Nios® II

Nios II Custom Instruction User Guide

ALTERA®

101 Innovation Drive
San Jose, CA 95134
(408) 544-7000
http://www.altera.com

Copyright © 2004 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

UG-N2CSTNST-1.2

I.S. EN ISO 9001

Contents

About this User Guide

This user guide provides comprehensive information about Altera®
Nios II custom instructions.

Table 1–1 shows the user guide revision history.   December

Table 1–1. Tutorial Revision History

Date	Description
December 2004	Updates for the Nios II version 1.1 release.
September 2004	Updates for the Nios II version 1.01 release.
May 2004	First release of custom instruction user guide for the Nios II processor.

How to Find Information

- The Adobe Acrobat Find feature allows you to search the contents of a PDF file. Click the binoculars toolbar icon to open the Find dialog box
- Bookmarks serve as an additional table of contents
- Thumbnail icons, which provide miniature previews of each page, provide a link to the pages
- Numerous links, shown in green text, allow you to jump to related information

How to Contact Altera

For the most up-to-date information about Altera products, go to the Altera world-wide web site at www.altera.com. For technical support on this product, go to www.altera.com/mysupport. For additional information about Altera products, consult the sources shown below.

Information Type	USA & Canada	All Other Locations
Technical support	www.altera.com/mysupport/	altera.com/mysupport/
	(800) 800-EPLD (3753) (7:00 a.m. to 5:00 p.m. Pacific Time)	(408) 544-7000 *(1)* (7:00 a.m. to 5:00 p.m. Pacific Time)
Product literature	www.altera.com	www.altera.com
Altera literature services	lit_req@altera.com *(1)*	lit_req@altera.com *(1)*
Non-technical customer service	(800) 767-3753	(408) 544-7000 (7:30 a.m. to 5:30 p.m. Pacific Time)
FTP site	ftp.altera.com	ftp.altera.com

Note to table:

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

Typographic 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: **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	The caution indicates required information that needs special consideration and understanding and should be read prior to starting or continuing with the procedure or process.
⚠	The warning indicates information that should be read prior to starting or continuing the procedure or processes
↵	The angled arrow indicates you should press the Enter key.
👣	The feet direct you to more information on a particular topic.

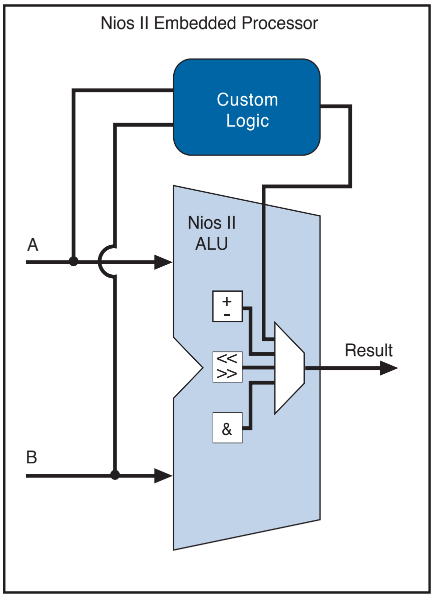
1. Nios II Custom Instruction Overview

Introduction

With the Altera® Nios® II embedded processor, system designers can accelerate time-critical software algorithms by adding custom instructions to the Nios instruction set. With custom instructions, system designers can reduce a complex sequence of standard instructions to a single instruction implemented in hardware. System designers can use this feature for a variety of applications, e.g., to optimize software inner loops for digital signal processing (DSP), packet header processing, and computation-intensive applications. The Nios II CPU configuration wizard, which is accessed via the Quartus® II software's SOPC Builder, provides a graphical user interface (GUI) used to add up to 256 custom instructions to the Nios II processor.

The custom instruction logic connects directly to the Nios II arithmetic logic unit (ALU) as shown in Figure 1–1.

Figure 1–1. Custom Instruction Logic Connects to the Nios II ALU



This chapter:

- Describes the Nios II processor custom instruction feature
- Discusses the requirements for implementing a custom instruction in hardware & software
- Defines custom instruction architectural types

For information regarding custom instructions software interface, refer to Chapter 2, Software Interface. A tutorial with design files and step-by-step instructions for implementing a custom instruction, is found in Chapter 3, Implementing a Nios II Processor Custom Instruction.

Custom Instruction Overview

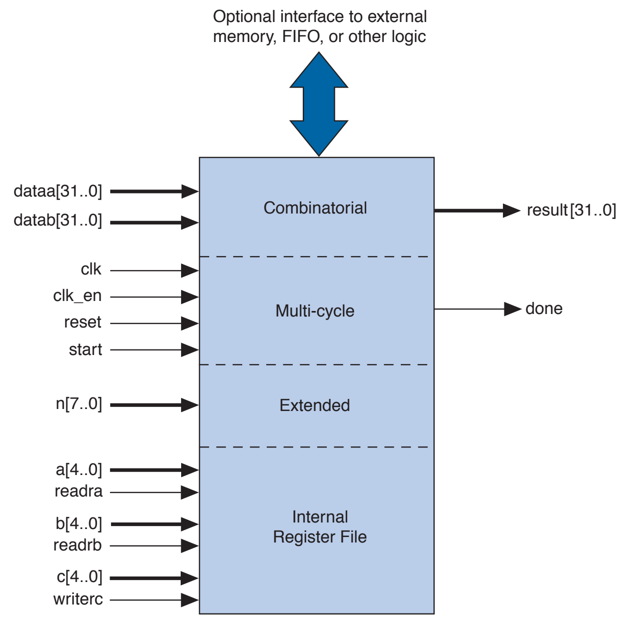
With Nios II processor custom instructions, system designers are able to take full advantage of the flexibility of FPGAs to meet system performance requirements. Custom instructions allow system designers to add custom functionality to the Nios II processor ALU.

Nios II processor custom instructions are custom logic blocks adjacent to the ALU in the CPU's data path. This gives system designers the ability to tailor the Nios II processor core to meet the needs of a particular application. System designers have the ability to accelerate time critical software algorithms by converting them to custom hardware logic blocks. Because it is easy to alter the design of the FPGA-based Nios II processor, custom instructions provide an easy way to experiment with hardware/software trade-offs during an embedded system's implementation phase—rather than the specification phase.

Implementing Custom Instruction Hardware

Figure 1–2 is a hardware block diagram of a Nios II processor custom instruction.

Figure 1–2. Hardware Block Diagram of a Nios II Processor Custom Instruction



The basic operation of Nios II custom instruction logic is to receive input on the `dataa[31..0]` and/or `datab[31..0]` and drive out the result on its `result[31..0]` port. The designer generates the custom instruction logic that produces the results.

The Nios II processor supports different architectural types of custom instructions. Figure 1–2 lists the additional signals that accommodate different architectural types. Only the ports used for the specific custom instruction implementation are required.

Figure 1–2 also shows an optional interface to external logic. The interface to external logic allows designers to include a custom interface to system resources outside of the Nios II processor data path.

Implementing Custom Instruction Software

The Nios II processor custom instruction software interface is simple and abstracts the details of the custom instruction from the programmer. For each custom instruction, the Nios II integrated development environment (IDE) produces a macro that is defined in the system header file. You can call the macro from C or C++ application code as a normal function call and you do not need to program assembly to access custom instructions. Custom instructions can also be accessed via the Nios II processor assembly code.

For more information, refer to Chapter 2, Software Interface.

Custom Instruction Architectural Types

There are different custom instruction architectures available to suit the application's requirements. The architectures range from a simple, single-cycle combinatorial architecture to an extended variable-length, multi-cycle custom instruction architecture. The chosen architecture determines what the hardware interface looks like.

Table 1–1 shows custom instruction architectural types, application, and the associated hardware interface.

Table 1–1. Custom Instruction Architectural Types, Application & Hardware Interface

Architectural Type	Application	Hardware Interface
Combinatorial	Single clock cycle custom logic blocks	`dataa[31..0], datab[31..0], result[31..0]`
Multi-cycle	Multi clock cycle custom logic block of fixed or variable durations	`dataa[31..0], datab[31..0], result[31..0], clk, clk_en, start, reset, done`
Extended	Custom logic blocks that are capable of performing multiple operations	`dataa[31..0], datab[31..0], result[31..0], clk, clk_en, start, reset, done, n[7..0]`
Internal Register File	Custom logic blocks that access internal register file for input and/or output	`dataa[31..0], datab[31..0], result[31..0], clk, clk_en, start, reset, done, n[7..0], a[4..0], readra, b[4..0], readrb, c[4..0], writerc`
External Interface	Custom logic blocks that interface to logic outside of the NIOS II processor's data path	Standard custom instruction signals, plus user-defined interface to external logic.

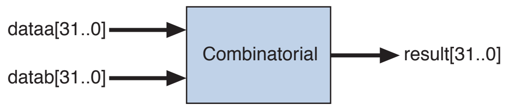
This section discusses the basic functionality and hardware interface of each custom instruction architecture type listed in Table 1–1.

Combinatorial Custom Instruction Architecture

Combinatorial custom instruction architecture consists of a logic block that is able to complete in a single clock cycle.

Figure 1–3 shows a block diagram of a combinatorial custom instruction architecture.

Figure 1–3. Combinatorial Custom Instruction Architecture



The Figure 1–3 combinatorial custom instruction diagram uses `dataa[31..0]` and `datab[31..0]` ports as inputs and drives the results on the `result[31..0]` port. Because the logic is able to complete in a single clock cycle, control signals are not needed.

Table 1–2 lists the combinatorial custom instruction signals.

Table 1–2. Combinatorial Custom Instruction Signals

Signal Name	Direction	Required	Purpose
`dataa[31..0]`	Input	No	Input Operand to custom instruction
`datab[31..0]`	Input	No	Input Operand to custom instruction
`result[31..0]`	Output	Yes	Result from custom instruction

The only required port for combinatorial custom instructions is the `result[31..0]` port. The `dataa[31..0]` and `datab[31..0]` signals are optional, and should only be included if the application requires input operands. If only a single data port is needed, use `dataa[31..0]`.

Combinatorial Port Operation

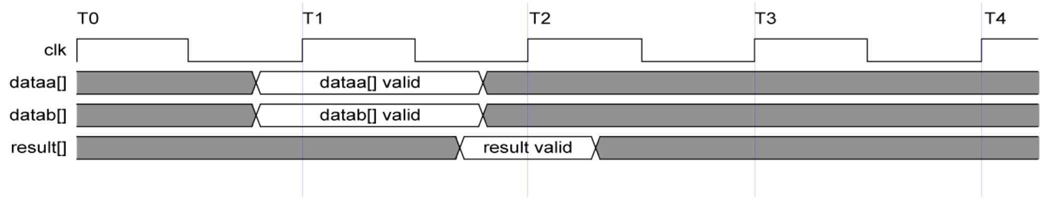
This section describes the combinatorial custom instruction hardware interface port operation. Figure 1–4 shows the combinatorial custom instruction hardware interface timing diagram.

In Figure 1–4, the CPU presents the `dataa[31..0]` and `datab[31..0]` ports on the rising edge of the CPU clock. The CPU reads the `result[31..0]` port on the following rising edge of the CPU clock.

The Nios II processor issues combinatorial custom instructions speculatively, and therefore combinatorial custom instructions cannot have an external interface.

Combinatorial custom instructions can be further optimized by utilizing the extended custom instructions architecture. Refer to "Extended Custom Instruction Architecture" on page 1–9.

Figure 1–4. Combinatorial Custom Instruction Interface Timing Diagram

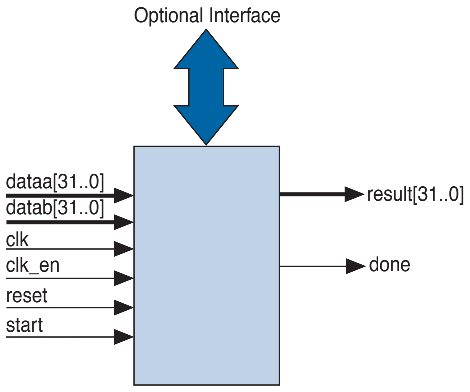


Multi-Cycle Custom Instruction Architecture

Multi-cycle, or sequential, custom instructions consists of a logic block that requires two or more clocks to complete an operation. Multi-cycle custom instruction can complete in either a fixed- or variable-number of clock cycles. Additional control signals are required for multi-cycle custom instructions. See Table 1–3.

Figure 1–5 shows the multi-cycle custom instruction block diagram.

Figure 1–5. Multi-Cycle Custom Instruction Block Diagram



As stated previously, multi-cycle custom instructions can be either fixed or variable length in duration:

- Fixed length: You specify the required number of clock cycles during system generation
- Variable length: The `start` and `done` signals are used in a handshaking scheme to determine when the custom instruction execution is complete.

Table 1–3 lists multi-cycle custom instruction signals.

Table 1–3. Multi-Cycle Custom Instruction Signals

Signal Name	Direction	Required	Application
`clk`	Input	Yes	System clock
`clk_en`	Input	Yes	Clock enable
`reset`	Input	Yes	Synchronous reset
`start`	Input	No	Signals custom instruction logic to start execution
`done`	Output	No	Custom instruction logic signals the CPU that execution is complete.
`dataa[31..0]`	Input	No	Input operand to custom instruction
`datab[31..0]`	Input	No	Input operand to custom instruction
`result[31..0]`	Output	No	Result from custom instruction

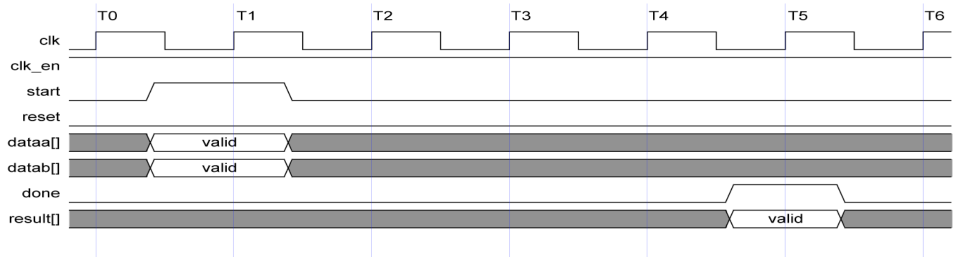
As indicated in Table 1–3, the `clk`, `clk_en`, and `reset` signals are required for multi-cycle custom instructions. However, the `start`, `done`, `dataa[31..0]`, `datab[31..0]`, and `result[31..0]` signals are optional, and should only be used if required for the specific application.

Multi-Cycle Port Operation

The section provides operational details for the multi-cycle, custom instruction hardware interface. Figure 1–6 shows the multi-cycle custom instruction timing diagram.

- The CPU asserts the active high start port on the first clock cycle of execution when the custom instruction issues through the ALU. At this time, the `dataa[31..0]` and `datab[31..0]` signals have valid values and remain valid throughout the duration of the custom instruction execution.
- Fixed or variable length custom instruction port operation:
  - Fixed length: The CPU asserts `start`, waits a specified number of clock cycles, and then reads `result[31..0]`. For an *n*-cycle operation, the custom logic block must present valid data on the (*n*-1) rising edge after the `start` signal is asserted.
  - Variable length: The CPU waits until the active high `done` signal is asserted. The CPU reads the `result[31..0]` port on the clock edge that done is asserted. The custom logic block should present data on the `result[31..0]` port on the same clock that the `done` signal is asserted.
- The Nios II system clock feeds the custom logic block's `clk` signal, and the Nios II master reset feeds the active high `reset` signal. The `reset` signal is asserted only when the whole Nios II system is reset.
- The custom logic block should use the active high `clk_en` signal as a conventional clock qualifier signal and should ignore all clock rising edges while `clk_en` is deasserted.
- Any port in the custom logic block that is not recognized as a custom instruction signal is considered to be an external interface signal.
- Multi-cycle custom instructions can be further optimized utilizing the extended, internal register file, and external interface custom instructions. Refer to "Extended Custom Instruction Architecture" on page 1–9, "Internal Register File Custom Instruction Architecture" on page 1–10, or "External Interface Custom Instruction" on page 1–12.

Figure 1–6. Multi-Cycle Custom Instruction Timing Diagram

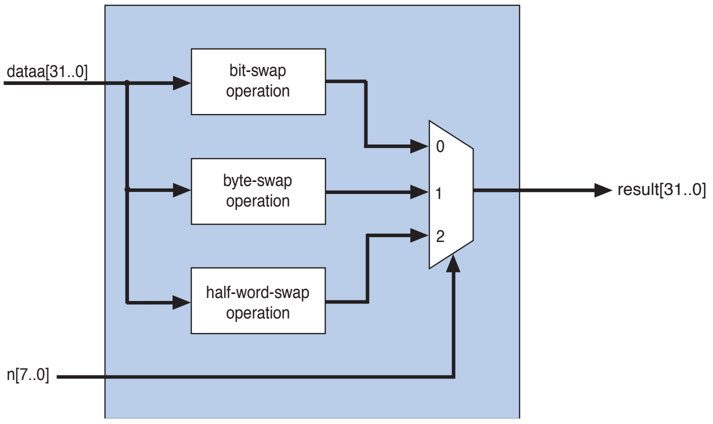


Extended Custom Instruction Architecture

Extended custom instruction architecture allows for a single custom logic block to output results for different operations. Extended custom instructions make use of the `N` field to specify which logic operation is performed by the custom logic. The 8-bit wide `N` field in the op-code allows for 256 different operations for a single block of custom logic.

Figure 1–7 is a block diagram of an extended custom instruction with bit-swap, byte-swap, and half-word-swap operations.

Figure 1–7. Extended Custom Instruction with Swap Operations



The Figure 1–7 swap operations are performed on data coming in via the `dataa[31..0]` port. The `n[7..0]` port is used as a select signal on an output multiplexer to select which operation is presented to the `result[31..0]` port.

Extended custom instructions can be either combinatorial or multi-cycle custom instructions. To implement an extended custom instruction, simply add an `n[7..0]` port to the interface for your custom instruction logic. The bit width of the `n[7..0]` port is a function of a number of operations the extended custom instruction can perform.

Extended Custom Instruction Port Operation

The `n[7..0]` port behaves similarly to the `dataa[31..0]` port. The CPU presents the `n[7..0]` port for execution on the rising edge of clock when `start` is asserted, and the `n[7..0]` port remains stable throughout the execution of the custom instruction. Each custom instruction's bit-width of the `n[7..0]` port is a function of the number of unique operations the custom logic block is able to perform.

All other custom instruction port operations remain the same.

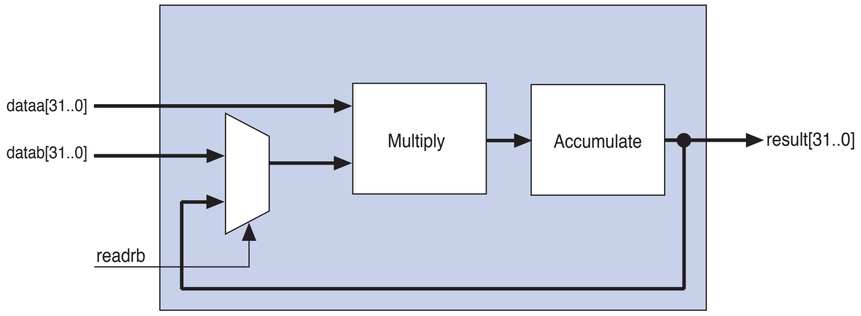
Internal Register File Custom Instruction Architecture

The Nios II processor allows custom instruction logic to access its own internal register file for I/O, which provides you the flexibility to specify if operands should be read from the Nios II processor's register file or the custom instructions internal register file. In addition, results from operations can be written to the local register file rather than the Nios II processor's register file.

Internal registers accessing custom instructions use `readra`, `readrb`, and `writerc` to determine if I/O should take place between the Nios II register file or an internal register file. Additionally, signals `a[4..0]`, `b[4..0]`, and `c[4..0]` specify which internal registers to read from and/or write to. For example, if `readra` is deasserted (i.e., read from the internal register), `a[4..0]` provides an index to the internal register file.

Figure 1–8 shows a simple, multiply-accumulate custom logic block.

Figure 1–8. Multiply-Accumulate Custom Logic Block



When `readrb` is deasserted, the multiplication of `dataa[31..0]` and `datab[31..0]` occurs, and the results are stored in the accumulate register. Those results can be read back by the Nios II processor, or alternatively that value in the accumulator can be read as input to the multiplier by asserting `readrb`.

Table 1–4 lists the internal register file custom instructions signals. The signals are optional and should only be used if required by the application.

Table 1–4. Internal Register File Custom Instruction Signals

Signal Name	Direction	Required	Application
`readra`	Input	No	If `readra` is high, `dataa[31..0]` and `datab[31..0]` are supplied by the Nios II CPU. If `readra` is low, custom instruction logic should read the internal register file indexed by `a[4..0]`.
`readrb`	Input	No	If `readrb` is high, `dataa[31..0]` and `datab[31..0]` are supplied by the Nios II CPU. If `readrb` is low, custom instruction logic should read the internal register file indexed by `a[4..0]`.
`writerc`	Input	No	Signal's custom instructions to write result of `c[4..0]` to custom instruction internal register file.
`a[4..0]`	Input	No	Custom instruction internal register file index
`b[4..0]`	Input	No	Custom instruction internal register file index
`c[4..0]`	Input	No	Custom instruction internal register file index

Internal Register File Custom Instruction Port Operation

The `readra`, `readrb`, `writerc`, and `a[4..0]`, `b[4..0]`, and `c[4..0]` ports behave similarly to `dataa[31..0]`. When the `start` signal is asserted, the CPU presents the `readra`, `readrb`, `writerc`, `a[4..0]`, `b[4..0]`, and `c[4..0]` signals on the rising edge of the CPU clock. All the ports remain stable throughout the execution of the custom instructions.

To determine how to handle register file I/O, custom instruction logic should read the active high `readra`, `readrb`, and `writerc` signals. The `a[4..0]`, `b[4..0]`, and `c[4..0]` ports should be used as register file indexes. When `readra` or `readrb` are not asserted, the custom instruction logic should ignore the corresponding `a[4..0]` or `b[4..0]` port. When `writec` is not asserted, the CPU ignores the value driven on the `result[31..0]` port.

All other custom instructions port operations remain the same.

External Interface Custom Instruction

Figure 1–9 shows that the Nios II processor custom instructions allow you to add an interface to communicate with logic outside of the processor's data path. At system generation, any signals that are not recognized as custom instruction signals will propagate out to the top level of the SOPC Builder module where external logic can access the signals.

Figure 1–9. Custom Instructions Allow the Addition of an External Interface

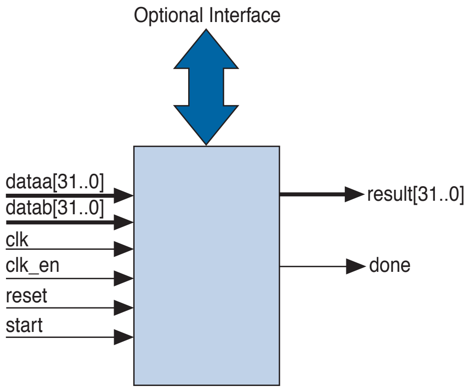


Figure 1–9 shows a multi-cycle custom instruction that has an external memory interface. Because the custom instruction logic is able to access memory external to the CPU, it extends the capabilities of the custom instruction logic.

Custom instruction logic can perform various tasks, e.g., store intermediate results, or read memory to control the custom instruction operation. The optional external interface also provides a dedicated path for data to flow into, or out of, the CPU. For example, custom instruction logic can feed data directly from the CPU's register file to an external FIFO memory buffer, bypassing the processor's data bus.

2. Software Interface

Introduction

The Nios II processor custom instruction details are abstracted from the application code. During the build process the Nios II integrated development environment (IDE) automatically generates macros that allow easy access from application code to custom instructions.

Chapter Overview

This chapter provides custom instruction software interface details including:

- Bit-swap custom instruction examples
- Built-in functions & user-defined macros
- Custom instruction assembly software interface

Bit-Swap Custom Instruction — Example A

Example A shows a portion of the **system.h** header file that defines the macro for a bit-swap custom instruction. This bit-swap example uses one 32-bit input and performs only one function:

```
#define ALT_CI_BSWAP_N 0x00

#define ALT_CI_BSWAP(A) __builtin_custom_ini(ALT_CI_BSWAP_N,(A))
```

In bit-swap Example A, `ALT_CI_BSWAP_N` is defined to be `0x0`, which is the custom instruction's op-code number. The `ALT_CI_BSWAP(A)` macro is mapped to a `gcc` built-in macro that takes a single argument.

Bit-Swap Custom Instruction — Example B

The following bit-swap example illustrates a bit swap custom instruction used in application code.

```
#include "system.h"


int main (void)
{
 int a = 0x12345678;
 int a_swap = 0;

 a_swap = ALT_CI_BSWAP(a);
 return 0;
}
```

In this example, the **system.h** file is included on line 1 to locate the custom instruction macro definitions. Two integers are declared; one on line 6 and one on line 7. Integer `a` is passed as input to the bit swap custom instruction with the results loaded into `a_swap` on line 9.

The bit-swap Example B accommodates most applications using custom instructions. The macros defined by the Nios II IDE only make use of `c` integer types. Occasionally, applications need to make use of input types other than integers, and therefore, need to pass expected return values other than integers.

The Nios II processor custom instructions allow you to define custom macros that allow for other 32-bit input types to interface with custom instructions.

Built in Functions & User-Defined Macros

The Nios II processor uses `gcc` built-in functions to map to custom instructions. Using built-in functions allows for types other than integers to be used with custom instructions. There are 52 uniquely-defined, built-in functions to accommodate the different combinations of the supported types.

Refer to Appendix B, Custom Instruction Built-In Functions for more information on custom instruction's built-in functions.

Built-in functions have the following format:

__builtin_custom_<*return type*>**n**<*parameter types*>

Table 2–1 shows 32-bit input types supported by custom instructions as parameters and return types, as well as the abbreviations used in the built-in function definition.

Table 2–1. 32-Bit Input Types Support by Custom Instructions

Input Type	Built-In Function Abbreviation
`int`	i
`float`	f
`void *`	p

Example C shows the prototype definition for two built-in functions.

Example C:

```
void     __builtin_custom_nf (int n, float dataa);
float    __builtin_custom_fnp(int n, void *dataa);
```

In Example C, the `_builtin_custom_nf` function takes an `int` and a `float` as inputs, and does not return a value. Whereas, the `_builtin_custom_fnp` function takes an integer and a pointer as an input, and returns a float.

To support non-integer input types, you should define macros that map to the specific built-in function required for the application.

Refer to Appendix B, Custom Instruction Built-In Functions for a list of built-in functions.

Example D shows user-defined custom instruction macros used in an application.

Example D:

```
1. /* define void udef_macro1(float data); */
2. #define UDEF_MACRO1_N 0x00
3. #define UDEF_MACRO1(A) __builtin_custom_nf(UDEF_MACRO1_N,
(A));
4. /* define float udef_macro2(void *data); */
5. #define UDEF_MACRO2_N 0x01
6. #define UDEF_MACRO2(B) __builtin_custom_fnp(UDEF_MACRO2_N,
(B));
7.
8. int main (void)
9. {
10. float a = 1.789;
11. float b = 0.0;
12. float *pt_a = &a;
13.
14. UDEF_MACRO1(a);
15. b = UDEF_MACRO2((void *)pt_a);
16. return 0;
17. }
```

On lines 2 through 6, the user-defined macros are declared and mapped to the appropriate built-in functions. The macro `UDEF_MACRO1` takes a float as an input parameter and does not return anything. The macro `UDEF_MACRO2` takes a pointer as an input parameter and returns a float. Lines 14 and 15 show the use of the two user-defined macros.

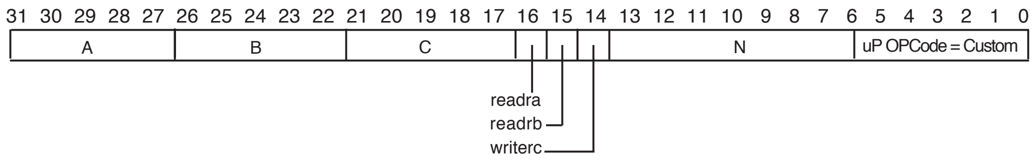
Custom Instruction Assembly Software Interface

The Nios II processor custom instructions are also accessible in assembly code. This section describes the assembly interface.

Custom instructions are R-type instructions with a 6-bit op-code, three 5-bit register index fields, and an 11-bit op-code-extension field. The 11-bit op-code extension field is broken into an 8-bit `N` field for the extended custom instruction and 3 bits for the `readra`, `readrb` and `writerc` bits.

Figure 2–1 is a diagram of the op-code for custom instructions, excerpted from the "Instruction Set Reference" chapter in the *Nios II Processor Reference Handbook*.

Figure 2–1. Op-Code for Custom Instructions Diagram



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

The assembler syntax for the custom instruction is:

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

Where `N` is the custom instruction op-code number, `xC` is the destination register for the `result[31..0]` port, `xA` is operand1, and `xB` is operand2. To access the Nios II CPU's register file, replace `x` with `r`. To access a custom register file, replace `x` with `c`.

The following shows the syntax for two examples of custom instruction assembler calls:

Example 1: `custom 0, r6, r7, r8`

Example 2: `custom 3, c1, r2, c4`

Example 1 executes a custom instruction with an op-code number of `0`. The contents of the Nios II processor register `r7` and `r8` are used as input with the results stored in the Nios II processor register `r6`.

Example 2 executes a custom instruction with an op-code number of `3`. The contents of the Nios II processor register `r2` and custom register `c4` are used as inputs. The results are stored in the custom register `c1`.

3. Implementing a Nios II Processor Custom Instruction

Introduction

This chapter walks you through the process of implementing a Nios® II processor custom instruction, and illustrates the enormous time-savings that are possible with Nios II custom instructions.

Hardware & Software Requirements

The instructions in this chapter require the following hardware and software:

- Quartus® II software version 4.1, SP1 or later
- Nios II development kit
- Nios development board, Stratix® II, Stratix, Stratix Professional, or Cyclone™ Edition

Tutorial Files

The tutorial design files are installed with the Nios II development kit. The hardware design files are stored in the tutorials directory: *<Nios II kit path>***tutorials****Nios2_Custom_Instruction***<board version>*\

Each development board has its own tutorial design file directory (see Table 3–1). The Quartus II project files are contained in the **quartus_project** directory and the hardware for the custom instruction is contained in the **rtl** directory.

Table 3–1. Design File Directories

Nios Development Board	Tutorial Directory
Stratix II Edition	niosII_stratixII_2s60_es
Stratix Edition	niosII_stratix_1s10 & niosII_stratix_1s10_es
Stratix Professional Edition	niosII_stratix_1s40
Cyclone Edition	niosII_cyclone_1c20

This tutorial uses the Nios II integrated development environment (IDE) software template design files located in the following directory:

*<Nios II kit path>***examples****software****ci_tutorial**

☞ The software files in this directory are copied to your working project directory in the Nios II IDE, so there is no need to move the files.

Design Example: Leading Zeros Detector

The leading-zeros-detector design is a simple application that is relevant to floating-point math algorithms. The number of leading zeros found in floating point operands is used during the normalization process before the floating point operation takes place.

This design example counts the number of leading zeros of an array of numbers. Without Nios II custom instructions, the software algorithm loops until it finds the first `1`, which takes several iterations and multiple CPU clocks cycles. However with Nios II custom instructions, the same algorithm can complete in a single clock cycle using priority encoder custom logic block.

Running the Software Algorithm in Nios II IDE

The following guides you through the steps required to run the leading-zeros-detector-software algorithm, while providing an opportunity to see the design's functionality and software algorithm's performance.

This section includes:

- Creating a new Nios II IDE project
- Building and downloading the software application

Creating a New Nios II IDE Project

In this section you will create a new Nios II IDE project using a software template. The example's design files are pre-installed with the Nios II development kit. To create a new Nios II IDE project, perform the following steps:

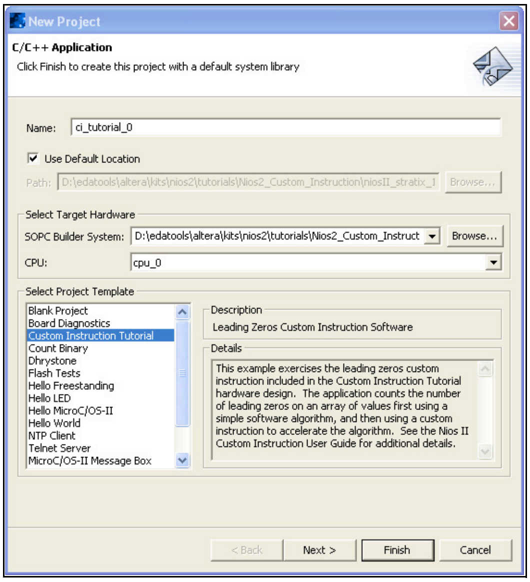
1. Choose **Programs >Altera > Nios II Development Kit** <*version number*>**Nios II IDE** (Windows Start menu).

2. Choose **New > C/C++ Application…** (File menu). The first page of the **New Project** wizard appears. See Figure 3–1 on page 3–3.

3. From Select Project Template, select **Custom Instruction Tutorial**.

4. Leave the default selection for the project's name and ensure that **Use Default Location** is checked.

Figure 3–1. New Project Window



C/C++ Application

Click Finish to create this project with a default system library

Name: ci_tutorial_0

☑ Use Default Location

Path: D:\edatools\altera\kits\nios2\tutorials\Nios2_Custom_Instruction\niosII_stratix_1

Select Target Hardware

SOPC Builder System: D:\edatools\altera\kits\nios2\tutorials\Nios2_Custom_Instruct

CPU: cpu_0

Select Project Template

Blank Project
Board Diagnostics
Custom Instruction Tutorial
Count Binary
Dhrystone
Flash Tests
Hello Freestanding
Hello LED
Hello MicroC/OS-II
Hello World
NTP Client
Telnet Server
MicroC/OS-II Message Box

Description

Leading Zeros Custom Instruction Software

Details

This example exercises the leading zeros custom instruction included in the Custom Instruction Tutorial hardware design. The application counts the number of leading zeros on an array of values first using a simple software algorithm, and then using a custom instruction to accelerate the algorithm. See the Nios II Custom Instruction User Guide for additional details.

5. Click **Browse** in Select Target Hardware. The **Select Target Hardware** dialog box appears.

6. Browse to the custom instruction tutorial hardware design for the Nios development board that you are targeting.

7. Choose the **system.ptf SOPC Builder system** file.

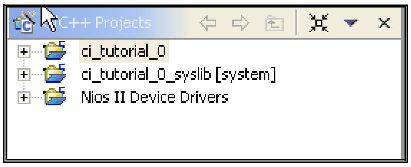
8. Click **Open** to return to the **New Project** wizard. The SOPC Builder System field from the Select Target Hardware window is now specified with the custom instruction project SOPC Builder system. (See Figure 3–1.) In addition, the CPU field now contains the name of the CPU in the system.

9. Click **Finish**.

At this point, the new Nios II project file creation is complete. Figure 3–2 shows that upon successful project creation, the **C/C++ Projects** window contains the following:

- Application project: **ci_tutorial_0**
- HAL software library for the custom instruction hardware: **ci_tutorial_0_syslib**
- Nios II device drivers

Figure 3–2. C/C++ Projects Window



☞ For more information on the HAL software library, refer to the *Nios II Software Developer's Handbook*.

Building & Downloading the Software Application

This section provides the steps to download the leading-zeros software application to the Nios development board.

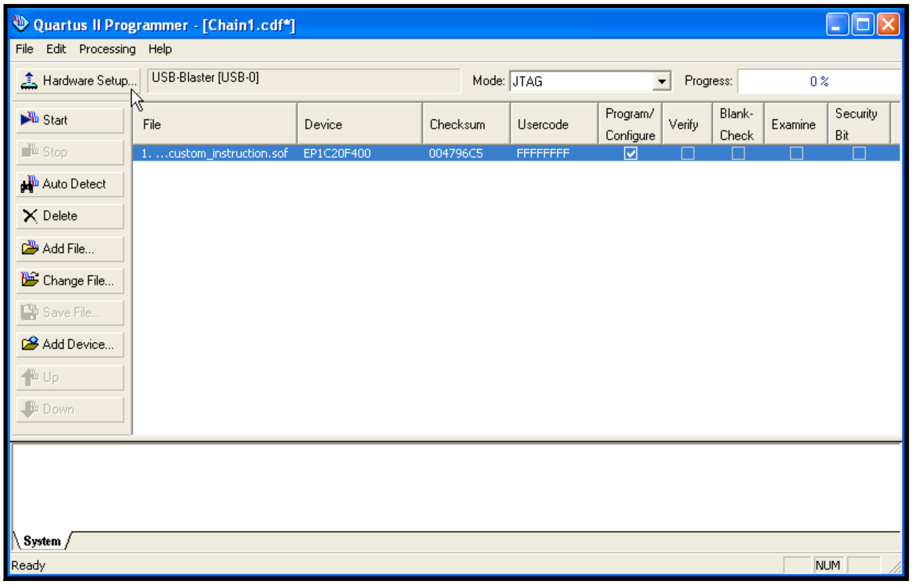
Before the application is executed, you should examine the **leading_zeros_ci.c** file:

- The contents of main should include three sets of test data, i.e., best case, worse case, and random data sets that have the leading zeros counted and placed into another array.
- There are conditional compile statements based on the existence of the `ALT_CI_LEADING_ZERO_DETECTOR` symbol. This is the name of the macro that is defined when the leading-zeros custom instruction is added to the system later in the tutorial.

1. Choose **Run As >Nios II Hardware** (Run menu). The build process begins.

   Depending on the current hardware image on the Nios development board, Nios II IDE might recognize that the current hardware image is not the image required for the tutorial design. If this occurs, the Nios II IDE displays an error message and launches the Quartus II Programmer (see Figure 3–3). If the Quartus II Programmer does not launch, skip to Step 8.

Figure 3–3. Quartus II Programmer Window



USB-Blaster [USB-0]

Mode: JTAG

Progress: 0 %

File	Device	Checksum	Usercode	Program/ Configure	Verify	Blank-Check	Examine	Security Bit
1. ...custom_instruction.sof	EP1C20F400	004796C5	FFFFFFFF	✓	☐	☐	☐	☐

When the Nios II IDE detects that the SOPC Builder system for the current project differs from the SOPC Builder system on the board, you must download an appropriate configuration file for the FPGA.

To download a new FPGA configuration file—SRAM object file (**.sof**)—to the Nios development board, perform the following steps:

2. Choose **Open** (File menu). A Windows Explorer dialog box appears.

3. Select **custom_instruction.sof**. See Figure 3–3.

4. Click **Open** to add the **custom_instruction.sof** programming file to the Quartus II Programmer file list and return to the Quartus II Programmer.

5. From the file list, turn on **Program/Configure** for the **custom_instruction.sof** programming file. See Figure 3–3.

6. Choose **Start** (Processing menu) to download the programming file.

Your Altera programming hardware must first be configured correctly before you can click the **Start** button. If necessary, click **Hardware Setup...** to configure your programming hardware.

7. Exit the Quartus II Programmer and return to the Nios II IDE.

8. In the Nios II IDE, choose **Run As** > **Nios II Hardware** (Run menu). This will start the build process and download the software image to the development board.

After the image is downloaded, the terminal will display the results of running 500 samples through the leading zeros detector in software. The worse case number is if all the samples are a value of `0x1`. The best-case numbers are for the case of `0x80000000`. The random case is random samples. The following is an example of the three sets of test data:

```
Now measuring the time to find leading zeros for 500 samples
**********************************************
Worst Case
[Software] Number of clocks 138410
The number mills-seconds: 2.7681999207

Random Case
[Software] Number of clocks 21926
The number mills-seconds: 0.4385199845

Best Case
[Software] Number of clocks 12434
The number mills-seconds: 0.2486800104
**********************************************
Program Complete.
```

Implementing Custom Instruction Hardware in SOPC Builder

This section walks you through the process of implementing Nios II custom instructions in hardware, and also provides custom instruction tool-flow explanations.

To implement the Nios II custom instruction for the leading-zeros design, you must:

1. Open the custom instruction tutorial hardware design.

2. Add the leading-zeros custom instruction logic to the Nios II CPU.

3. Generate the SOPC Builder system and compile the design in Quartus II.

Open The Custom Instruction Hardware Design

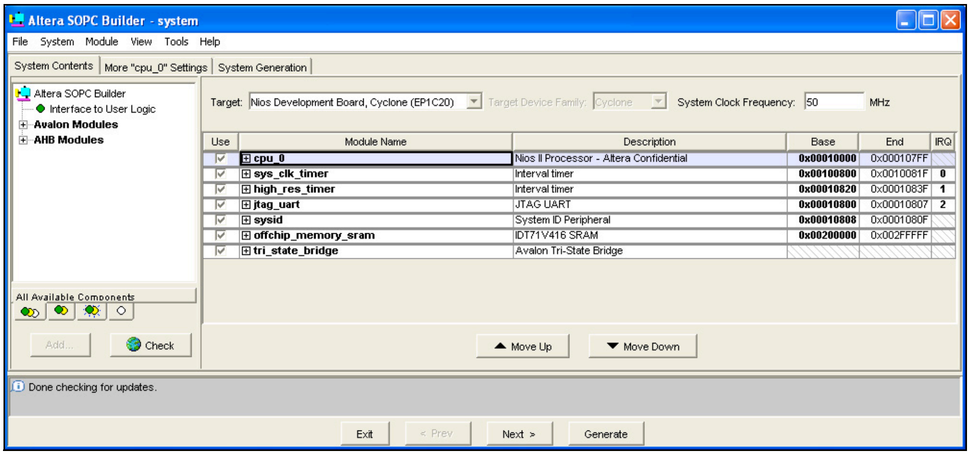
1. Choose **Programs > Altera > Quartus II** <*version*> (Windows Start menu).
2. Choose **Open Project...** (File menu).
3. Browse to the **quartus_project** directory for your board.
4. Choose the **custom_instruction.qpf** and click **Open**.
5. Choose **SOPC Builder…**(Tools menu) to start SOPC Builder.

Add The Leading-Zeros Custom Instruction Logic

This section walks you through the process of adding a custom instruction to an SOPC Builder system, and also provides custom instruction tool-flow explanations.

1. Select **cpu_0** in the **Altera SOPC Builder System Contents** page. See Figure 3–4.

Figure 3–4. SOPC Builder System Contents Page



Target: Nios Development Board, Cyclone (EP1C20) Target Device Family: Cyclone System Clock Frequency: 50 MHz

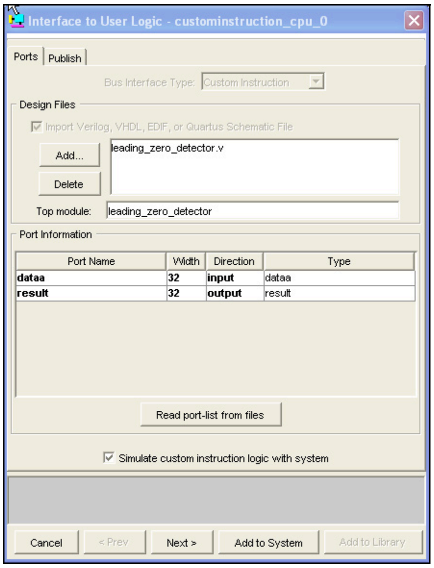
Use	Module Name	Description	Base	End	IRQ
✓	cpu_0	Nios II Processor - Altera Confidential	0x00010000	0x000107FF	
✓	sys_clk_timer	Interval timer	0x00100800	0x0010081F	0
✓	high_res_timer	Interval timer	0x00010820	0x0001083F	1
✓	jtag_uart	JTAG UART	0x00010800	0x00010807	2
✓	sysid	System ID Peripheral	0x00010808	0x0001080F	
✓	offchip_memory_sram	IDT71V416 SRAM	0x00200000	0x002FFFFF	
✓	tri_state_bridge	Avalon Tri-State Bridge			

Done checking for updates.

2. Choose **Edit…** (Module menu). The Nios II Processor Configuration wizard appears.
3. Click on the **Custom Instructions** tab.

4. Click **Import…** The Interface to User Logic wizard appears. See Figure 3–5.

Figure 3–5. Interface to User Logic Wizard


Interface to User Logic - custominstruction_cpu_0
Ports
Publish
Cancel
< Prev
Next >
Add to System
Add to Library
Add...
Delete
Read port-list from files

Bus Interface Type: Custom Instruction

Design Files

☑ Import Verilog, VHDL, EDIF, or Quartus Schematic File

leading_zero_detector.v

Top module: leading_zero_detector

Port Information

Port Name	Width	Direction	Type
dataa	32	input	dataa
result	32	output	result

☑ Simulate custom instruction logic with system

The Interface to User Logic wizard is used to import Nios II custom instruction logic. To import custom instruction logic into the system, you must:

- Add HDL source files to the list.
- Specify the top level module.
- Read in the port list.

Nios II custom instructions require specific port names (see "Custom Instruction Architectural Types" on page 1–4 of Chapter 1"). Any port name not matching the expected port names will be listed as a type export, i.e., export is a type assigned to a signal that is not an expected custom instruction name. Exported signal types are considered to be a part of the custom instruction's external interface.

In addition to specifying custom instruction port information, you have the option of specifying whether or not the custom instruction will be simulated with the system or if it will be black boxed. Also custom instructions can be published for later re-use in different projects.

For more information, refer to *AN 333: Developing Peripherals for SOPC Builder*.

5. Click **Add**. A Windows Explorer dialog box appears. Browse up one directory and descend into the **../rtl** directory.

6. Choose the **leading_zero_detector.v** file in the **../rtl** directory

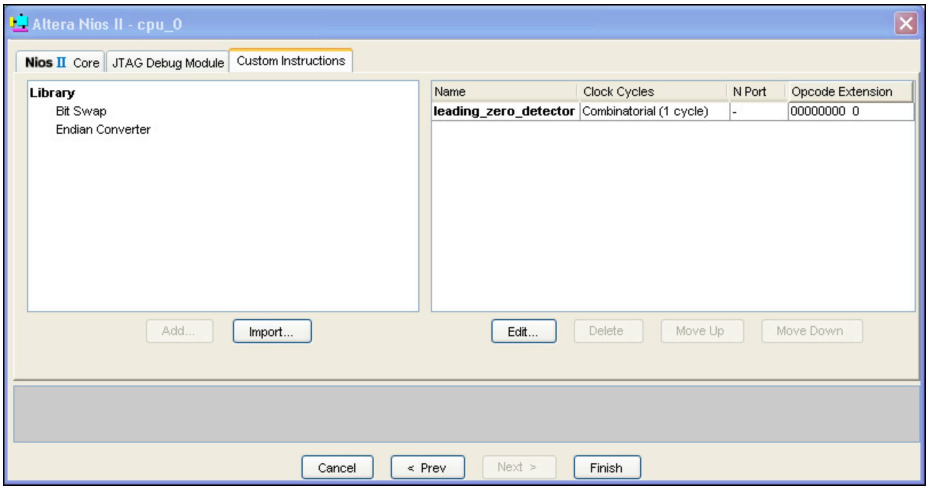
7. Click **Open** to select the **leading_zero_detector.v** file and return back to the Interface to User Logic wizard.

8. Click the **Read port-list from files** button. This will read the port information from the HDL files. The Figure 3–5 example uses `dataa[31..0]` and `result[31..0]` ports.

9. Click **Add to System** to complete the custom instruction importing process. The Nios II Processor Configuration wizard appears.

Figure 3–6 shows that once the custom instruction is imported, the top level module name is listed in the **Name** field.

Figure 3–6. Altera Nios II -cpu_0



Name	Clock Cycles	N Port	Opcode Extension
leading_zero_detector	Combinatorial (1 cycle)	-	00000000 0

For this tutorial to work correctly, the custom instruction's top level module name must be **leading_zero_detector**.

The **Clock Cycles** field shows that the instruction is a combinatorial logic custom instruction. If the tutorial custom instruction was a fixed length, multi-cycle custom instruction instead, you can edit this field to specify the number of clocks. In the case of a variable length multi-cycle custom instruction, the **Clock Cycles** field displays **Variable**.

The **N port** field displays a "-," indicating that the **leading_zero_ detector** design is not an extended custom instruction. In the case of an extended custom instruction, this field shows the width of the N port. The **op-code extension** displays `00000000 0`, that indicates the encoding of the N field in the instruction word.

10. Click **Finish** to add the leading zeros detector custom instruction to the system and return to the SOPC Builder window.

Generate the SOPC Builder System & Compile in Quartus II Software

Now that the custom instruction logic has been added to the system, you are now ready for system generation and Quartus II compilation. During system generation, SOPC Builder will wire the custom logic to the Nios II CPU.

1. Click **Generate** in the SOPC Builder.
2. Click **Exit** when SOPC Builder system generation is complete.
3. Return to the Quartus II window.
4. Choose **Start Compilation** (Processing menu) to begin compilation.

Accessing the Custom Instruction from Software

Now that you have added the custom logic block to hardware, you are ready to access it from software. Because there is a change to the SOPC Builder system contents, the Nios II IDE project needs to be rebuilt to accommodate the changes. One important change will be that the **system.h** header file will be updated with the macros for the custom instruction.

Return to the Nios II IDE "Building & Downloading the Software Application " on page 3–4 and repeat steps 1 through 8. Once you are done, you will see the difference the custom instructions make in performance. Refer to the following console output:

```
Now measuring the time to find leading zeros for 500 samples
**********************************************
Worst Case
[Software] Number of clocks 139632
The number mills-seconds: 2.7926399708
[Hardware] Number of clocks 8443
The number mills-seconds: 0.1688600034

Random Case
[Software] Number of clocks 22419
The number mills-seconds: 0.4483799934
[Hardware] Number of clocks 8056
The number mills-seconds: 0.1611199975

Best Case
[Software] Number of clocks 12402
The number mills-seconds: 0.2480400205
[Hardware] Number of clocks 8032
The number mills-seconds: 0.1606400013
**********************************************
```

Appendix A. Custom Instruction Templates

VHDL & Verilog HDL Templates

This section provides VHDL and Verilog HDL custom instruction templates that you can reference when writing custom instructions in VHDL and Verilog HDL. You can download the template files from the Altera® world-wide website at **www.altera.com/nios**.

VHDL Template

Sample VHDL template file:

```
LIBRARY __library_name;
USE __library_name.__package_name.ALL;

ENTITY __entity_name IS
  PORT(
     signal clk : IN STD_LOGIC; -- CPU's master-input clk <required for multi-cycle>
     signal reset : IN STD_LOGIC; -- CPU's master asynchronous reset <required for multi-cycle>
     signal clk_en: IN STD_LOGIC; -- Clock-qualifier <required for multi-cycle>
     signal start: IN STD_LOGIC; -- True when this instr. issues <required for multi-cycle>
     signal done: OUT STD_LOGIC; -- True when instr. completes <required for variable muli-cycle>
     signal dataa: IN STD_LOGIC_VECTOR (31 DOWNTO 0); -- operand A <always required>
     signal datab: IN STD_LOGIC_VECTOR (31 DOWNTO 0); -- operand B <optional>
     signal n: IN STD_LOGIC_VECTOR (7 DOWNTO 0); -- N-field selector <required for extended>
     signal a: IN STD_LOGIC_VECTOR (4 DOWNTO 0); -- operand A selector <used for Internal register
        file access>
     signal b: IN STD_LOGIC_VECTOR (4 DOWNTO 0); -- operand B selector <used for Internal register
        file access>
     signal c: IN STD_LOGIC; -- result destination selector <used for Internal register file
        access>
     signal readra: IN STD_LOGIC; -- register file index <used for Internal register file access>
     signal readrb: IN STD_LOGIC; -- register file index <used for Internal register file access>
     signal writerc: IN STD_LOGIC; -- register file index <used for Internal register file access>
     signal result : OUT STD_LOGIC_VECTOR (31 DOWNTO 0) -- result <always required>
);
END __entity_name;

ARCHITECTURE a OF __entity_name IS
     signal clk: IN STD_LOGIC;
     signal reset : IN STD_LOGIC;
     signal clk_en: IN STD_LOGIC;
     signal start: IN STD_LOGIC;
     signal readra: IN STD_LOGIC;
     signal readrb: IN STD_LOGIC;
     signal writerc: IN STD_LOGIC;
     signal n:     IN STD_LOGIC_VECTOR (7 DOWNTO 0);
     signal a:     IN STD_LOGIC_VECTOR (4 DOWNTO 0);
     signal b:     IN STD_LOGIC_VECTOR (4 DOWNTO 0);
     signal c:     IN STD_LOGIC_VECTOR (4 DOWNTO 0);
     signal dataa: IN STD_LOGIC_VECTOR (31 DOWNTO 0);
     signal datab: IN STD_LOGIC_VECTOR (31 DOWNTO 0);

     signal result : OUT STD_LOGIC_VECTOR (31 DOWNTO 0);
     signal done: OUT STD_LOGIC;
BEGIN
```

```
-- Process Statement
-- Concurrent Procedure Call
-- Concurrent Signal Assignment
-- Conditional Signal Assignment
-- Selected Signal Assignment
-- Component Instantiation Statement
-- Generate Statement

END a;
```

Verilog HDL Template

Sample Verilog HDL template file:

```
//Verilog Custom Instruction Template

module __module_name(
    clk,    // CPU's master-input clk <required for multi-cycle>
    reset,  // CPU's master asynchronous reset <required for multi-cycle>
    clk_en, // Clock-qualifier <required for multi-cycle>
    start,  // True when this instr. issues <required for multi-cycle>
    done,   // True when instr. completes <required for variable muli-cycle>
    dataa,  // operand A <always required>
    datab,  // operand B <optional>
    n,      // N-field selector  <required for extended>
    a,      // operand A selector <used for Internal register file access>
    b,      // operand b selector <used for Internal register file access>
    c,      // result destination selector <used for Internal register file access>
    readra, // register file index <used for Internal register file access>
    readrb, // register file index <used for Internal register file access>
    writerc,// register file index <used for Internal register file access>
    result  // result <always required>
);

input clk;
    input reset;
input clk_en;
    input start;
    input readra;
    input readrb;
    input writerc;
    input [7:0] n;
    input [4:0] a;
    input [4:0] b;
    input [4:0] c;
input [31:0]dataa;
    input [31:0]datab;

    output[31:0]result;
    output done;

// Port Declaration

// Wire Declaration

// Integer Declaration

// Concurrent Assignment

// Always Construct

endmodule
```

Appendix B. Custom Instruction Built-In Functions

Built-In Functions

This section lists the following custom instruction built-in functions:

- Returning void
- Returning int
- Returning float
- Returning a pointer

Built-In Functions Returning Void

```
void __builtin_custom_n(int n);
void __builtin_custom_ni(int n, intdataa);
void __builtin_custom_nf(int n, floatdataa);
void __builtin_custom_np(int n, void *dataa);
void __builtin_custom_nii(int n, intdataa, intdatab);
void __builtin_custom_nif(int n, intdataa, floatdatab);
void __builtin_custom_nip(int n, intdataa, void *datab);
void __builtin_custom_nfi(int n, floatdataa, intdatab);
void __builtin_custom_nff(int n, floatdataa, floatdatab);
void __builtin_custom_nfp(int n, floatdataa, void *datab);
void __builtin_custom_npi(int n, void *dataa, intdatab);
void __builtin_custom_npf(int n, void *dataa, floatdatab);
void __builtin_custom_npp(int n, void *dataa, void *datab);
```

Built-in Functions Returning int

```
int __builtin_custom_in(int n);
int __builtin_custom_ini(int n, intdataa);
int __builtin_custom_inf(int n, floatdataa);
int __builtin_custom_inp(int n, void *dataa);
int __builtin_custom_inii(int n, intdataa, intdatab);
int __builtin_custom_inif(int n, intdataa, floatdatab);
int __builtin_custom_inip(int n, intdataa, void *datab);
int __builtin_custom_infi(int n, floatdataa, intdatab);
int __builtin_custom_inff(int n, floatdataa, floatdatab);
int __builtin_custom_infp(int n, floatdataa, void *datab);
int __builtin_custom_inpi(int n, void *dataa, intdatab);
int __builtin_custom_inpf(int n, void *dataa, floatdatab);
int __builtin_custom_inpp(int n, void *dataa, void *datab);
```

Built-in Functions Returning float

```
float __builtin_custom_fn(int n);
float __builtin_custom_fni(int n, intdataa);
float __builtin_custom_fnf(int n, floatdataa);
float __builtin_custom_fnp(int n, void *dataa);
float __builtin_custom_fnii(int n, intdataa, intdatab);
float __builtin_custom_fnif(int n, intdataa, floatdatab);
float __builtin_custom_fnip(int n, intdataa, void *datab);
float __builtin_custom_fnfi(int n, floatdataa, intdatab);
float __builtin_custom_fnff(int n, floatdataa, floatdatab);
float __builtin_custom_fnfp(int n, floatdataa, void *datab);
float __builtin_custom_fnpi(int n, void *dataa, intdatab);
float __builtin_custom_fnpf(int n, void *dataa, floatdatab);
float __builtin_custom_fnpp(int n, void *dataa, void *datab);
```

Built-in Functions Returning a Pointer

```
void * __builtin_custom_pn(int n);
void * __builtin_custom_pni(int n, intdataa);
void * __builtin_custom_pnf(int n, floatdataa);
void * __builtin_custom_pnp(int n, void *dataa);
void * __builtin_custom_pnii(int n, intdataa, intdatab);
void * __builtin_custom_pnif(int n, intdataa, floatdatab);
void * __builtin_custom_pnip(int n, intdataa, void *datab);
void * __builtin_custom_pnfi(int n, floatdataa, intdatab);
void * __builtin_custom_pnff(int n, floatdataa, floatdatab);
void * __builtin_custom_pnfp(int n, floatdataa, void *datab);
void * __builtin_custom_pnpi(int n, void *dataa, intdatab);
void * __builtin_custom_pnpf(int n, void *dataa, floatdatab);
void * __builtin_custom_pnpp(int n, void *dataa, void *datab);
```

Appendix C. Porting First-Generation Nios Custom Instructions to Nios II Systems

Hardware & Software Porting Considerations

Most first-generation Nios custom instructions will port over to a Nios II system with minimal changes. This section clarifies hardware and software considerations when porting first-generation Nios custom instructions to your Nios II system.

Hardware Porting Considerations

Both combinatorial and multi-cycle first-generation Nios custom instructions will work with a Nios II system without any changes. However, because parameterized first-generation Nios custom instructions allow a prefix to be passed to the custom instruction logic block, parameterized first-generation Nios custom instructions require a design change.

There is no strict definition for the use of prefixes in first-generation Nios systems, but in most cases the prefix controls the operation performed by the custom instruction. However in a Nios II system, the prefix option is supported directly by extended custom instructions. Therefore, any parameterized first-generation Nios custom instruction that uses a prefix to control the operation executed by the custom instruction should be ported to a Nios II extended custom instruction. Refer to "Extended Custom Instruction Architecture" on page 1–9.

Any other use of the prefix may be accomplished with one of the Nios II custom instruction architecture types. Refer to "Custom Instruction Architectural Types" on page 1–4.

Software Porting Considerations

All first-generation Nios custom instructions will require a small change to application software. Assuming no hardware changes (i.e., not a parameterized first-generation custom instruction), software porting should be nothing more than a search and replace operation. The first-generation Nios and Nios II system macro definition nomenclature is different; therefore first-generation Nios macro calls should be replaced by the Nios II macros. In the case of parameterized first-generation custom instructions, additional changes will be required depending on the implementation. Refer to Chapter 2, Software Interface.