

# Nios<sup>®</sup> II

## Nios II Custom Instruction User Guide

---



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

Copyright © 2008 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-1.5



I.S. EN ISO 9001

## Chapter 1. Nios II Custom Instruction Overview

Introduction .....	1-1
Custom Instruction Overview .....	1-2
Implementing Custom Instruction Hardware .....	1-3
Implementing Custom Instruction Software .....	1-4
Custom Instruction Types .....	1-4
Combinational Custom Instruction .....	1-5
Combinational Port Operation .....	1-6
Multi-Cycle Custom Instruction .....	1-7
Multi-Cycle Port Operation .....	1-8
Extended Custom Instruction .....	1-9
Extended Custom Instruction Port Operation .....	1-11
Internal Register File Custom Instruction .....	1-11
Internal Register File Custom Instruction Port Operation .....	1-13
External Interface Custom Instruction .....	1-13

## Chapter 2. Software Interface

Introduction .....	2-1
Custom Instruction Examples .....	2-1
Built-In Functions and User-Defined Macros .....	2-2
Custom Instruction Assembly Software Interface .....	2-4

## Chapter 3. Implementing a Nios II Custom Instruction

Introduction .....	3-1
Design Example: Cyclic Redundancy Checksum (CRC) .....	3-1
Implementing Custom Instruction Hardware in SOPC Builder .....	3-1
Open the Component Editor .....	3-2
Add the Synthesis HDL File .....	3-2
Add Simulation Files .....	3-3
Configure the Custom Instruction Signal Type .....	3-3
Set Up Custom Instruction Interfaces .....	3-4
Set the Component Wizard Details .....	3-5
Save and Add the Custom Instruction .....	3-5
Generate the SOPC Builder System and Compile in the Quartus II Software .....	3-6
Accessing the Custom Instruction from Software .....	3-6
Viewing Results on Nios II Console .....	3-7
User-defined Custom Instruction Macro .....	3-9

## Appendix A. Custom Instruction Templates

Overview .....	A-1
----------------	-----

VHDL Template .....	A-1
Verilog HDL Template .....	A-2

**Appendix B. Custom Instruction Built-In Functions**

Overview .....	B-1
Built-In Functions Returning void .....	B-1
Built-in Functions Returning int .....	B-1
Built-in Functions Returning float .....	B-2
Built-in Functions Returning a Pointer .....	B-2

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

Overview .....	C-1
Hardware Porting Considerations .....	C-1
Software Porting Considerations .....	C-1

**Appendix D. Floating Point Custom Instructions**

Overview .....	D-1
Adding Floating Point Custom Instructions .....	D-1

**Additional Information**

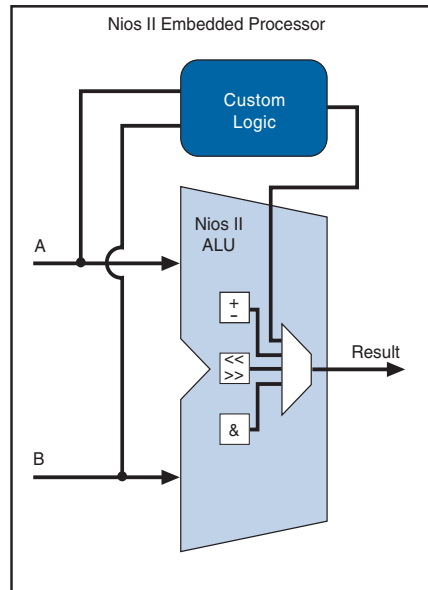
Revision History .....	Info-i
How to Contact Altera .....	Info-i
Typographic Conventions .....	Info-ii

## Introduction

With the Altera Nios II embedded processor, you as the system designer can accelerate time-critical software algorithms by adding custom instructions to the Nios II processor instruction set. Using custom instructions, you can reduce a complex sequence of standard instructions to a single instruction implemented in hardware. You can use this feature for a variety of applications, for example, to optimize software inner loops for digital signal processing (DSP), packet header processing, and computation-intensive applications. The Nios II configuration wizard, part of 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 contains the following sections:

- “Custom Instruction Overview” on page 1–2.
- “Custom Instruction Types” on page 1–4.

For information regarding the custom instruction software interface, refer to [Chapter 2, Software Interface](#). For step-by-step instructions for implementing a custom instruction, see [Chapter 3, Implementing a Nios II Custom Instruction](#).

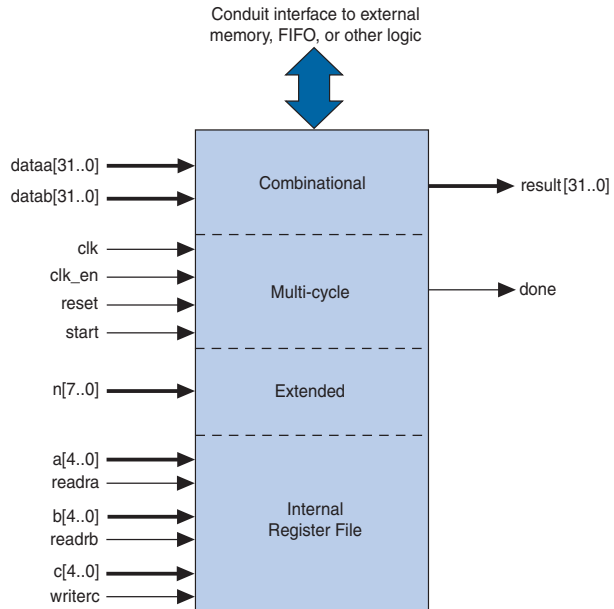
## Custom Instruction Overview

Nios II custom instructions are custom logic blocks adjacent to the ALU in the processor’s data path. Custom instructions give you the ability to tailor the Nios II processor core to meet the needs of a particular application. You 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 tradeoffs at any point in the design process.

## Implementing Custom Instruction Hardware

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

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



The basic operation of Nios II custom instruction logic is to receive input on the `dataa` and/or `datab` port, and drive out the result on its `result` port. The custom instruction logic provides a result based on the inputs provided by the Nios II processor.

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

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

## Implementing Custom Instruction Software

The Nios II custom instruction software interface is simple and abstracts the details of the custom instruction from the software developer. For each custom instruction, the Nios II integrated development environment (IDE) generates a macro in the system header file, `system.h`. You can use the macro directly in your C or C++ application code, and you do not need to program assembly to access custom instructions. Software can also invoke custom instructions in Nios II processor assembly language.



For more information on custom instruction software interface, refer to [Chapter 2, Software Interface](#).

## Custom Instruction Types

There are different types of custom instruction available to suit the requirements of the application. The chosen type determines what the hardware interface looks like.

[Table 1–1](#) shows custom instruction types, application, and the associated hardware ports.

Type	Application	Hardware Ports
Combinational	Single clock cycle custom logic blocks	<ul style="list-style-type: none"> <li>• <code>dataa[31..0]</code></li> <li>• <code>datab[31..0]</code></li> <li>• <code>result[31..0]</code></li> </ul>
Multi-cycle	Multi clock cycle custom logic block of fixed or variable durations	<ul style="list-style-type: none"> <li>• <code>dataa[31..0]</code></li> <li>• <code>datab[31..0]</code></li> <li>• <code>result[31..0]</code></li> <li>• <code>clk</code></li> <li>• <code>clk_en(1)</code></li> <li>• <code>start</code></li> <li>• <code>reset</code></li> <li>• <code>done</code></li> </ul>
Extended	Custom logic blocks that are capable of performing multiple operations	<ul style="list-style-type: none"> <li>• <code>dataa[31..0]</code></li> <li>• <code>datab[31..0]</code></li> <li>• <code>result[31..0]</code></li> <li>• <code>clk</code></li> <li>• <code>clk_en(1)</code></li> <li>• <code>start</code></li> <li>• <code>reset</code></li> <li>• <code>done</code></li> <li>• <code>n[7..0]</code></li> </ul>



**Table 1–1. Custom Instruction Types, Application and Hardware Ports**

Type	Application	Hardware Ports
Internal Register File	Custom logic blocks that access internal register files for input and/or output	<ul style="list-style-type: none"> <li>• dataa[31..0]</li> <li>• datab[31..0]</li> <li>• result[31..0]</li> <li>• clk</li> <li>• clk_en</li> <li>• start</li> <li>• reset</li> <li>• done</li> <li>• n[7..0]</li> <li>• a[4..0]</li> <li>• readra</li> <li>• b[4..0]</li> <li>• readrb</li> <li>• c[4..0]</li> <li>• writerc</li> </ul>
External Interface	Custom logic blocks that interface to logic outside of the Nios II processor's data path	Standard custom instruction ports, plus user-defined interface to external logic.

**Note for Table 1–1:**

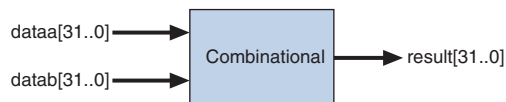
- (1) The `clk_en` must be connected to the `clk_en` of all registers in custom instruction in case the Nios II processor needs to stall the custom instruction during execution.

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

### Combinational Custom Instruction

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

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

**Figure 1–3. Combinational Custom Instruction Block Diagram**

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

[Table 1–2](#) lists the combinational custom instruction ports.

<i>Table 1–2. Combinational Custom Instruction Ports</i>			
Port Name	Direction	Required	Application
<code>dataa[31..0]</code>	Input	No	Input operand to custom instruction
<code>datab[31..0]</code>	Input	No	Input operand to custom instruction
<code>result[31..0]</code>	Output	Yes	Result from custom instruction

The only required port for combinational custom instructions is the `result` port. The `dataa` and `datab` ports are optional. Include them only if the custom instruction functionality requires input operands. If the custom instruction requires only a single input port, use `dataa`.

### *Combinational Port Operation*

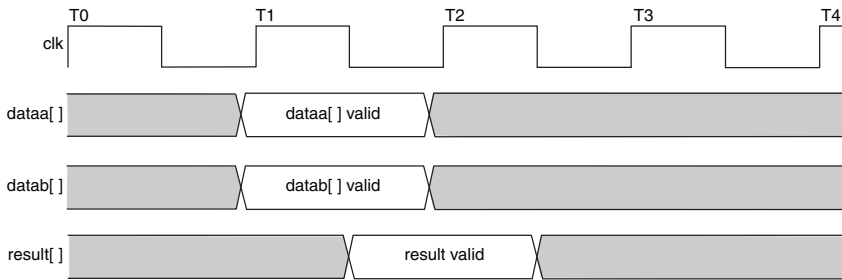
This section describes the combinational custom instruction hardware port operation. [Figure 1–4](#) shows the combinational custom instruction hardware port timing diagram.

In [Figure 1–4](#), the processor presents the input data on the `dataa` and `datab` ports on the rising edge of the processor clock. The processor reads the `result` port on the rising edge of the following processor clock.

The Nios II processor issues a combinational custom instruction speculatively; that is, it optimizes execution by issuing the instruction before knowing whether it is necessary, and ignores the result if it is not required. Therefore, a combinational custom instruction must not have side effects. In particular, a combinational custom instruction cannot have an external interface.

You can further optimize combinational custom instructions by implementing the extended custom instruction. Refer to [“Extended Custom Instruction” on page 1–9](#).

**Figure 1–4. Combinational Custom Instruction Port Timing Diagram**

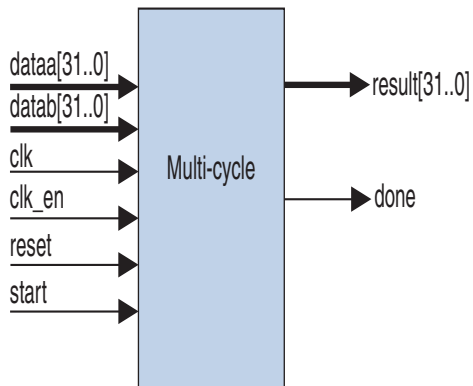


### Multi-Cycle Custom Instruction

Multi-cycle or sequential, custom instructions consist of a logic block that requires two or more clock cycles to complete an operation. Additional control ports 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**



Multi-cycle custom instructions can complete in either a fixed or variable number of clock cycles.

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

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

Port Name	Direction	Required	Application
<code>clk</code>	Input	Yes	System clock
<code>clk_en</code>	Input	Yes	Clock enable
<code>reset</code>	Input	Yes	Synchronous reset
<code>start</code>	Input	No	Commands custom instruction logic to start execution
<code>done</code>	Output	No	Custom instruction logic indicates to the processor that execution is complete.
<code>dataa[31..0]</code>	Input	No	Input operand to custom instruction
<code>datab[31..0]</code>	Input	No	Input operand to custom instruction
<code>result[31..0]</code>	Output	No	Result from custom instruction

As indicated in Table 1–3, the `clk`, `clk_en`, and `reset` ports are required for multi-cycle custom instructions. However, the `start`, `done`, `dataa`, `datab`, and `result` ports are optional. Implement them only if the custom instruction functionality specifically needs them.

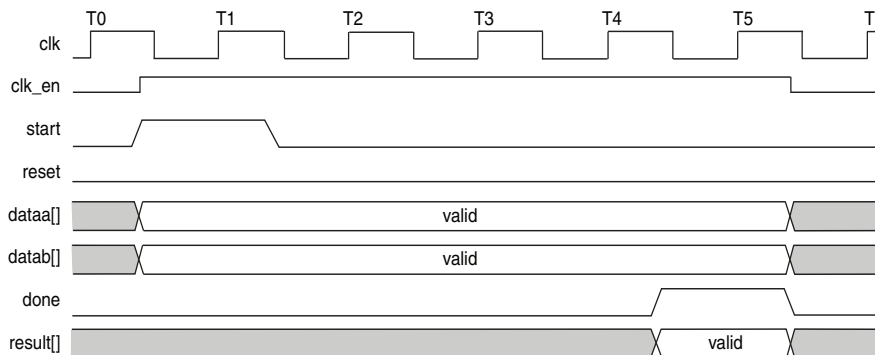
### *Multi-Cycle Port Operation*

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

- The processor asserts the active high `start` port on the first clock cycle of the custom instruction execution. At this time, the `dataa` and `datab` ports have valid values and remain valid throughout the duration of the custom instruction execution. The `start` signal is asserted for a single clock cycle.
- Fixed or variable length custom instruction port operation:
  - Fixed length: Once the custom instruction is started, the processor waits a specified number of clock cycles, and then reads `result`. For an  $n$ -cycle operation, the custom logic block

- must present valid data on the  $n^{\text{th}}$  rising edge after the custom instruction is executed.
- Variable length: The processor waits until the active high `done` port is asserted. The processor reads the `result` port on the clock edge that `done` is asserted. The custom logic block must present data on the `result` port on the same clock cycle that the `done` port is asserted.
  - The Nios II system clock feeds the custom logic block's `clk` port, and the Nios II system's master reset feeds the active high `reset` port. The `reset` port is asserted only when the whole Nios II system is reset.
  - The custom logic block must treat the active high `clk_en` port as a conventional clock qualifier signal, ignoring `clk` while `clk_en` is deasserted.
  - You can further optimize multi-cycle custom instructions by implementing the extended internal register file, or by creating external interface custom instructions. Refer to “[Extended Custom Instruction](#)” on page 1–9, “[Internal Register File Custom Instruction](#)” on page 1–11, or “[External Interface Custom Instruction](#)” on page 1–13.

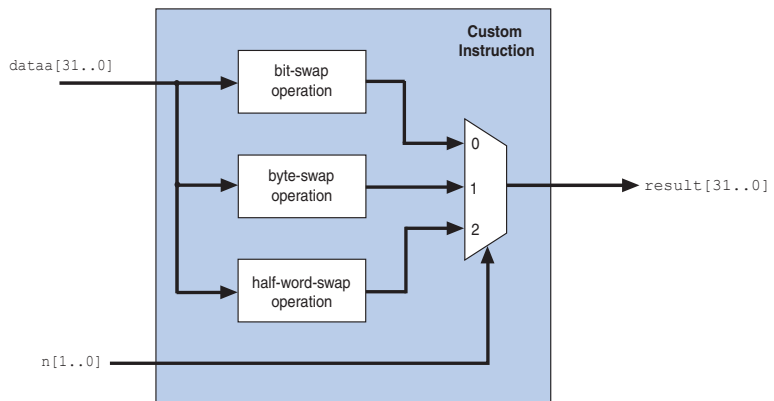
**Figure 1–6. Multi-Cycle Custom Instruction Timing Diagram**




## Extended Custom Instruction

Extended custom instruction allows a single custom logic block to implement several different operations. Extended custom instructions use an index to specify which operation the logic block performs. The index can be up to eight-bits wide, allowing a single custom logic block to implement up to 256 different operations.

[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 custom instruction in [Figure 1–7](#) performs swap operations on data received at the `dataa` port. It uses the two-bit-wide `n` port to select the output from a multiplexer, determining which result is presented to the `result` port.

 This logic is just a simple example, using a multiplexer on the output. You can implement function selection based on an index in any way that is appropriate for your application.

Extended custom instructions can be combinational or multi-cycle custom instructions. To implement an extended custom instruction, simply add an `n` port to your custom instruction logic. The bit width of the `n` port is a function of the number of operations the custom logic block can perform.

Extended custom instructions occupy multiple custom instruction indices. For example, the custom instruction illustrated in [Figure 1–7](#) occupies 4 indices, because `n` is two bits wide. Therefore, when this instruction is implemented in a Nios II system, it leaves  $256 - 4 = 252$  available indices.

For information about the custom instruction index, see [“Custom Instruction Assembly Software Interface”](#) on page 2–4.

### *Extended Custom Instruction Port Operation*

The `n` port behaves similarly to the `dataa` port. For example, with an extended variable multi-cycle custom instruction, the processor presents the index value to the `n` port on the rising edge of the clock when `start` is asserted, and the `n` port remains stable throughout the execution of the custom instruction.

All other custom instruction port operations remain the same as combinational and multi-cycle custom instructions.

### **Internal Register File Custom Instruction**

The Nios II processor allows custom instruction logic to access its own internal register file. This provides you the flexibility to specify if the custom instruction reads its operands from the Nios II processor's register file or from the custom instruction's own internal register file. In addition, a custom instruction can write its results to the local register file rather than the Nios II processor's register file.

Custom instructions containing internal register files use `readra`, `readrb`, and `writerc` to determine if the custom instruction should use the internal register file or the `dataa`, `datab`, and `result` signals. Additionally, ports `a`, `b`, and `c` specify which internal registers to read from and/or write to. For example, if `readra` is deasserted (that is, read from the internal register), `a` provides an index to the internal register file. Ports `a`, `b`, and `c` are five bits each, allowing you to address up to 32 registers.

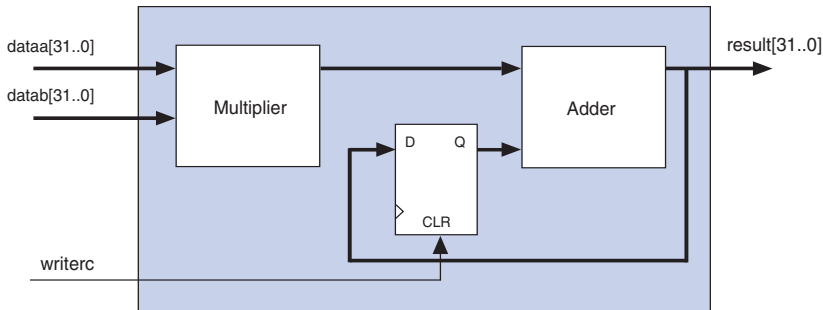
For further details of Nios II custom instruction implementation, refer to the *Instruction Set Reference* chapter of the *Nios II Processor Reference Handbook*.

Table 1-4 lists the internal register file custom instruction ports. Use these optional ports only if the custom instruction functionality requires them.

Port Name	Direction	Required	Application
readra	Input	No	If readra is high, the Nios II processor supplies dataa. If readra is low, custom instruction logic reads the internal register file indexed by a.
readrb	Input	No	If readrb is high, the Nios II processor supplies datab. If readrb is low, custom instruction logic reads the internal register file indexed by b.
writerc	Input	No	If writerc is high, the Nios II processor writes to the result port. If writerc is low, custom instruction logic writes to the internal register file indexed by c.
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

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

**Figure 1-8. Multiply-Accumulate Custom Logic Block**



When writerc is deasserted, the Nios II processor ignores the value driven by result port. The accumulated value is then stored into an internal register. On the other hand, the processor can read the value on result port by asserting writerc. At the same time, the internal register is cleared so that it is ready for a new round of multiply and accumulate operations.



### Internal Register File Custom Instruction Port Operation

The `readra`, `readrb`, `writerc`, and `a`, `b`, and `c` ports behave similarly to `dataa`. When the custom instructions are started, the processor presents the `readra`, `readrb`, `writerc`, `a`, `b`, and `c` ports on the rising edge of the processor clock. All the ports remain stable throughout the execution of the custom instructions.

To determine how to handle register file, custom instruction logic reads the active high `readra`, `readrb`, and `writerc` ports. The logic uses the `a`, `b`, and `c` ports as register file indexes. When `readra` or `readrb` are asserted, the custom instruction logic ignores the corresponding `a` or `b` port. When `writerc` is asserted, the custom instruction logic ignores the `c` port.

All other custom instruction port operations remain the same as combinational and multi-cycle custom instructions.

### External Interface Custom Instruction

Figure 1–9 shows that the Nios II custom instructions allow you to add an interface to communicate with logic outside of the processor's data path. At system generation, conduits propagate out to the top level of the SOPC Builder module where external logic can access the signals.

Because the custom instruction logic is able to access memory external to the processor, it extends the capabilities of the custom instruction logic.

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

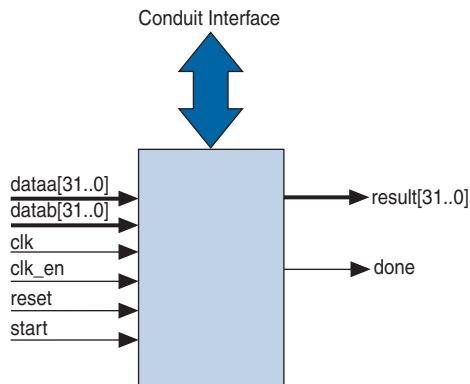


Figure 1–9 shows a multi-cycle custom instruction that has an external memory interface.

Custom instruction logic can perform various tasks, for example, store intermediate results, or read memory to control the custom instruction operation. The conduit interface also provides a dedicated path for data to flow into, or out of, the processor. For example, custom instruction logic can feed data directly from the processor's register file to an external first-in first-out (FIFO) memory buffer.

### Introduction

The Nios II custom instruction software interface abstracts logic implementation details from the application code. During the build process the Nios II IDE generates macros that allow easy access from application code to custom instructions.

This chapter provides custom instruction software interface details including:

- “Custom Instruction Examples” on page 2-1
- “Built-In Functions and User-Defined Macros” on page 2-2
- “Custom Instruction Assembly Software Interface” on page 2-4

### Custom Instruction Examples

Example 2-1 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.

---

#### Example 2-1. Bit Swap Macro Definition

```
#define ALT_CI_BITSWAP_N 0x00

#define ALT_CI_BITSWAP(A) __builtin_custom_ini(ALT_CI_BITSWAP_N, (A))
```

---

In Example 2-1, `ALT_CI_BITSWAP_N` is defined to be `0x0`, which is the custom instruction’s index. The `ALT_CI_BITSWAP(A)` macro is mapped to a `gcc` built-in function that takes a single argument.



For more information on `gcc` built-in function, see [Appendix B, Custom Instruction Built-In Functions](#).

Example 2-2 illustrates a bit-swap custom instruction used in application code.

---

### Example 2-2. Bit Swap Instruction Usage

```
1. #include "system.h"
2.
3.
4. int main (void)
5. {
6.     int a = 0x12345678;
7.     int a_swap = 0;
8.
9.     a_swap = ALT_CI_BITSWAP(a);
10. return 0;
11.}
```

---

In Example 2-2, the **system.h** file is included so that the application software can use the custom instruction macro definitions. The example declares two integers, `a` and `a_swap`. Integer `a` is passed as input to the bit swap custom instruction with the results loaded into `a_swap`.

Example 2-2 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.



You can define custom macros for Nios II custom instructions, that allow for other 32-bit input types to interface with custom instructions.

## Built-In Functions and User-Defined Macros

The Nios II processor uses `gcc` built-in functions to map to custom instructions. By default, the integer type custom instruction is defined in **system.h** file. However, by using built-in functions, software can use non-integer types with custom instructions. There are 52 uniquely-defined built-in functions to accommodate the different combinations of the supported types.

Built-in function names have the following format:

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

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

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

Example 2-3 shows the prototype definitions for two built-in functions.

---

### **Example 2-3. Built-in Functions**

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

---

In Example 2-3, the `__builtin_custom_nf` function takes a `float` as an input, and does not return a value. In contrast, the `__builtin_custom_fnp` function takes a pointer as an input, and returns a `float`.

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



Refer to [Appendix B, Custom Instruction Built-In Functions](#) for detailed information, and a list of built-in functions.

Example 2-4 shows user-defined custom instruction macros used in an application.

### Example 2–4. Custom Instruction Macro Usage

```
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 line numbers 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. Line numbers 14 and 15 show the use of the two user-defined macros.

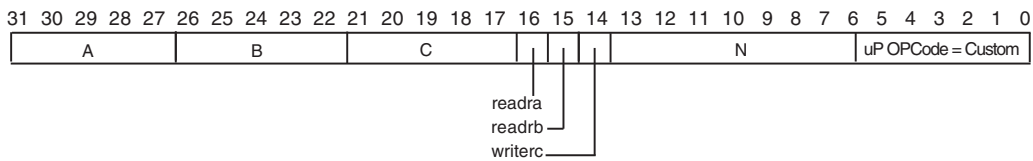
## Custom Instruction Assembly Software Interface

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

Custom instructions are R-type instructions, containing:

- A 6-bit opcode
- Three 5-bit register index fields
- Three 1-bit fields for the `readra`, `readrb`, and `writerc` signals
- An 8-bit `N` field, used for the custom instruction index (opcode extension), and optionally including a function select subfield

Figure 2–1 on page 2–5 is a diagram of the custom instruction word.

**Figure 2–1. Custom Instruction Word**

**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 processor's register rA, 0 otherwise  
 readrb = 1 if instruction uses processor's register rB, 0 otherwise  
 writerc = 1 if instruction provides result for processor's register rC, 0 otherwise

Bits 5–0 are the Nios II custom instruction opcode, as specified in the “*Instruction Opcodes*” section in the *Instruction Set Reference* chapter of the *Nios II Processor Reference Handbook*. This value appears in every custom instruction.

The N field, bits 13–6, is the custom instruction index. The custom instruction index distinguishes between different custom instructions, allowing the Nios II processor to support up to 256 distinct custom instructions. Depending on the type of custom instruction, the N field represents one of the following:

- A unique custom instruction index, for logic that implements a single custom function
- An extended custom instruction index, for logic that implements several custom functions

Example 2–5 shows the assembly language syntax for the custom instruction.

#### **Example 2–5. Custom Instruction Assembly Syntax**

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

In Example 2–5, N is the custom instruction index, xC is the destination for the result[31..0] port, xA is the dataa port, and xB is the datab port. To access the Nios II processor's register file, replace x with r. To

access a custom register file, replace  $x$  with  $c$ . The usage of  $r$  and  $c$  determines whether the custom instruction is presented `readra`, `readrb` and `writerc` high or low.

Examples 2-6, 2-7 and 2-8 show the syntax for custom instruction assembler calls.

---

### **Example 2-6. Custom Instruction Index=0**

```
custom 0, r6, r7, r8
```

---

Example 2-6 executes a custom instruction with an index of 0. The contents of the Nios II processor registers `r7` and `r8` are used as input, with the results stored in the Nios II processor register `r6`.

---

### **Example 2-7. Custom Instruction Index=3**

```
custom 3, c1, r2, c4
```

---

Example 2-7 executes a custom instruction with an index 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`.

---

### **Example 2-8. Custom Instruction Index=4**

```
custom 4, r6, c9, r2
```

---

Example 2-8 executes a custom instruction with an index of 4. The contents of the custom register `c9` and Nios II processor register `r2` are used as inputs. The result is stored in the Nios II processor register `r6`.

For further information about the binary format of custom instructions, refer to the *Instruction Set Reference* chapter of the *Nios II Processor Reference Handbook*.



### Introduction

This chapter describes the process of implementing a Nios II custom instruction with the SOPC Builder component editor. The component editor enables you to create new SOPC Builder components, including Nios II custom instructions. This chapter also describes the process of accessing Nios II custom instruction from software.

For detailed information about the SOPC Builder component editor, refer to the *Component Editor* chapter of the *Quartus II Handbook Volume 4: SOPC Builder*.

### Design Example: Cyclic Redundancy Checksum (CRC)

The cyclic redundancy check (CRC) algorithm detects the corruption of data during transmission. It detects a higher percentage of errors than a simple checksum. The CRC calculation consists of an iterative algorithm involving XOR and shift operations. These operations are carried out concurrently in hardware and iteratively in software. Since the operations are carried out concurrently, the execution is much faster in hardware.

The CRC design files are used to demonstrate the steps to implement an extended multi-cycle Nios II custom instruction. These design files are available on the Altera website, at [www.altera.com/literature/lit-nio2.jsp](http://www.altera.com/literature/lit-nio2.jsp), accompanying this document.

Before you start the design example, you must set up the design environment to accommodate the processes described in the following sections. Refer to the **readme.txt** file in the extracted design files archived under the Quartus II Project Setup section.

### Implementing Custom Instruction Hardware in SOPC Builder

This section describes the custom instruction tool-flow, and walks you through the process of implementing a Nios II custom instruction. Implementing a Nios II custom instruction hardware entails the following tasks:

- “Open the Component Editor” on page 3–2
- “Add the Synthesis HDL File” on page 3–2
- “Add Simulation Files” on page 3–3
- “Set Up Custom Instruction Interfaces” on page 3–4
- “Configure the Custom Instruction Signal Type” on page 3–3
- “Set the Component Wizard Details” on page 3–5
- “Save and Add the Custom Instruction” on page 3–5

- “Generate the SOPC Builder System and Compile in the Quartus II Software” on page 3–6

The following tasks detail the steps required to import the custom instruction into the design, and add it to the Nios II processor.

### Open the Component Editor

1. Open the SOPC Builder system.



For detailed information about opening and working with SOPC Builder systems, refer to the *Volume 4: SOPC Builder* of the *Quartus II Handbook*, or to the Quartus II Help system.

2. On the SOPC Builder **System Contents** tab, double-click **cpu**. The **Nios II Processor** configuration wizard appears.
3. On the **Parameter Settings** page, click the **Custom Instructions** tab.
4. Click **Import**. The component editor appears, displaying the **Introduction** tab.

### Add the Synthesis HDL File

1. Click **Next** to display the **HDL Files** tab.
2. Click **Add**.
3. Browse to the directory containing the hardware description language (HDL) file(s), select the files needed, and click **Open**. The files to be added in this demonstration are **CRC\_Custom\_Instruction.v** and **CRC\_Component.v** located in **crc\_hw** directory.
4. Turn on the **Synth** parameter for each of the HDL files added. This is to indicate that these files have synthesis equivalents. Both HDL files imported in this demonstration have synthesis equivalents.
5. Turn on the **Top** parameter to indicate where the top-level HDL file is. The top-level HDL file for this demonstration is **CRC\_Custom\_Instruction.v**.



The Quartus II Analysis and Synthesis checks the design for errors in the background when a top-level file is selected. A message will appear to indicate the analysis is complete. Make sure there is no error message.

6. Click **Top Level Module** and select the name of the top-level module of your custom instruction logic. The top-level module of the design in this demonstration is **CRC\_Custom\_Instruction**.

## Add Simulation Files

These steps are performed only if you wish to simulate the system in ModelSim.

1. Click **Add**.
2. Browse to the directory containing the simulation files, select the files needed, and click **Open**.
3. Turn on the **Sim** parameter for each of the simulation files added.



If the HDL file is used for both synthesis and simulation purposes, turn on both the **Synth** and **Sim** parameters. Turn on only the **Sim** parameter if the file has no synthesis equivalent or it is used solely for simulation.

## Configure the Custom Instruction Signal Type

1. Click **Next** to display the **Signals** tab. There are several ports (signals) listed.
2. For every port listed, carry out the following steps:
  - a. Select the port.
  - b. In the **Interface** drop-down list, select the interface name you wish to assign the port with, e.g. **nios\_custom\_instruction\_slave\_0**. The name you see may vary as this depends on what name you set in the **Interface** tab. For this demonstration, select **nios\_custom\_instruction\_slave\_0** as the interface type for every port.
  - c. In the **Signal Type** drop-down list, select the signal type corresponding to the port name. For example, if the custom instruction hardware presents the result on a port named **output**, you set the **Signal Type** to **result**.



For further information about signal types, see [“Custom Instruction Types”](#) on page 1–4.

### Set Up Custom Instruction Interfaces

The following steps describe how to setup interfaces for your custom instruction logic.

1. Click **Next** to display the **Interfaces** tab. The default interface type displayed is **Custom Instruction Slave**.
2. Rename the interface by typing the desired name in the **Name** field. You can use the default name if you do not intend to change the name.
3. Set the **Operands** parameter value to the number of operands used for the custom instruction. The custom instruction used in this demonstration has only one operand, so set the **Operands** parameter to one.
4. If you are using fixed multi-cycle type custom instruction, set the **Clock Cycles** parameter value to the number of clock cycles your custom instruction logic needs. The design in this demonstration is of variable multi-cycle type, so set the **Clock Cycles** parameter to zero.
5. If a message reporting **Interface has no signals** appears, click **Remove Interfaces With No Signals** to remove the message.

If your custom instruction logic requires additional interfaces, either to the Avalon-MM system interconnect fabric or outside the SOPC Builder system, you can specify the interfaces here. The following steps will show you to add the additional interfaces.



Most custom instructions use some combination of standard custom instruction ports, such as `dataaa`, `datab` and `result`, and do not require additional interfaces.

1. Click **Add Interface**. The added interface has **Custom Instruction Slave** interface type as default.
2. Select the interface type that you prefer under the **Type** list.
3. Set the parameters for the newly created interface according to the system requirement. The design in this demonstration does not have any external interface, so you can skip these steps.

## Set the Component Wizard Details

The following steps guide you to set the component wizard details.

1. Click **Next** to display the **Component Wizard** tab.
2. Specify the following values to the corresponding fields.
  - a. Type **CRC** in the **Class Name** and **Display Name** fields.
  - b. Assign **1.0** in the **Version** field.
  - c. Leave the **Group** field blank.
  - d. It is optional for you to fill in the **Description**, **Created by** and **Icon** fields.
3. When you complete steps 1 to 3, the bottom pane of the dialog box displays **Info: No errors or warnings** message. If it does not, review the preceding steps in the previous sections.

## Save and Add the Custom Instruction

1. Click **Finish**. A dialog box appears prompting you to save the changes made before exiting.
2. Click **Yes, Save** to finish importing the custom instruction and return to the Nios II processor configuration wizard.
3. You need to restart the Nios II processor configuration wizard. To do this, click **Finish** to close the wizard and double-click **cpu**.
4. Select **CRC** from the library on the left side panel of the **Custom Instructions** tab, and click **Add** to add it to the Nios II processor.
  - The **Clock Cycles** field indicates the type of custom instruction: combinational logic, multi-cycle, extended, internal register file, or external interface. If the custom instruction is a fixed length, multi-cycle custom instruction, you must edit this field to specify the number of clocks. You must determine this number based on knowledge of the custom instruction state machine. In the case of a variable length multi-cycle custom instruction, the **Clock Cycles** field displays **Variable**.
  - The **N port** field indicates whether the custom instruction is an extended type. In the case of an extended custom instruction, this field indicates which bits in the n port serve as the function select. Otherwise it displays a dash (-).

- The **Opcode Extension** field displays the custom instruction index (N field) in the instruction word. The value appears in both binary and decimal formats. For further information about the N field, see “[Custom Instruction Assembly Software Interface](#)” on page 2–4.
5. In the Nios II processor configuration wizard, click **Finish** to finish adding the custom instruction to the system and return to the SOPC Builder window.

### Generate the SOPC Builder System and Compile in the Quartus II Software

After the custom instruction logic is added to the system, you are ready for system generation and the Quartus II compilation. During system generation, SOPC Builder connects the custom logic to the Nios II processor.

1. Click **Generate** in SOPC Builder. SOPC Builder generates the system RTL. This might take several seconds.
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.



For detailed instructions on generating SOPC Builder systems, refer to the *Volume 4: SOPC Builder* of the *Quartus II Handbook*, or to the SOPC Builder Help system.

## Accessing the Custom Instruction from Software

Adding a custom instruction to a Nios II processor results in a significant change to the SOPC Builder system. In this section, you create and build a new software project using Nios II software build flow, and run the software that accesses the custom instruction. You can locate the application software source files in the design files archive.

Table 3–1 shows the application software source files and their corresponding descriptions.

<i>Table 3–1. CRC Application Software Source Files</i>	
File Name	Description
<code>crc_main.c</code>	Main program that populates random test data, executes the CRC both in software and with the custom instruction, validates the output and reports the processing time.
<code>crc.c</code>	Software CRC algorithm run by Nios II processor.
<code>crc.h</code>	Header file for <code>crc.c</code>
<code>ci_crc.c</code>	Program that accesses CRC custom instruction.
<code>ci_crc.h</code>	Header file for <code>ci_crc.c</code>



You can refer to the comments inside each software file for details.

To run the application software, you need to create an Executable and Linked Format (`.elf`) file first. Refer to `readme.txt` file in the design files archive under Nios II Software Build Flow section.

## Viewing Results on Nios II Console

The application program runs the software version of the CRC first, followed by the custom instruction CRC. The processing times, as well as the throughput for each implementation, are calculated to show the improvement of using a custom instruction over a software algorithm.

Example 3–1 shows the expected result while running the software.

### Example 3–1. The Expected Result for Running CRC Example

```
*****
Comparison between software and custom instruction CRC32
*****

System specification
-----
System clock speed = 85.0 MHz
Number of buffer locations = 16
Size of each buffer = 65535 bytes
```

## Accessing the Custom Instruction from Software

---

Initializing all of the buffers with pseudo-random data

-----  
Initialization completed

Running the software CRC

-----  
Completed

Running the optimized software CRC

-----  
Completed

Running the custom instruction CRC

-----  
Completed

Validating the CRC results from all implementations

-----  
All CRC implementations produced the same results

Processing time for each implementation

-----  
Software CRC = 23885.87 ms  
Optimized software CRC = 16360.97 ms  
Custom instruction CRC = 343.67 ms

Processing throughput for each implementation

-----  
Software CRC = 0.35 Mbps  
Optimized software CRC = 0.51 Mbps  
Custom instruction CRC = 24.41

Speedup ratio

-----  
Custom instruction CRC vs software CRC = 69.5  
Custom instruction CRC vs optimized software CRC = 47.6  
Optimized software CRC vs software CRC = 1.5

---

The results you see may be different depending on the memory characteristics of the target board and the clock speed of the example design.

Example 3-1 shows that the custom instruction CRC is about 70 times faster than the unoptimized CRC calculated purely in software and is about 50 times faster than the optimized version of the software CRC.



## User-defined Custom Instruction Macro

This example software uses a user-defined macro to access the CRC custom instruction. Example 3-2 shows the macro that is defined in `ci_crc.c` file.

---

### Example 3-2. CRC Custom Instruction Accessing Macro

```
#define CRC_CI_MACRO(n, A) \
__builtin_custom_ini(ALT_CI_CRC_N + (n & 0x7), (A))
```

---

This macro takes only one `int` type input operand and return an `int` type value. The CRC custom instruction comprises extended type, so the `n` value in macro `CRC_CI_MACRO()` is used to indicate which operation is to be performed by the custom instruction. The custom instruction index is added with the value of `n`. The `n` value must be masked because of the fact that the `n` port of a custom instruction consists of only three bits.

To use the macro to initialize the custom instruction, for example, code in Example 3-3 can be placed in the application software.

---

### Example 3-3. Using User-defined Macro to Initialize Custom Instruction Logic

```
/* The custom instruction CRC will initialize to the initial remainder value */
CRC_CI_MACRO (0,0);
```



For details on each operation of the CRC custom instruction and the corresponding `n` value, refer to the comments in `ci_crc.c` file.

As shown in Examples 3-2 and 3-3, you can define the macro in your application to accommodate your requirements. For example, you can determine the number and type of input operands, decide whether to assign a return value or not and vary the custom instruction index. However, the macro definition and usage must be consistent with the ports declaration of the custom instruction. For example, if you define the macro to be returning an `int` value, the custom instruction must have a result port.



For details about writing software for Nios II custom instructions, see [Chapter 2, Software Interface](#).



## Overview

This section provides VHDL and Verilog HDL custom instruction wrapper file templates that you can reference when writing custom instructions in VHDL and Verilog HDL.



You can get the template files from `<nios2eds installation directory>/examples/verilog/custom_instruction_templates` directory (if you are using Verilog HDL), or `<nios2eds installation directory>/examples/vhdl/custom_instruction_templates` directory (if you are using VHDL).

## VHDL Template

Sample VHDL template file:

```
-- VHDL Custom Instruction Template File for Internal Register Logic

library ieee;
use ieee.std_logic_1164.all;

entity custominstruction is
port(
    signal clk: in std_logic;-- CPU system clock (required for multi-cycle or extended multi-cycle)
    signal reset: in std_logic;-- CPU master asynchronous active high reset (required for multi-cycle or extended multi-cycle)
    signal clk_en: in std_logic;-- Clock-qualifier (required for multi-cycle or extended multi-cycle)
    signal start: in std_logic;-- Active high signal used to specify that inputs are valid (required for multi-cycle or extended multi-cycle)
    signal done: out std_logic;-- Active high signal used to notify the CPU that result is valid (required for variable multi-cycle or extended variable multi-cycle)
    signal n: in std_logic_vector(7 downto 0);-- N-field selector (required for extended), modify width to match the number of unique operations within the custom instruction
    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 a: in std_logic_vector(4 downto 0);-- Internal operand A index register
    signal b: in std_logic_vector(4 downto 0);-- Internal operand B index register
    signal c: in std_logic_vector(4 downto 0);-- Internal result index register
    signal readra: in std_logic;-- Read operand A from CPU (otherwise use internal operand A)
    signal readrb: in std_logic;-- Read operand B from CPU (otherwise use internal operand B)
    signal writerc: in std_logic;-- Write result to CPU (otherwise write to internal result)
    signal result: out std_logic_vector(31 downto 0)-- result (always required)
);
end entity custominstruction;
architecture a_custominstruction of custominstruction is

    -- local custom instruction signals

begin

    -- custom instruction logic (note: external interfaces can be used as well)

    -- use the n[7..0] port as a select signal on a multiplexer to select the value to feed result[31..0]

end architecture a_custominstruction;
```

# Verilog HDL Template

Sample Verilog HDL template file:

```
// Verilog Custom Instruction Template File for Internal Register Logic

module custominstruction(
    clk,        // CPU system clock (required for multi-cycle or extended multi-cycle)
    reset,     // CPU master asynchronous active high reset (required for multi-cycle or extended multi-
cycle)
    clk_en,    // Clock-qualifier (required for multi-cycle or extended multi-cycle)
    start,    // Active high signal used to specify that inputs are valid (required for multi-cycle or
extended multi-cycle)
    done,     // Active high signal used to notify the CPU that result is valid (required for variable
multi-cycle or extended variable multi-cycle)
    n,        // N-field selector (required for extended)
    dataa,    // Operand A (always required)
    datab,   // Operand B (optional)
    a,        // Internal operand A index register
    b,        // Internal operand B index register
    c,        // Internal result index register
    readra,  // Read operand A from CPU (otherwise use internal operand A)
    readrb,  // Read operand B from CPU (otherwise use internal operand B)
    writerc, // Write result to CPU (otherwise write to internal result)
    result   // Result (always required)
);

//INPUTS
inputclk;
inputreset;
inputclk_en;
inputstart;
input[7:0]n; // modify width to match the number of unique operations within the custom instruction
input[4:0]a;
input[4:0]b;
input[4:0]c;
inputreadra;
inputreadrb;
inputwriterc;
input[31:0]dataa;
input[31:0]datab;

//OUTPUTS
outputdone;
output[31:0]result;

// custom instruction logic (note: external interfaces can be used as well)

// use the n[7..0] port as a select signal on a multiplexer to select the value to feed result[31..0]

endmodule
```

## Overview

The Nios II gcc compiler is customized with built-in functions to support custom instructions. This section lists the built-in functions.



For more information about gcc built-in functions, refer to [www.gnu.org](http://www.gnu.org).

Nios II custom instruction built-in functions are of the following types:

- 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, int dataa);
void __builtin_custom_nf (int n, float dataa);
void __builtin_custom_np (int n, void *dataa);
void __builtin_custom_nii (int n, int dataa, int datab);
void __builtin_custom_nif (int n, int dataa, float datab);
void __builtin_custom_nip (int n, int dataa, void *datab);
void __builtin_custom_nfi (int n, float dataa, int datab);
void __builtin_custom_nff (int n, float dataa, float datab);
void __builtin_custom_nfp (int n, float dataa, void *datab);
void __builtin_custom_npi (int n, void *dataa, int datab);
void __builtin_custom_npf (int n, void *dataa, float datab);
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, int dataa);
int __builtin_custom_inf (int n, float dataa);
int __builtin_custom_inp (int n, void *dataa);
int __builtin_custom_inii (int n, int dataa, int datab);
int __builtin_custom_inif (int n, int dataa, float datab);
int __builtin_custom_inip (int n, int dataa, void *datab);
int __builtin_custom_infi (int n, float dataa, int datab);
int __builtin_custom_inff (int n, float dataa, float datab);
int __builtin_custom_infp (int n, float dataa, void *datab);
int __builtin_custom_inpi (int n, void *dataa, int datab);
int __builtin_custom_inpf (int n, void *dataa, float datab);
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, int dataa);
float __builtin_custom_fnf (int n, float dataa);
float __builtin_custom_fnp (int n, void *dataa);
float __builtin_custom_fnii (int n, int dataa, int datab);
float __builtin_custom_fnif (int n, int dataa, float datab);
float __builtin_custom_fnip (int n, int dataa, void *datab);
float __builtin_custom_fnfi (int n, float dataa, int datab);
float __builtin_custom_fnff (int n, float dataa, float datab);
float __builtin_custom_fnfp (int n, float dataa, void *datab);
float __builtin_custom_fnpi (int n, void *dataa, int datab);
float __builtin_custom_fnpf (int n, void *dataa, float datab);
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, int dataa);
void * __builtin_custom_pnf (int n, float dataa);
void * __builtin_custom_pnp (int n, void *dataa);
void * __builtin_custom_pnii (int n, int dataa, int datab);
void * __builtin_custom_pnif (int n, int dataa, float datab);
void * __builtin_custom_pnip (int n, int dataa, void *datab);
void * __builtin_custom_pnfi (int n, float dataa, int datab);
void * __builtin_custom_pnff (int n, float dataa, float datab);
void * __builtin_custom_pnfp (int n, float dataa, void *datab);
void * __builtin_custom_pnpi (int n, void *dataa, int datab);
void * __builtin_custom_pnpf (int n, void *dataa, float datab);
void * __builtin_custom_pnpp (int n, void *dataa, void *datab);
```



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

## Overview

You can port most first-generation Nios custom instructions to a Nios II system with minimal changes. This chapter clarifies hardware and software considerations when porting first-generation Nios custom instructions to your Nios II system.

## Hardware Porting Considerations

You can use both combinational and multi-cycle first-generation Nios custom instructions 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” on page 1-9](#).

Any other use of the prefix can be accomplished with one of the Nios II custom instruction architecture types. Refer to [“Custom Instruction 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 is simply 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](#).





## Overview

Adding the floating point custom instructions to your SOPC system may cause certain conflicts. This chapter gives you options to overcome these conflicts.

## Adding Floating Point Custom Instructions

When you add the floating point custom instructions to your SOPC system and then run the regular IDE create project flow, the following flags will get added to the your gcc command line. These flags indicate which custom instructions are present (by specifying the opcode extensions) and select the appropriate version of newlib that uses the custom instruction.

```
"gcc ... -mcustom-fpu-cfg=60-1"
```

```
"gcc ... -mcustom-fpu-cfg=60-2"
```

The 60-1 is used when you do not add the divider and 60-2 is used when you use a custom fp divider.

These `-mcustom-fpu` flags force single precision constants to be used. If you want double precision constants, you must remove the `-mcustom-fpu` flag and replace it with the individual compiler flags as shown in Example D-1.

### Example D-1. Mcustom-fpu Flags

```
"gcc ... -mcustom-fpu-cfg=60-1" change to
```

```
"gcc ... -mcustom-fmuls=252, -mcustom-fadds=253,  
-mcustom-fsubs=254, -mcustom-fdivs=255"
```

```
"gcc ... -mcustom-fpu-cfg=60-2" change to
```

```
"gcc ... -mcustom-fmuls=252, -mcustom-fadds=253,  
-mcustom-fsubs=254, -mcustom-fdivs=255"
```

Change the flags only if you need to. When you replace the `-mcustom-fpu` flags, you lose your floating point custom instruction support in your `newlib` calls, and you have to use the emulated or slower version of the instruction instead.



**Revision History** The table below displays the revision history for chapters in this User Guide.

Nios II Custom Instruction User Guide Revision History			
Chapter	Date	Version	Changes Made
All	May 2008	1.5	<ul style="list-style-type: none"><li>• Minor corrections to terminology and usage.</li><li>• Add new tutorial design.</li><li>• Describe new custom instruction import flow.</li></ul>
2	May 2007	1.4	Add title and core version number to page footers
All	May 2007	1.3	Minor corrections to terminology and usage.
1	May 2007	1.3	Describe new component editor import flow.
3	May 2007	1.3	Remove tutorial design
All	December 2004	1.2	Updates for the Nios® II version 1.1 release.
All	September 2004	1.1	Updates for the Nios II version 1.01 release.
All	May 2004	1.0	First release of custom instruction user guide for the Nios II processor.

## 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	<a href="http://www.altera.com/mysupport/">www.altera.com/mysupport/</a>
Technical training	<a href="http://www.altera.com/training/custrain@altera.com">www.altera.com/training/custrain@altera.com</a>
Product literature	<a href="http://www.altera.com/literature/">www.altera.com/literature/</a>
FTP site	<a href="ftp://ftp.altera.com">ftp.altera.com</a>

*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
<b>Bold Type with Initial Capital Letters</b>	Command names, dialog box titles, checkbox options, and dialog box options are shown in bold, initial capital letters. Example: <b>Save As</b> dialog box.
<b>bold type</b>	External timing parameters, directory names, project names, disk drive names, filenames, filename extensions, and software utility names are shown in bold type. Examples: <b>f<sub>MAX</sub></b> , <b>lqdesigns</b> directory, <b>d:</b> drive, <b>chiptrip.gdf</b> file.
<i>Italic Type with Initial Capital Letters</i>	Document titles are shown in italic type with initial capital letters. Example: <i>AN 75: High-Speed Board Design</i> .
<i>Italic type</i>	Internal timing parameters and variables are shown in italic type. Examples: <i>t<sub>PIA</sub></i> , <i>n + 1</i> .  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.
	A caution calls attention to a condition or possible situation that can damage or destroy the product or the user's work.
	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.