cmpne` instruction.

Overflow Detection (signed operands):

After the multiply operation, overflow can be detected using the following instruction sequence:

```
mul rC, rA, rB       ; The original mul operation
cmplt rD, rC, r0
mulxss rE, rA, rB
add rD, rD, rE       ; rD is non-zero if overflow
cmpne rD, rD, r0     ; rD is 1 if overflow, 0 if not
```

The `cmplt–mulxss–add` instruction sequence writes a non-zero value into rD if the product in rC cannot be represented in 32 bits (signed overflow). If a 0/1 result is desired, follow the instruction sequence with the `cmpne` instruction.

Instruction Type: R

Instruction Fields:
A = Register index of operand rA
B = Register index of operand rB
C = Register index of operand rC

31–27	26–22	21–17	16–11	10–6	5–0
A	B	C	0x27	0	0x3a

muli

multiply immediate

Operation: $rB \leftarrow (rA \times \sigma(IMM16))_{31..0}$

Assembler Syntax: `muli rB, rA, IMM16`

Example: `muli r6, r7, -100`

Description: Sign-extends the 16-bit immediate value IMM16 to 32 bits and multiplies it by the value of rA. Stores the 32 low-order bits of the product to rB. The result is independent of whether rA is treated as a signed or unsigned number.

Nios II processors that do not implement the `muli` instruction cause an unimplemented-instruction exception.

Carry Detection and Overflow Detection:

For a discussion of carry and overflow detection, see the `mul` instruction.

Instruction Type: I

Instruction Fields:

A = Register index of operand rA
B = Register index of operand rB
IMM16 = 16-bit signed immediate value

31–27	26–22	21–6	5–0
A	B	IMM16	0x24

mulxss

multiply extended signed/signed

Operation: $rC \leftarrow ((\text{signed})\ rA) \times ((\text{signed})\ rB))_{63..32}$

Assembler Syntax: `mulxss rC, rA, rB`

Example: `mulxss r6, r7, r8`

Description: Treating rA and rB as signed integers, `mulxss` multiplies rA times rB, and stores the 32 high-order bits of the product to rC.

Nios II processors that do not implement the `mulxss` instruction cause an unimplemented-instruction exception.

Usage: Use `mulxss` and `mul` to compute the full 64-bit product of two 32-bit signed integers. Furthermore, `mulxss` can be used as part of the calculation of a 128-bit product of two 64-bit signed integers. Given two 64-bit integers, each contained in a pair of 32-bit registers, (S1 : U1) and (S2 : U2), their 128-bit product is (U1 × U2) + ((S1 × U2) << 32) + ((U1 × S2) << 32) + ((S1 × S2) << 64). The `mulxss` and `mul` instructions are used to calculate the 64-bit product S1 × S2.

Instruction Type: R

Instruction Fields:
A = Register index of operand rA
B = Register index of operand rB
C = Register index of operand rC

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

A	B	C	0x1f	0	0x3a

mulxsu

multiply extended signed/unsigned

Operation: $rC \leftarrow ((\text{signed})\ rA) \times ((\text{unsigned})\ rB))_{63..32}$

Assembler Syntax: `mulxsu rC, rA, rB`

Example: `mulxsu r6, r7, r8`

Description: Treating rA as a signed integer and rB as an unsigned integer, `mulxsu` multiplies rA times rB, and stores the 32 high-order bits of the product to rC.

Nios II processors that do not implement the `mulxsu` instruction cause an unimplemented-instruction exception.

Usage: `mulxsu` can be used as part of the calculation of a 128-bit product of two 64-bit signed integers. Given two 64-bit integers, each contained in a pair of 32-bit registers, (S1 : U1) and (S2 : U2), their 128-bit product is: (U1 × U2) + ((S1 × U2) << 32) + ((U1 × S2) << 32) + ((S1 × S2) << 64). The `mulxsu` and `mul` instructions are used to calculate the two 64-bit products S1 × U2 and U1 × S2.

Instruction Type: R

Instruction Fields:
A = Register index of operand rA
B = Register index of operand rB
C = Register index of operand rC

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

A	B	C	0x17	0	0x3a

mulxuu

multiply extended unsigned/unsigned

Operation: rC ← ((unsigned) rA) × ((unsigned) rB)) $_{63..32}$

Assembler Syntax: `mulxuu rC, rA, rB`

Example: `mulxuu r6, r7, r8`

Description: Treating rA and rB as unsigned integers, `mulxuu` multiplies rA times rB and stores the 32 high-order bits of the product to rC.

Nios II processors that do not implement the `mulxss` instruction cause an unimplemented-instruction exception.

Usage: Use `mulxuu` and `mul` to compute the 64-bit product of two 32-bit unsigned integers. Furthermore, `mulxuu` can be used as part of the calculation of a 128-bit product of two 64-bit signed integers. Given two 64-bit signed integers, each contained in a pair of 32-bit registers, (S1 : U1) and (S2 : U2), their 128-bit product is (U1 × U2) + ((S1 × U2) << 32) + ((U1 × S2) << 32) + ((S1 × S2) << 64). The `mulxuu` and `mul` instructions are used to calculate the 64-bit product U1 × U2.

`mulxuu` also can be used as part of the calculation of a 128-bit product of two 64-bit unsigned integers. Given two 64-bit unsigned integers, each contained in a pair of 32-bit registers, (T1 : U1) and (T2 : U2), their 128-bit product is (U1 × U2) + ((U1 × T2) << 32) + ((T1 × U2) << 32) + ((T1 × T2) << 64). The `mulxuu` and `mul` instructions are used to calculate the four 64-bit products U1 × U2, U1 × T2, T1 × U2, and T1 × T2.

Instruction Type: R

Instruction Fields:
A = Register index of operand rA
B = Register index of operand rB
C = Register index of operand rC

31–27	26–22	21–17	16–11	10–6	5–0
A	B	C	0x07	0	0x3a

nextpc

get address of following instruction

Operation: rC ← PC + 4

Assembler Syntax: `nextpc rC`

Example: `nextpc r6`

Description: Stores the address of the next instruction to register rC.

Usage: A relocatable code fragment can use `nextpc` to calculate the address of its data segment. `nextpc` is the only way to access the PC directly.

Instruction Type: R

Instruction Fields: C = Register index of operand rC

<table>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="5">0</td><td colspan="5">0</td><td colspan="5">C</td><td colspan="6">0x1c</td><td colspan="5">0</td><td colspan="6">0x3a</td></tr>
</table>

nop
no operation

Operation:	None
Assembler Syntax:	`nop`
Example:	`nop`
Description:	`nop` does nothing.
Pseudoinstruction:	`nop` is implemented as `add r0, r0, r0`.

nor

bitwise logical nor

Operation: rC ← ~(rA | rB)

Assembler Syntax: `nor rC, rA, rB`

Example: `nor r6, r7, r8`

Description: Calculates the bitwise logical NOR of rA and rB and stores the result in rC.

Instruction Type: R

Instruction Fields: A = Register index of operand rA
B = Register index of operand rB
C = Register index of operand rC

31–27	26–22	21–17	16–11	10–6	5–0
A	B	C	0x06	0	0x3a

or

bitwise logical or

Operation: rC ← rA | rB

Assembler Syntax: `or rC, rA, rB`

Example: `or r6, r7, r8`

Description: Calculates the bitwise logical OR of rA and rB and stores the result in rC.

Instruction Type: R

Instruction Fields:
A = Register index of operand rA
B = Register index of operand rB
C = Register index of operand rC

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

A	B	C	0x16	0	0x3a

orhi

bitwise logical or immediate into high halfword

Operation: rB ← rA | (IMM16 : 0x0000)

Assembler Syntax: `orhi rB, rA, IMM16`

Example: `orhi r6, r7, 100`

Description: Calculates the bitwise logical OR of rA and (IMM16 : 0x0000) and stores the result in rB.

Instruction Type: I

Instruction Fields: A = Register index of operand rA
B = Register index of operand rB
IMM16 = 16-bit signed immediate value

31–27	26–22	21–6	5–0
A	B	IMM16	0x34

ori

bitwise logical or immediate

Operation: rB ← rA | (0x0000 : IMM16)

Assembler Syntax: `ori rB, rA, IMM16`

Example: `ori r6, r7, 100`

Description: Calculates the bitwise logical OR of rA and (0x0000 : IMM16) and stores the result in rB.

Instruction Type: I

Instruction Fields: A = Register index of operand rA
B = Register index of operand rB
IMM16 = 16-bit unsigned immediate value

31–27	26–22	21–6	5–0
A	B	IMM16	0x14

rdctl

read from control register

Operation: rC ← ctlN

Assembler Syntax: `rdctl rC, ctlN`

Example: `rdctl r3, ctl31`

Description: Reads the value contained in control register ctlN and writes it to register rC.

Instruction Type: R

Instruction Fields: C = Register index of operand rC
N = Control register index of operand ctlN

31–27	26–22	21–17	16–11	10–6	5–0
0	0	C	0x26	N	0x3a

ret

return from subroutine

Operation: PC ← `ra`

Assembler Syntax: `ret`

Example: `ret`

Description: Transfers execution to the address in ra.

Usage: Any subroutine called by `call` or `callr` must use `ret` to return.

Instruction Type: R

Instruction Fields: None

31–27	26–22	21–17	16–11	10–6	5–0
0x1f	0	0	0x05	0	0x3a

rol
rotate left

Operation: rC ← rA rotated left $rB_{4..0}$ bit positions

Assembler Syntax: `rol rC, rA, rB`

Example: `rol r6, r7, r8`

Description: Rotates rA left by the number of bits specified in $rB_{4..0}$ and stores the result in rC. The bits that shift out of the register rotate into the least-significant bit positions. Bits 31–5 of rB are ignored.

Instruction Type: R

Instruction Fields:
A = Register index of operand rA
B = Register index of operand rB
C = Register index of operand rC

<table>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="5">A</td><td colspan="5">B</td><td colspan="5">C</td><td colspan="6">0x03</td><td colspan="5">0</td><td colspan="6">0x3a</td></tr>
</table>

roli

rotate left immediate

Operation: rC ← rA rotated left IMM5 bit positions

Assembler Syntax: `roli rC, rA, IMM5`

Example: `roli r6, r7, 3`

Description: Rotates rA left by the number of bits specified in IMM5 and stores the result in rC. The bits that shift out of the register rotate into the least-significant bit positions.

Usage: In addition to the rotate-left operation, `roli` can be used to implement a rotate-right operation. Rotating left by (32 – IMM5) bits is the equivalent of rotating right by IMM5 bits.

Instruction Type: R

Instruction Fields:
A = Register index of operand rA
C = Register index of operand rC
IMM5 = 5-bit unsigned immediate value

31–27	26–22	21–17	16–11	10–6	5–0
A	0	C	0x02	IMM5	0x3a

ror
rotate right

Operation: rC ← rA rotated right $rB_{4..0}$ bit positions

Assembler Syntax: `ror rC, rA, rB`

Example: `ror r6, r7, r8`

Description: Rotates rA right by the number of bits specified in $rB_{4..0}$ and stores the result in rC. The bits that shift out of the register rotate into the most-significant bit positions. Bits 31– 5 of rB are ignored.

Instruction Type: R

Instruction Fields: A = Register index of operand rA
B = Register index of operand rB
C = Register index of operand rC

31–27	26–22	21–17	16–11	10–6	5–0
A	B	C	0x0b	0	0x3a

sll
shift left logical

Operation: rC ← rA << ($rB_{4..0}$)

Assembler Syntax: `sll rC, rA, rB`

Example: `sll r6, r7, r8`

Description: Shifts rA left by the number of bits specified in $rB_{4..0}$ (inserting zeroes), and then stores the result in rC. `sll` performs the << operation of the C programming language.

Instruction Type: R

Instruction Fields: A = Register index of operand rA
B = Register index of operand rB
C = Register index of operand rC

31–27	26–22	21–17	16–11	10–6	5–0
A	B	C	0x13	0	0x3a

slli

shift left logical immediate

Operation: rC ← rA << IMM5

Assembler Syntax: `slli rC, rA, IMM5`

Example: `slli r6, r7, 3`

Description: Shifts rA left by the number of bits specified in IMM5 (inserting zeroes), and then stores the result in rC.

Usage: `slli` performs the << operation of the C programming language.

Instruction Type: R

Instruction Fields:
A = Register index of operand rA
C = Register index of operand rC
IMM5 = 5-bit unsigned immediate value

<table>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="5">A</td><td colspan="5">0</td><td colspan="5">C</td><td colspan="6">0x12</td><td colspan="5">IMM5</td><td colspan="6">0x3a</td></tr>
</table>

sra
shift right arithmetic

Operation: rC ← (signed) rA >> ((unsigned) $rB_{4..0}$)

Assembler Syntax: `sra rC, rA, rB`

Example: `sra r6, r7, r8`

Description: Shifts rA right by the number of bits specified in $rB_{4..0}$ (duplicating the sign bit), and then stores the result in rC. Bits 31–5 are ignored.

Usage: `sra` performs the signed >> operation of the C programming language.

Instruction Type: R

Instruction Fields: A = Register index of operand rA
B = Register index of operand rB
C = Register index of operand rC

<table>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="5">A</td><td colspan="5">B</td><td colspan="5">C</td><td colspan="6">0x3b</td><td colspan="5">0</td><td colspan="6">0x3a</td></tr>
</table>

srai

shift right arithmetic immediate

Operation: rC ← (signed) rA >> ((unsigned) IMM5)

Assembler Syntax: `srai rC, rA, IMM5`

Example: `srai r6, r7, 3`

Description: Shifts rA right by the number of bits specified in IMM5 (duplicating the sign bit), and then stores the result in rC.

Usage: `srai` performs the signed >> operation of the C programming language.

Instruction Type: R

Instruction Fields: A = Register index of operand rA
C = Register index of operand rC
IMM5 = 5-bit unsigned immediate value

<table>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="5">A</td><td colspan="5">0</td><td colspan="5">C</td><td colspan="6">0x3a</td><td colspan="5">IMM5</td><td colspan="6">0x3a</td></tr>
</table>

srl
shift right logical

Operation: rC ← (unsigned) rA >> ((unsigned) $rB_{4..0}$)

Assembler Syntax: `srl rC, rA, rB`

Example: `srl r6, r7, r8`

Description: Shifts rA right by the number of bits specified in $rB_{4..0}$ (inserting zeroes), and then stores the result in rC. Bits 31–5 are ignored.

Usage: `srl` performs the unsigned >> operation of the C programming language.

Instruction Type: R

Instruction Fields: A = Register index of operand rA
B = Register index of operand rB
C = Register index of operand rC

31–27	26–22	21–17	16–11	10–6	5–0
A	B	C	0x1b	0	0x3a

srli

shift right logical immediate

Operation: rC ← (unsigned) rA >> ((unsigned) IMM5)

Assembler Syntax: `srli rC, rA, IMM5`

Example: `srli r6, r7, 3`

Description: Shifts rA right by the number of bits specified in IMM5 (inserting zeroes), and then stores the result in rC.

Usage: `srli` performs the unsigned >> operation of the C programming language.

Instruction Type: R

Instruction Fields:
A = Register index of operand rA
C = Register index of operand rC
IMM5 = 5-bit unsigned immediate value

<table>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="5">A</td><td colspan="5">0</td><td colspan="5">C</td><td colspan="6">0x1a</td><td colspan="5">IMM5</td><td colspan="6">0x3a</td></tr>
</table>

stb / stbio

store byte to memory or I/O peripheral

Operation: Mem8[rA + σ (IMM16)] ← $rB_{7..0}$

Assembler Syntax:

```
stb rB, byte_offset(rA)
stbio rB, byte_offset(rA)
```

Example:

```
stb r6, 100(r5)
```

Description: Computes the effective byte address specified by the sum of rA and the instruction's signed 16-bit immediate value. Stores the low byte of rB to the memory byte specified by the effective address.

Usage: In processors with a data cache, this instruction may not generate an Avalon-MM bus cycle to non-cache data memory immediately. Use the `stbio` instruction for peripheral I/O. In processors with a data cache, `stbio` bypasses the cache and is guaranteed to generate an Avalon-MM data transfer. In processors without a data cache, `stbio` acts like `stb`.

Instruction Type: I

Instruction Fields:
A = Register index of operand rA
B = Register index of operand rB
IMM16 = 16-bit signed immediate value

31–27	26–22	21–6	5–0
A	B	IMM16	0x05

Instruction format for `stb`

31–27	26–22	21–6	5–0
A	B	IMM16	0x25

Instruction format for `stbio`

sth / sthio

store halfword to memory or I/O peripheral

Operation: Mem16[rA + σ (IMM16)] $\leftarrow$ $rB_{15..0}$

Assembler Syntax:
```
sth rB, byte_offset(rA)
sthio rB, byte_offset(rA)
```

Example:
```
sth r6, 100(r5)
```

Description: Computes the effective byte address specified by the sum of rA and the instruction's signed 16-bit immediate value. Stores the low halfword of rB to the memory location specified by the effective byte address. The effective byte address must be halfword aligned. If the byte address is not a multiple of 2, the operation is undefined.

Usage: In processors with a data cache, this instruction may not generate an Avalon-MM data transfer immediately. Use the sthio instruction for peripheral I/O. In processors with a data cache, `sthio` bypasses the cache and is guaranteed to generate an Avalon-MM data transfer. In processors without a data cache, `sthio` acts like `sth`.

Instruction Type: I

Instruction Fields:
A = Register index of operand rA
B = Register index of operand rB
IMM16 = 16-bit signed immediate value

31–27	26–22	21–6	5–0
A	B	IMM16	0x0d

Instruction format for `sth`

31–27	26–22	21–6	5–0
A	B	IMM16	0x2d

Instruction format for `sthio`

stw / stwio

store word to memory or I/O peripheral

Operation: Mem32[rA + σ (IMM16)] ← rB

Assembler Syntax:

```
stw rB, byte_offset(rA)
stwio rB, byte_offset(rA)
```

Example:

```
stw r6, 100(r5)
```

Description: Computes the effective byte address specified by the sum of rA and the instruction's signed 16-bit immediate value. Stores rB to the memory location specified by the effective byte address. The effective byte address must be word aligned. If the byte address is not a multiple of 4, the operation is undefined.

Usage: In processors with a data cache, this instruction may not generate an Avalon-MM data transfer immediately. Use the `stwio` instruction for peripheral I/O. In processors with a data cache, `stwio` bypasses the cache and is guaranteed to generate an Avalon-MM bus cycle. In processors without a data cache, `stwio` acts like `stw`.

Instruction Type: I

Instruction Fields:
A = Register index of operand rA
B = Register index of operand rB
IMM16 = 16-bit signed immediate value

31–27	26–22	21–6	5–0
A	B	IMM16	0x15

Instruction format for `stw`

31–27	26–22	21–6	5–0
A	B	IMM16	0x35

Instruction format for `stwio`

sub
subtract

Operation: rC ← rA – rB

Assembler Syntax: `sub rC, rA, rB`

Example: `sub r6, r7, r8`

Description: Subtract rB from rA and store the result in rC.

Usage:

Carry Detection (unsigned operands):

The carry bit indicates an unsigned overflow. Before or after a `sub` operation, a carry out of the MSB can be detected by checking whether the first operand is less than the second operand. The carry bit can be written to a register, or a conditional branch can be taken based on the carry condition. Both cases are shown below.

```
sub rC, rA, rB     ; The original sub operation (optional)
cmpltu rD, rA, rB  ; rD is written with the carry bit

sub rC, rA, rB     ; The original sub operation (optional)
bltu rA, rB, label ; Branch if carry was generated
```

Overflow Detection (signed operands):

Detect overflow of signed subtraction by comparing the sign of the difference that is written to rC with the signs of the operands. If rA and rB have different signs, and the sign of rC is different than the sign of rA, an overflow occurred. The overflow condition can control a conditional branch, as shown below.

```
sub rC, rA, rB     ; The original sub operation
xor rD, rA, rB     ; Compare signs of rA and rB
xor rE, rA, rC     ; Compare signs of rA and rC
and rD, rD, rE     ; Combine comparisons
blt rD, r0, label  ; Branch if overflow occurred
```

Instruction Type: R

Instruction Fields:
A = Register index of operand rA
B = Register index of operand rB
C = Register index of operand rC

31–27	26–22	21–17	16–11	10–6	5–0
A	B	C	0x39	0	0x3a

subi
subtract immediate

Operation:	rB ← rA – σ (IMMED)
Assembler Syntax:	`subi rB, rA, IMMED`
Example:	`subi r8, r8, 4`
Description:	Sign-extends the immediate value IMMED to 32 bits, subtracts it from the value of rA and then stores the result in rB.
Usage:	The maximum allowed value of IMMED is 32768. The minimum allowed value is –32767.
Pseudoinstruction:	`subi` is implemented as `addi rB, rA, -IMMED`

sync

memory synchronization

Operation: None

Assembler Syntax: `sync`

Example: `sync`

Description: Forces all pending memory accesses to complete before allowing execution of subsequent instructions. In processor cores that support in-order memory accesses only, this instruction performs no operation.

Instruction Type: R

Instruction Fields: None

31–27	26–22	21–17	16–11	10–6	5–0
0	0	0	0x36	0	0x3a

trap

Operation:

`estatus` ← `status`
PIE ← 0
U ← 0
`ea` ← PC + 4
PC ← exception handler address

Assembler Syntax: `trap`

Example: `trap`

Description: Saves the address of the next instruction in register `ea`, saves the contents of the `status` register in `estatus`, disables interrupts, and transfers execution to the exception handler. The address of the exception handler is specified at system generation time.

Usage: To return from the exception handler, execute an `eret` instruction.

Instruction Type: R

Instruction Fields: None

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	0	0x1d	0x2d	0	0x3a

wrctl

write to control register

Operation: ctlN ← rA

Assembler Syntax: `wrctl ctlN, rA`

Example: `wrctl ctl6, r3`

Description: Writes the value contained in register rA to the control register ctlN.

Instruction Type: R

Instruction Fields: A = Register index of operand rA
N = Control register index of operand ctlN

31–27	26–22	21–17	16–11	10–6	5–0
A	0	0	0x2e	N	0x3a

xor

bitwise logical exclusive or

Operation: rC ← rA ^ rB

Assembler Syntax: `xor rC, rA, rB`

Example: `xor r6, r7, r8`

Description: Calculates the bitwise logical exclusive XOR of rA and rB and stores the result in rC.

Instruction Type: R

Instruction Fields:
A = Register index of operand rA
B = Register index of operand rB
C = Register index of operand rC

31–27	26–22	21–17	16–11	10–6	5–0
A	B	C	0x1e	0	0x3a

xorhi

bitwise logical exclusive or immediate into high halfword

Operation: rB ← rA ^ (IMM16 : 0x0000)

Assembler Syntax: `xorhi rB, rA, IMM16`

Example: `xorhi r6, r7, 100`

Description: Calculates the bitwise logical exclusive XOR of rA and (IMM16 : 0x0000) and stores the result in rB.

Instruction Type: I

Instruction Fields: A = Register index of operand rA
B = Register index of operand rB
IMM16 = 16-bit unsigned immediate value

31–27	26–22	21–6	5–0
A	B	IMM16	0x3c

xori

bitwise logical exclusive or immediate

Operation: rB ← rA ^ (0x0000 : IMM16)

Assembler Syntax: `xori rB, rA, IMM16`

Example: `xori r6, r7, 100`

Description: Calculates the bitwise logical exclusive `or` of rA and (0x0000 : IMM16) and stores the result in rB.

Instruction Type: I

Instruction Fields:
A = Register index of operand rA
B = Register index of operand rB
IMM16 = 16-bit unsigned immediate value

31–27	26–22	21–6	5–0
A	B	IMM16	0x1c

Referenced Documents

This chapter references no other documents.

Document Revision History

Table 8–6 shows the revision history for this document.

Table 8–6. Document Revision History

Date & Document Version	Changes Made	Summary of Changes
May 2007 v7.1.0	● Added table of contents to Introduction section. ● Added Referenced Documents section.	
March 2007 v7.0.0	No change from previous release.	
November 2006 v6.1.0	No change from previous release.	
May 2006 v6.0.0	No change from previous release.	
October 2005 v5.1.0	● Correction to the `blt` instruction. ● Added U bit operation for `break` and `trap` instructions.	
July 2005 v5.0.1	● `new flushda` instruction. ● `flushd` instruction updated. ● Instruction Opcode table updated with `flushda` instruction.	
May 2005 v5.0.0	No change from previous release.	
December 2004 v1.2	● `break` instruction update. ● `srli` instruction correction.	
September 2004 v1.1	Updates for Nios II 1.01 release.	
May 2004 v1.0	Initial release.	