# Nios II Custom Instruction User Guide

2015-11-02

UG-N2CSTNST



You can accelerate time-critical software algorithms by adding custom instructions to the Nios II processor.

Custom instructions allow you to 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. Each custom instruction is a component in the Qsys system. You can add as many as 256 custom instructions to your system.

## Nios II Custom Instruction Overview

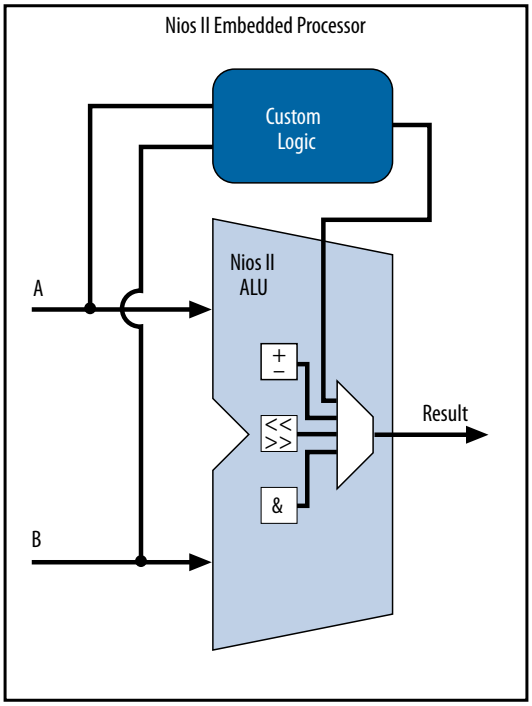
Custom instructions give you the ability to tailor the Nios II processor to meet the needs of a particular application. You can accelerate time critical software algorithms by converting them to custom hardware logic blocks. Because it is easy to alter the design of the FPGA-based Nios II processor, custom instructions provide an easy way to experiment with hardware-software tradeoffs at any point in the design process.

The custom instruction logic connects directly to the Nios II arithmetic logic unit (ALU) as shown in the following figure.



101 Innovation Drive, San Jose, CA 95134

**Figure 1: Custom Instruction Logic Connects to the Nios II ALU**




**Related Information**

- **Implementing a Nios II Custom Instruction in Qsys** on page 17
  Step-by-step instructions for implementing a custom instruction
- **Software Interface** on page 12
  Information about the custom instruction software interface

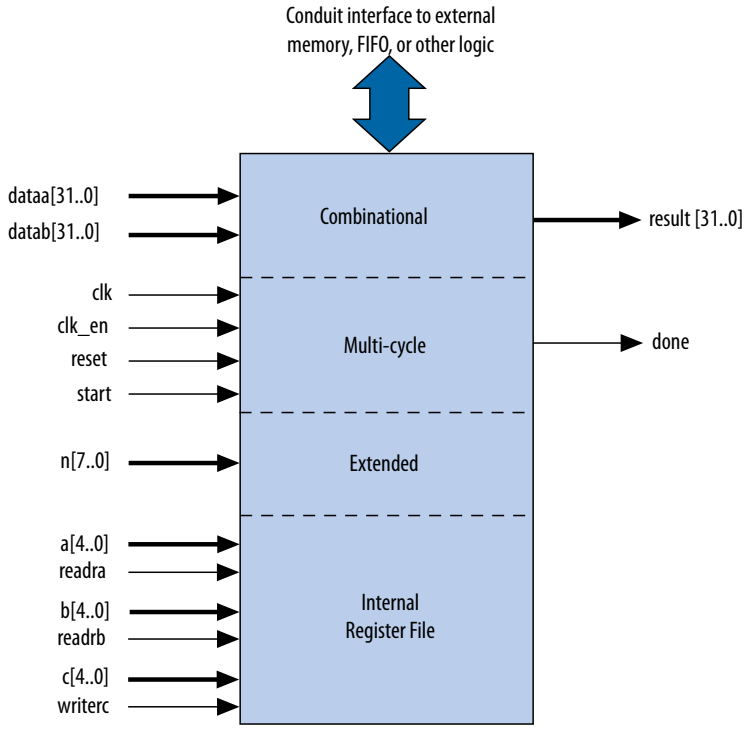
## Custom Instruction Implementation

Nios II custom instructions are custom logic blocks adjacent to the arithmetic logic unit (ALU) in the processor's datapath.

When custom instructions are implemented in a Nios II system, each custom operation is assigned a unique selector index. The selector index allows software to specify the desired operation from among up to 256 custom operations. The selector index is determined at the time the hardware is instantiated with the Qsys software. Qsys exports the selection index value to **system.h** for use by the Nios II software build tools.

## Custom Instruction Hardware Implementation

**Figure 2: Hardware Block Diagram of a Nios II Custom Instruction**




A Nios II custom instruction logic block interfaces with the Nios II processor through three ports: `dataa`, `datab`, and `result`.

The custom instruction logic provides a result based on the inputs provided by the Nios II processor. The Nios II custom instruction logic receives input on its `dataa` port, or on its `dataa` and `datab` ports, and drives the result to its `result` port.

The Nios II processor supports several types of custom instructions. The figure above shows all the ports required to accommodate all custom instruction types. Any particular custom instruction implementation requires only the ports specific to its custom instruction type.

The figure above 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 datapath.

## Custom Instruction Software Implementation

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 Embedded Design Suite (EDS) 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 code to access custom instructions. Software can also invoke custom instructions in Nios II processor assembly language.

**Related Information**

**Software Interface** on page 12
More information about the custom instruction software interface

## Custom Instruction Types

Different types of custom instructions are available to meet the requirements of your application. The type you choose determines the hardware interface for your custom instruction.

**Table 1: Custom Instruction Types, Applications, and Hardware Ports**

| Instruction Type | Application | Hardware Ports |
|---|---|---|
| Combinational | Single clock cycle custom logic blocks. | • `dataa[31:0]`<br>• `datab[31:0]`<br>• `result[31:0]` |
| Multicycle | Multi-clock cycle custom logic blocks of fixed or variable durations. | • `dataa[31:0]`<br>• `datab[31:0]`<br>• `result[31:0]`<br>• `clk`<br>• `clk_en` (1)<br>• `start`<br>• `reset`<br>• `done` |
| Extended | Custom logic blocks that are capable of performing multiple operations | • `dataa[31:0]`<br>• `datab[31:0]`<br>• `result[31:0]`<br>• `clk`<br>• `clk_en` (1)<br>• `start`<br>• `reset`<br>• `done`<br>• `n[7:0]` |

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

| Instruction Type | Application | Hardware Ports |
|---|---|---|
| Internal register file | Custom logic blocks that access internal register files for input or output or both. | • dataa[31:0]<br>• datab[31:0]<br>• result[31:0]<br>• clk<br>• clk_en(1)<br>• start<br>• reset<br>• done<br>• n[7:0]<br>• a[4:0]<br>• readra<br>• b[4:0]<br>• readrb<br>• c[4:0]<br>• writerc |
| External interface | Custom logic blocks that interface to logic outside of the Nios II processor's datapath | Standard custom instruction ports, plus user-defined interface to external logic. |

## Combinational Custom Instructions

A combinational custom instruction is a logic block that completes its logic function in a single clock cycle.

A combinational custom instruction must not have side effects. In particular, a combinational custom instruction cannot have an external interface. This restriction exists because the Nios II processor issues combinational custom instructions speculatively, to optimize execution. It issues the instruction before knowing whether it is necessary, and ignores the result if it is not required.

A basic combinational custom instruction block, with the required ports shown in "Custom Instruction Types", implements a single custom operation. This operation has a selection index determined when the instruction is instantiated in the system using Qsys.

You can further optimize combinational custom instructions by implementing the extended custom instruction. Refer to "Extended Custom Instructions".

**Related Information**

- **Extended Custom Instructions** on page 8
- **Custom Instruction Types** on page 4
  List of standard custom instruction hardware ports, to be used as signal types

### Combinational Custom Instruction Ports

A combinational custom instruction must have a `result` port, and may have optional `dataa` and `datab` ports.

**Figure 3: Combinational Custom Instruction Block Diagram**


In the figure above, the `dataa` and `datab` ports are inputs to the logic block, which drives the results on the `result` port. Because the logic function completes in a single clock cycle, a combinational custom instruction does not require control ports.

**Table 2: Combinational Custom Instruction Ports**

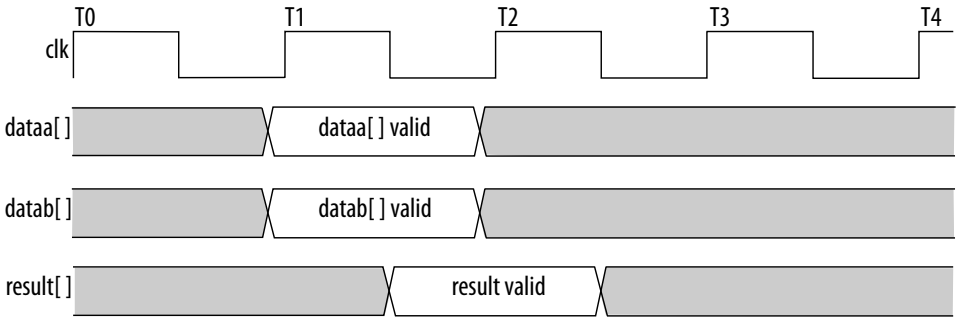
| Port Name | Direction | Required | Description |
|---|---|---|---|
| `dataa[31:0]` | Input | No | Input operand to custom instruction |
| `datab[31:0]` | Input | No | Input operand to custom instruction |
| `result[31:0]` | Output | Yes | Result of custom instruction |

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 requires input operands. If the custom instruction requires only a single input port, use `dataa`.

### Combinational Custom Instruction Timing

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 cycle.

**Figure 4: Combinational Custom Instruction Timing Diagram**



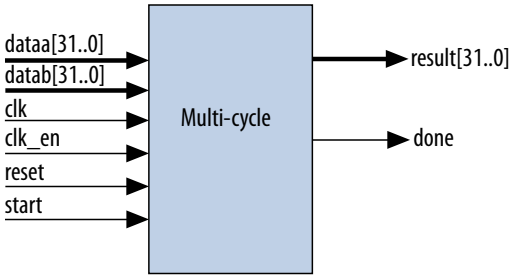

**Related Information**

**Combinational Custom Instruction Ports** on page 5
Block diagram showing the `dataa`, `datab`, and `result` ports

## Multicycle Custom Instructions

Multicycle (sequential) custom instructions consist of a logic block that requires two or more clock cycles to complete an operation.

**Figure 5: Multicycle Custom Instruction Block Diagram**




Multicycle custom instructions complete in either a fixed or variable number of clock cycles. For a custom instruction that completes in a fixed number of clock cycles, you specify the required number of clock cycles at system generation. For a custom instruction that requires a variable number of clock cycles, you instantiate the `start` and `done` ports. These ports participate in a handshaking scheme to determine when the custom instruction execution is complete.

A basic multicycle custom instruction block, with the required ports shown in "Custom Instruction Types", implements a single custom operation. This operation has a selection index determined when the instruction is instantiated in the system using Qsys.

You can further optimize multicycle custom instructions by implementing the extended internal register file, or by creating external interface custom instructions.

**Related Information**

- **Extended Custom Instructions** on page 8
- **Internal Register File Custom Instructions** on page 10
- **External Interface Custom Instructions** on page 11
- **Custom Instruction Types** on page 4
  List of standard custom instruction hardware ports, to be used as signal types

## Multicycle Custom Instruction Ports

**Table 3: Multicycle Custom Instruction Ports**

| Port Name | Direction | Required | Description |
|---|---|---|---|
| `clk` | Input | Yes | System clock |
| `clk_en` | Input | Yes | Clock enable |
| `reset` | Input | Yes | Synchronous reset |
| `start` | Input | No | Commands custom instruction logic to start execution |
| `done` | Output | No | Custom instruction logic indicates to the processor 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 of custom instruction |

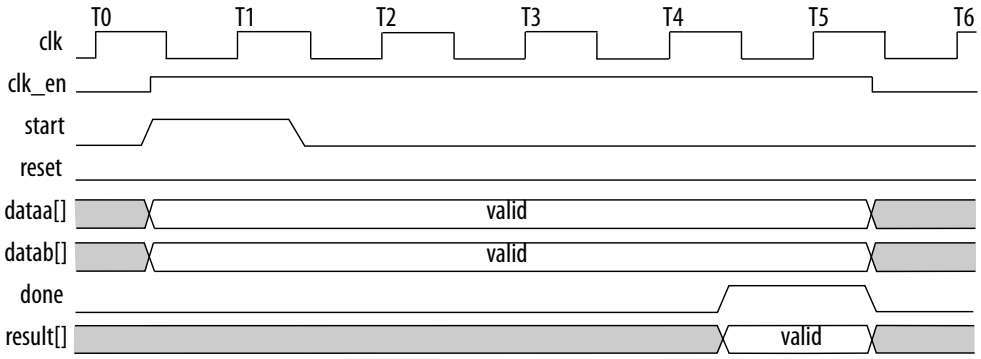
The `clk`, `clk_en`, and `reset` ports are required for multicycle custom instructions. The `start`, `done`, `dataa`, `datab`, and `result` ports are optional. Implement them only if the custom instruction requires them.

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.

### Multicycle Custom Instruction Timing

**Figure 6: Multicycle 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.

For a fixed length multicycle custom instruction, after the instruction starts, the processor waits the specified number of clock cycles, and then reads the value on the `result` signal. For an n-cycle operation, the custom logic block must present valid data on the $n^{th}$ rising edge after the custom instruction begins execution.

For a variable length multicycle custom instruction, the processor waits until the active high `done` signal is asserted. The processor reads the `result` port on the same clock edge on which `done` is asserted. The custom logic block must present data on the `result` port on the same clock cycle on which it asserts the `done` signal.

## Extended Custom Instructions

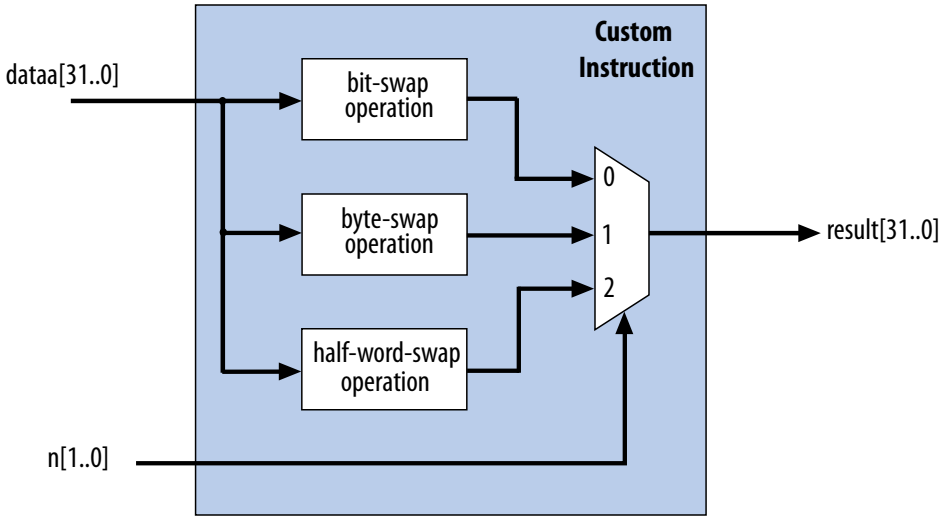
An extended custom instruction allows a single custom logic block to implement several different operations.

Extended custom instruction components occupy multiple select indices. The selection indices are determined when the custom instruction hardware block is instantiated in the system using Qsys.

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

The following block diagram shows an extended custom instruction with bit-swap, byte-swap, and half-word swap operations.

**Figure 7: Extended Custom Instruction with Swap Operations**




The custom instruction in the preceding figure performs swap operations on data received at the `dataa` port. The instruction hardware uses the two bit wide `n` port to select the output from a multiplexer, determining which result is presented to the `result` port.

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

Extended custom instructions can be combinational or multicycle custom instructions. To implement an extended custom instruction, 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.

An extended custom instruction block occupies several contiguous selection indices. When the block is instantiated, Qsys determines a base selection index. When the Nios II processor decodes a `custom` instruction, the custom hardware block's `n` port decodes the low-order bits of the selection index. Thus, the extension index extends the base index to produce the complete selection index.

For example, suppose the custom instruction block in **Figure 7** is instantiated in a Nios II system with a base selection index of 0x1C. In this case, individual swap operations are selected with the following selection indices:

- 0x1C—Bit swap
- 0x1D—Byte swap
- 0x1E—Half-word swap
- 0x1F—reserved

Therefore, if `n` is <*m*> bits wide, the extended custom instruction component occupies $2^{<m>}$ select indices.

For example, the custom instruction illustrated above occupies four indices, because `n` is two bits wide. Therefore, when this instruction is implemented in a Nios II system, 256 - 4 = 252 available indices remain.

**Related Information**

**Custom Instruction Assembly Language Interface** on page 14
Information about the custom instruction index

### Extended Custom Instruction Timing

All extended custom instruction port operations are identical to those for the combinational and multicycle custom instructions, with the exception of the `n` port.

The `n` port timing is the same as that of the `dataa` port. For example, for an extended variable multicycle custom instruction, the processor presents the extension index to the `n` port on the same rising edge of the clock at which `start` is asserted, and the `n` port remains stable during execution of the custom instruction.

The `n` port is not present in combinational and multicycle custom instructions.

## Internal Register File Custom Instructions

The Nios II processor allows custom instruction logic to access its own internal register file.

Internal register file access gives you the flexibility to specify whether 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 to the Nios II processor's register file.

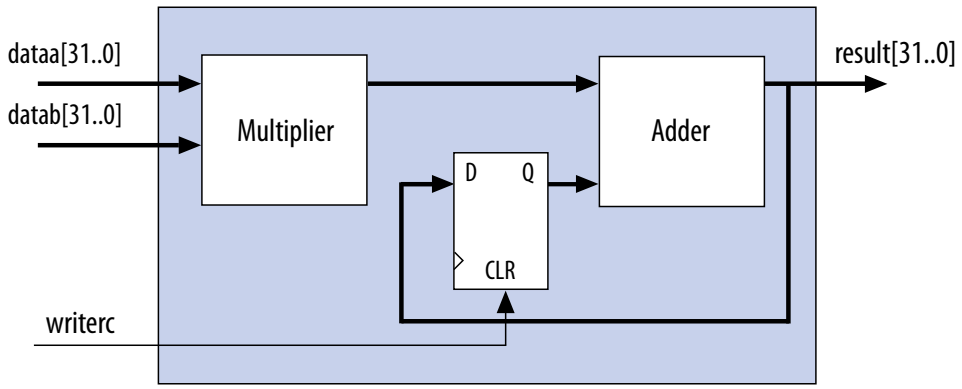
Custom instructions containing internal register files use `readra`, `readrb`, and `writerc` signals to determine if the custom instruction should use the internal register file or the `dataa`, `datab`, and `result` signals. Ports `a`, `b`, and `c` specify the internal registers from which to read or to which to write. For example, if `readra` is deasserted (specifying a read operation from the internal register), the `a` signal value provides the register number in the internal register file. Ports `a`, `b`, and `c` are five bits each, allowing you to address as many as 32 registers.

**Related Information**

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

### Internal Register File Custom Instruction Example

**Figure 8: Multiply-accumulate Custom Logic Block**




This example shows how a custom instruction can access the Nios II internal register file.

When writerc is deasserted, the Nios II processor ignores the value driven on the result port. The accumulated value is stored in an internal register. Alternatively, the processor can read the value on the 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 Ports

To access the Nios II internal register file, you must implement several custom instruction-specific ports.

The following table lists the internal register file custom instruction-specific optional ports. Use the optional ports only if the custom instruction requires them.

**Table 4: Internal Register File Custom Instruction Ports**

| Port Name | Direction | Required | Description |
|---|---|---|---|
| readra | Input | No | If readra is high, Nios II processor register a supplies dataa. If readra is low, custom instruction logic reads internal register a. |
| readrb | Input | No | If readrb is high, Nios II processor register b supplies datab. If readrb is low, custom instruction logic reads internal register b. |
| writerc | Input | No | If writerc is high, the Nios II processor writes the value on the result port to register c. If writerc is low, custom instruction logic writes to internal register c. |
| a[4:0] | Input | No | Custom instruction internal register number for data source A. |
| b[4:0] | Input | No | Custom instruction internal register number for data source B. |
| c[4:0] | Input | No | Custom instruction internal register number for data destination. |

The readra, readrb, writerc, a, b, and c ports behave similarly to dataa. When the custom instruction begins, the processor presents the new values of the readra, readrb, writerc, a, b, and c ports on the rising edge of the processor clock. All six of these ports remain stable during execution of the custom instructions.

To determine how to handle the 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 numbrs. When readra or readrb is asserted, the custom instruction logic ignores the corresponding a or b port, and receives data from the dataa or datab port. When writerc is asserted, the custom instruction logic ignores the c port and writes to the result port.

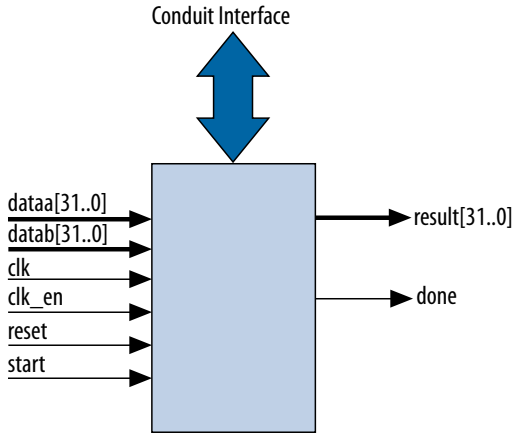
All other custom instruction port operations behave the same as for combinational and multicycle custom instructions.

## External Interface Custom Instructions

Nios II external interface custom instructions allow you to add an interface to communicate with logic outside of the processor's datapath.

At system generation, conduits propagate out to the top level of the Qsys system, where external logic can access the signals. By enabling custom instruction logic to access memory external to the processor, external interface custom instructions extend the capabilities of the custom instruction logic.

**Figure 9: Custom Instruction with External Interface**




Custom instruction logic can perform various tasks such as storing intermediate results or reading 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 with an external interface can feed data directly from the processor's register file to an external first-in first-out (FIFO) memory buffer.

# Software Interface

The Nios II custom instruction software interface abstracts logic implementation details from the application code.

During the build process the Nios II software build tools generate macros that allow easy access from application code to custom instructions.

## Custom Instruction Software Examples

These examples illustrate how the Nios II custom instruction software interface fits into your software code.

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

```
#define ALT_CI_BITSWAP_N 0x00
#define ALT_CI_BITSWAP(A) __builtin_custom_ini(ALT_CI_BITSWAP_N,(A))
```

In this example, `ALT_CI_BITWSWAP_N` is defined to be 0x0, which is the custom instruction's selection index. The `ALT_CI_BITSWAP(A)` macro accepts a single argument, abstracting out the selection index `ALT_CI_BITWSWAP_N`. The macro maps to a GNU Compiler Collection (GCC) Nios II built-in function.

The next example illustrates application code that uses the bit-swap custom instruction.

```
#include "system.h"

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

```
    int a_swap = 0;

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

The code in this example includes the **system.h** file to enable the application software to use the custom instruction macro definition. The example code declares two integers, `a` and `a_swap`. Integer `a` is passed as input to the bit swap custom instruction and the results are loaded in `a_swap`.

The example above illustrates how most applications use custom instructions. The macros defined by the Nios II software build tools use C integer types only. Occasionally, applications require input types other than integers. In those cases, you can use a custom instruction macro to process non-integer return values.

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

**Related Information**

**Custom Instruction Built-in Functions** on page 33
More information about the GCC built-in functions

## 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 a **system.h** file. However, by using built-in functions, software can use 32 bit non-integer types with custom instructions. Fifty-two built-in functions are available to accommodate the different combinations of supported types.

Built-in function names have the following format:

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

<*return type*> and <*parameter types*> represent the input and output types, encoded as follows:

- `i`—`int`
- `f`—`float`
- `p`—`void *`

The following example shows the prototype definitions for two built-in functions.

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

`n` is the selection index. The built-in function `__builtin_custom_nf()` accepts a `float` as an input, and does not return a value. The built-in function`__builtin_custom_fnp()` accepts a pointer as 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.

The following example shows user-defined custom instruction macros used in an application.

```
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()` accepts a `float` as an input parameter and does not return anything. The macro `UDEF_MACRO2()` accepts a pointer as an input parameter and returns a `float`. Lines 14 and 15 show code that uses the two user-defined macros.

**Related Information**

**Custom Instruction Built-in Functions** on page 33

# Custom Instruction Assembly Language Interface

The Nios II custom instructions are accessible in assembly code as well as C/C++.

## Custom Instruction Assembly Language Syntax

Nios II custom instructions use a standard assembly language syntax:

`custom` <*selection index*>, <*Destination*>, <*Source A*>, <*Source B*>

- <*selection index*>—The 8-bit number that selects the particular custom instruction
- <*Destination*>—Identifies the register where the result from the `result` port (if any) will be placed
- <*Source A*>—Identifies the register that provides the first input argument from the `dataa` port (if any)
- <*Source B*>—Identifies the register that provides the first input argument from the `datab` port (if any)

You designate registers in one of two formats, depending on whether you want the custom instruction to use a Nios II register or an internal register:

- `r`<*i*>—Nios II register <*i*>
- `c`<*i*>—Custom register <*i*> (internal to the custom instruction component)

The use of `r` or `c` controls the `readra`, `readrb`, and `writerc` fields in the the custom instruction word.

Custom registers are only available with internal register file custom instructions.

**Related Information**

**Custom Instruction Word Format** on page 15
Detailed information about instruction fields and register file selection

## Custom Instruction Assembly Language Examples

These examples demonstrate the syntax for custom instruction assembly language calls.

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

The example above shows a call to a custom instruction with selection index 0. The input to the instruction is the current contents of the Nios II processor registers `r7` and `r8`, and the results are stored in the Nios II processor register `r6`.

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

The example above shows a call to a custom instruction with selection index 3. The input to the instruction is the current contents of the Nios II processor register `r2` and the custom register `c4`, and the results are stored in custom register `c1`.

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

The example above shows a call to a custom instruction with selection index 4. The input to the instruction is the current contents of the custom register `c9` and the Nios II processor register `r2`, and the results are stored in Nios II processor register `r6`.

**Related Information**

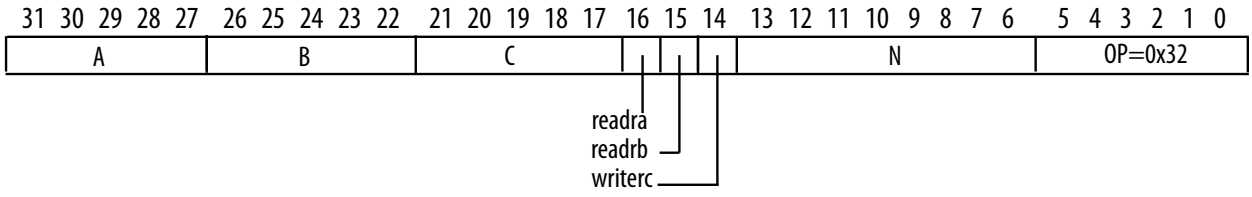
**custom**
Refer to "custom" in the *Instruction Set Reference* chapter of the *Nios II Processor Reference Guide* for more information about the binary format of custom instructions.

## Custom Instruction Word Format

Custom instructions are R-type instructions.

The instruction word specifies the 8-bit custom instruction selection index and register usage.

**Figure 10: Custom Instruction Word Format**




Custom instruction fields:

**Table 5: Custom Instruction Fields**

| Field Name | Purpose | Corresponding Signal |
|---|---|---|
| `A` | Register address of input operand A | |
| `B` | Register address of input operand B | |
| `C` | Register address of output operand C | |
| `readra` | Register file selector for input operand A | `readra` |
| `readrb` | Register file selector for input operand B | `readrb` |
| `writerc` | Register file selector for ouput operand C | `writerc` |
| `N` | Custom instruction select index (optionally includes an extension index) | |

| Field Name | Purpose | Corresponding Signal |
|---|---|---|
| OP | custom opcode, 0x32 | n/a |

The register file selectors determine whether the custom instruction component accesses Nios II processor registers or custom registers, as follows:

**Table 6: Register File Selection**

| Register File Selector Value | Register File |
|---|---|
| 0 | Custom instruction component internal register file |
| 1 | Nios II processor register file |

**Related Information**

- **R-Type**
  Refer to "R-Type" in the *Instruction Set Reference* chapter of the *Nios II Processor Reference Guide* for information about R-type instructions.
- **custom**
  Refer to "custom" in the *Instruction Set Reference* chapter of the *Nios II Processor Reference Guide* for more information about the binary format of custom instructions.

### Select Index Field (N)

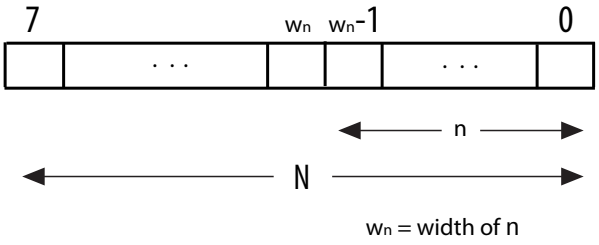
The custom instruction N field, bits 13:6, is the custom instruction select index. The select index determine which custom instruction executes.

The Nios II processor supports up to 256 distinct custom instructions through the custom opcode. A custom instruction component can implement a single instruction, or multiple instructions.

In the case of a simple (non-extended) custom instruction, the select index is a simple 8-bit value, assigned to the custom instruction block when it is instantiated in Qsys.

Components that implement multiple instructions possess an n port, as described in "Extended Custom Instructions". The n port implements an extension index, which is a subfield of the select index, as shown in the following figure.

**Figure 11: Select Index Format**




**Note:** Do not confuse N, the selection index field of the custom instruction, with n, the extension index port. Although n can be 8 bits wide, it generally corresponds to the low-order bits of N.

# Implementing a Nios II Custom Instruction in Qsys

You use the Qsys component editor to instantiate a Nios II custom instruction based on your custom hardware.

The Qsys component editor enables you to create new Qsys components, including Nios II custom instructions.

**Related Information**

- **Creating Qsys Components**
  For detailed information about the Qsys component editor, refer to "Creating Qsys Components" in the *Quartus Prime Pro Edition Handbook Volume 1: Design and Synthesis*.
- **Creating Qsys Components**
  For detailed information about the Qsys component editor, refer to "Creating Qsys Components" in the *Quartus Prime Standard Edition Handbook Volume 1: Design and Synthesis*.

## Design Example: Cyclic Redundancy Check

The cyclic redundancy check (CRC) algorithm is a useful example of a Nios II custom instruction.

The 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. Because the operations are carried out concurrently, the execution is much faster in hardware.

The CRC design files demonstrate the steps to implement an extended multicycle Nios II custom instruction.

**Related Information**

**Nios II Custom Instruction Design Example**
The design files are available for you to download from the Nios II Custom Instruction Design Example web page.

## Implementing Custom Instruction Hardware in Qsys

Implementing a Nios II custom instruction involves using the custom instruction tool flow.

Implementing a Nios II custom instruction hardware entails the following tasks:

1. Opening the component editor
2. Specify the custom instruction component type
3. Displaying the custom instruction block symbol
4. Adding the HDL files
5. Configuring the custom instruction parameter type
6. Setting up the custom instruction interfaces
7. Configuring the custom instruction signal type
8. Saving and adding the custom instruction
9. Generating the system and compiling in the Quartus® Prime software

## Setting up the Design Environment for the Design Example

Before you start the design example, you must set up the design environment to accommodate the custom instruction implementation process.

To set up the design example environment, follow these steps:

1. Download the **ug_custom_instruction_files.zip** file from the Nios II Custom Instruction Design Example web page.
2. Open the **ug_custom_instruction_files.zip** file and extract all the files to a new directory.
3. Follow the instructions in the Quartus Prime Project Setup section in the **readme_qsys.txt** file in the extracted design files. The instructions direct you to determine a <*project_dir*> working directory for the project and to open the design example project in the Quartus Prime software.

**Related Information**

**Nios II Custom Instruction Design Example**
The design files are available for you to download from the Nios II Custom Instruction Design Example web page.

## Opening the Component Editor

After you finish setting up the design environment, you can open Qsys and the component editor.

Before performing this task, you must perform the steps in "Setting up the Design Environment for the Design Example". After performing these steps, you have a Quartus Prime project located in the <*project_dir*> directory and open in the Quartus Prime software.

To open the component editor, follow these steps:

1. To open Qsys, on the Tools menu, click **Qsys**.
2. In Qsys, on the File menu, click **Open**.
3. Browse to the <*project_dir*> directory if necessary, select the **.qsys** file, and click **Open**.
4. On the Qsys **Component Library** tab, click **New**. The component editor appears, displaying the **Introduction** tab.

**Related Information**

**Setting up the Design Environment for the Design Example** on page 18
Instructions for setting up the design environment

## Specifying the Custom Instruction Component Type

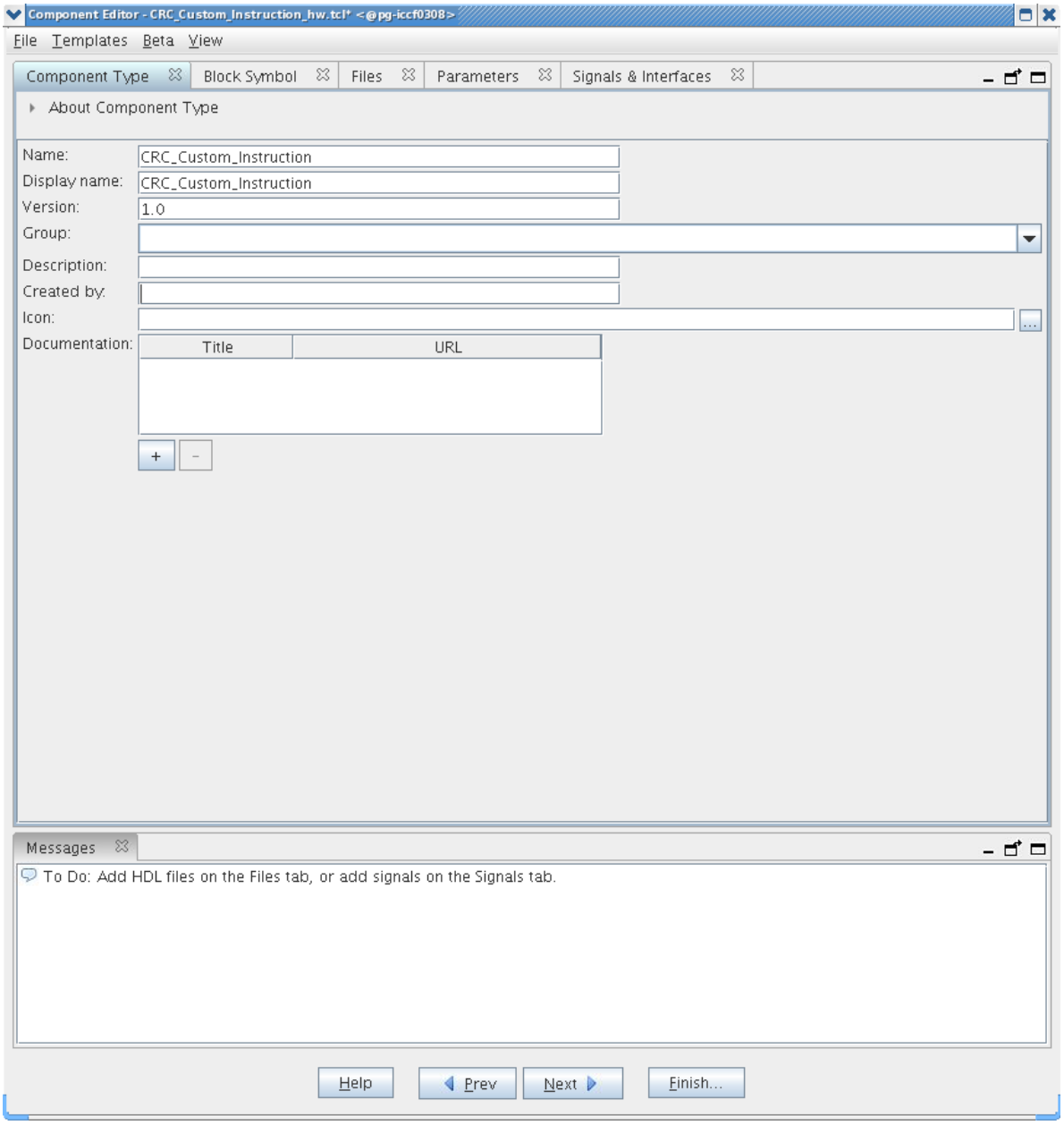
To specify the custom instruction component type, you specify a name, a display name, a version, and optionally a group, description (recommended), creator, and icon. These steps help define the **_hw.tcl** file for the new custom component.

First, make sure that the component editor displays the **Component Type** tab.

To specify the initial details in the custom instruction parameter editor, follow these steps:

1. For **Name** and for **Display Name**, type `CRC`.
2. For **Version**, type `1.0`.
3. Leave the **Group** field blank.
4. Optionally, set the **Description**, **Created by**, and **Icon** fields as you prefer.

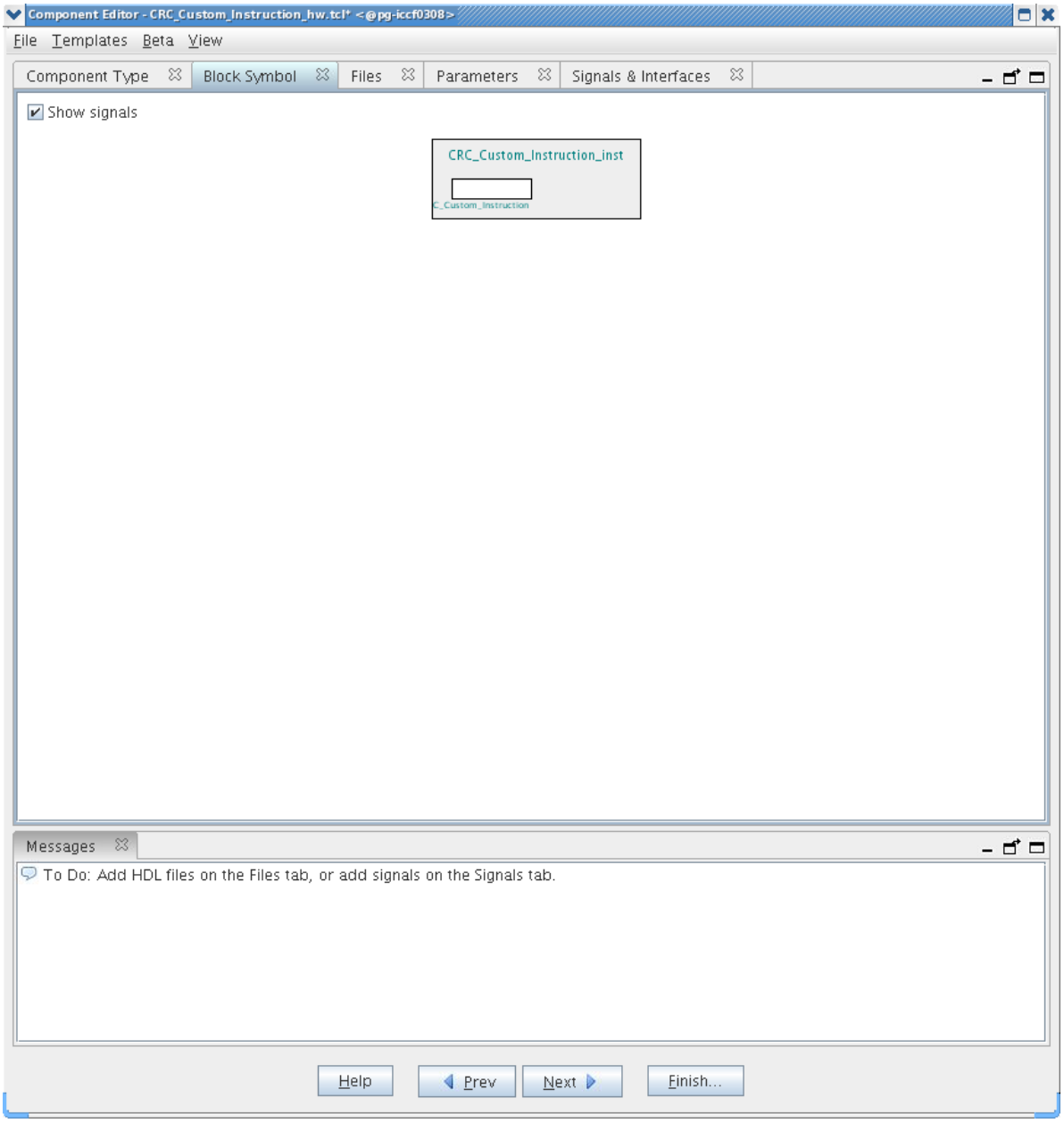
**Figure 12: Setting Custom Instruction Name and Version**



## Displaying the Block Symbol

Click **Next** to display the custom component in the **Block Symbol** tab.

**Figure 13: Viewing the Custom Instruction as a Block Symbol**



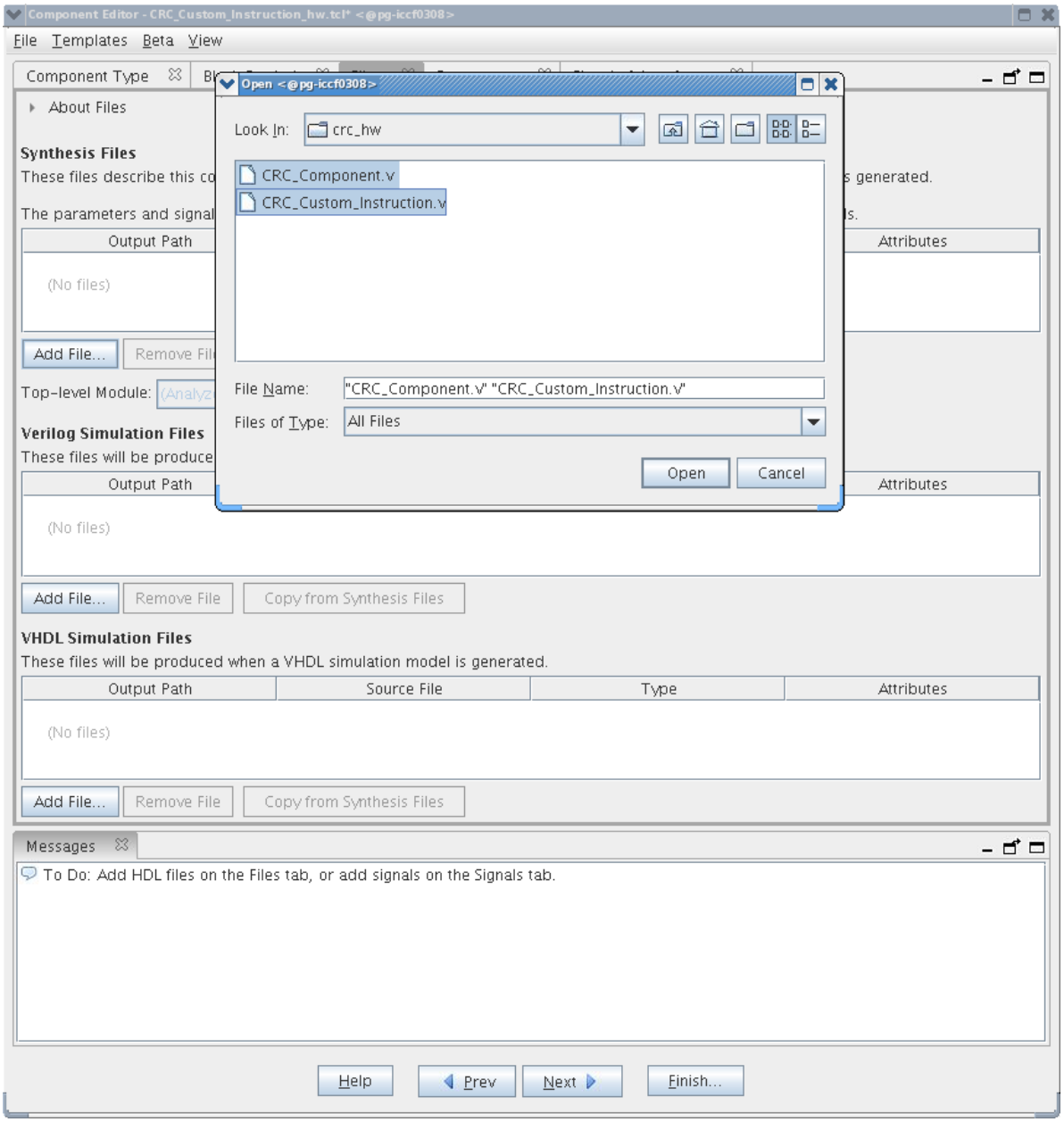

## Adding the Custom Instruction HDL Files

To specify the synthesis HDL files for your custom instruction, you browse to the HDL logic definition files in the design example.

To specify the synthesis files, follow these steps:

1. Click **Next** to display the **Files** tab.
2. Under **Synthesis Files**, click **Add Files**.
3. Browse to <*project_dir*>**/crc_hw**, the location of the HDL files for this design example.
4. Select the **CRC_Custom_Instruction.v** and **CRC_Component.v** files and click **Open**.

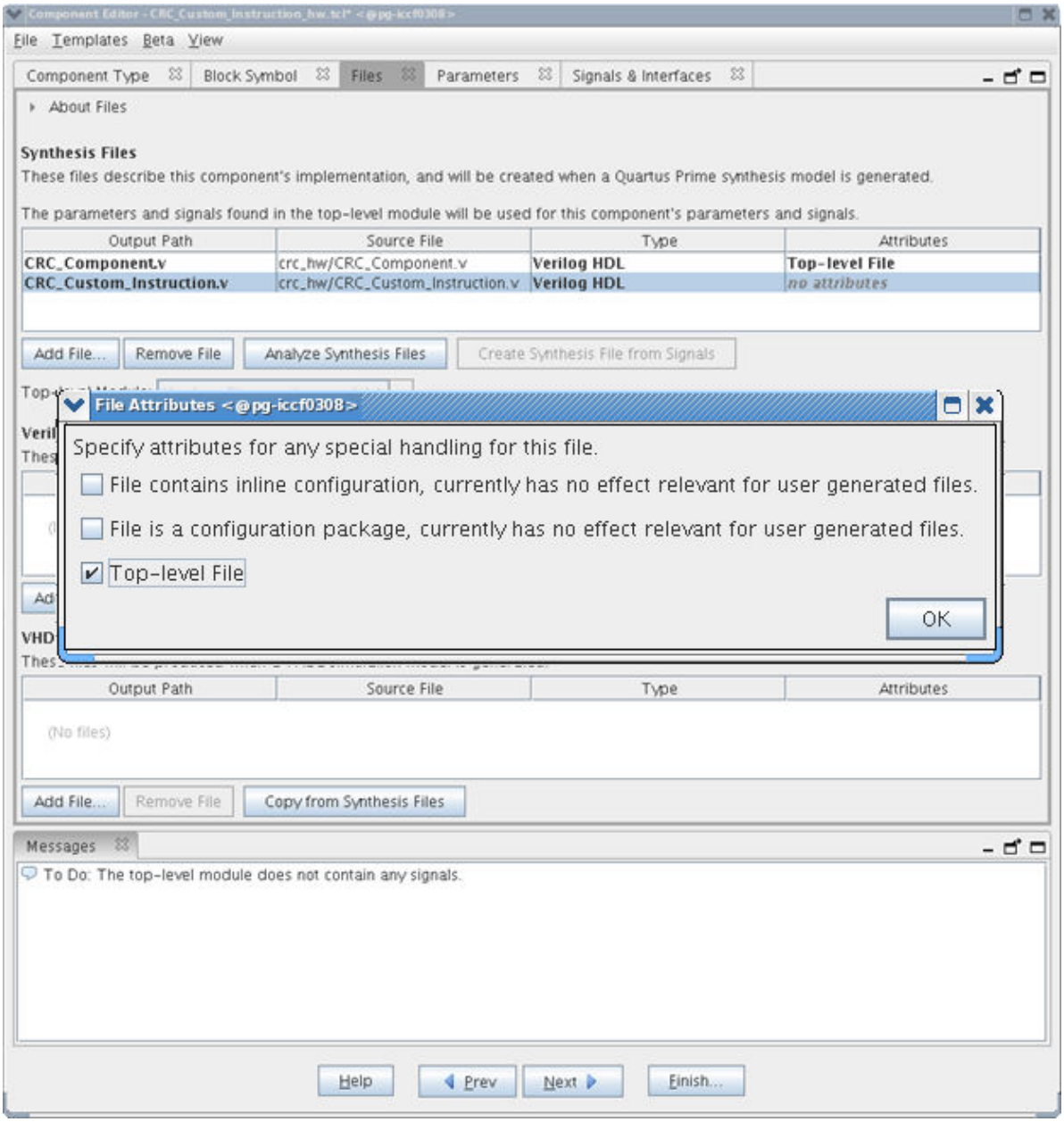
**Figure 14: Browsing to Custom Instruction HDL Files**




**Note:** The Quartus Prime Analysis and Synthesis program checks the design for errors when you add the files. Confirm that no error message appears.

5. Open the **File Attributes** dialog box by double-clicking the **Attributes** column in the **CRC_Custom_Instruction.v** line.

**Figure 15: File Attributes Dialog Box**




6. In the **File Attributes** dialog box, turn on the **Top-level File** attribute, as shown in the figure above. This attribute indicates that **CRC_Custom_Instruction.v** is the top-level HDL file for this custom instruction.
7. Click **OK**.

**Note:** The Quartus Prime Analysis and Synthesis program checks the design for errors when you select a top-level file. Confirm that no error message appears.

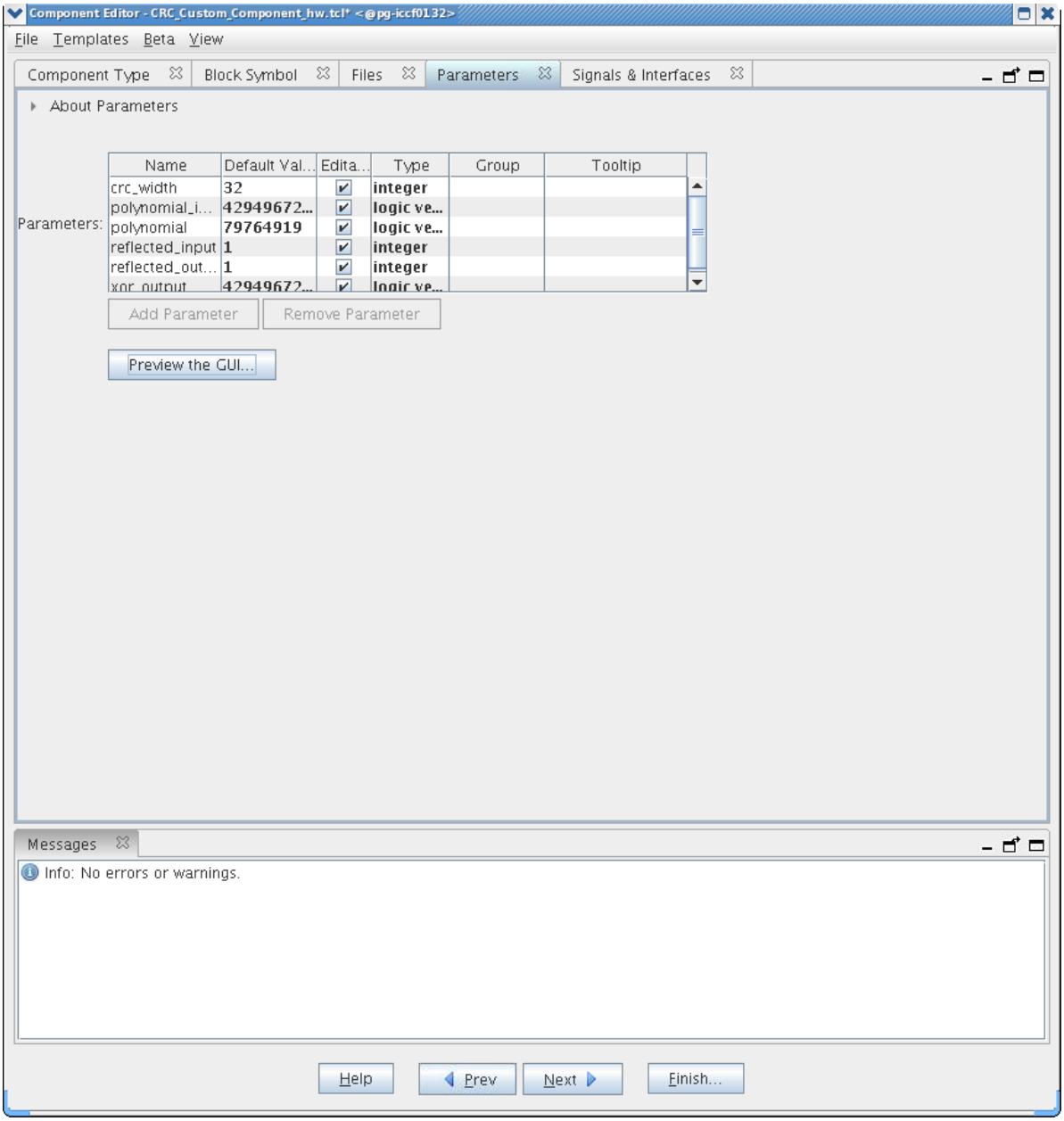
8. Click **Analyze Synthesis Files** to synthesize the top-level file.
9. To simulate the system with the ModelSim™ simulator, you can add your simulation files under **Verilog Simulation Files** or **VHDL Simulation Files** in the in the **Files** tab.

## Configuring the Custom Instruction Parameter Type

To configure the custom instruction parameter type, follow these steps:

1. Click **Next** to display the **Parameters** tab. The parameters in the **.v** files are displayed.

**Figure 16: Custom Instruction Parameters**




The **Editable** checkbox next to each parameter indicates whether the parameter will appear in the custom component's parameter editor. By default, all parameters are editable.

2. To remove a parameter from the custom instruction parameter editor, you can turn off **Editable** next to the parameter. For the CRC example, you can leave all parameters editable.

   When **Editable** is off, the user cannot see or control the parameter, and it is set to the value in the **Default Value** column. When **Editable** is on, the user can control the parameter value, and it defaults to the value in the **Default Value** column.
3. To see a preview of the custom component's parameter editor, you can click **Preview the GUI**.

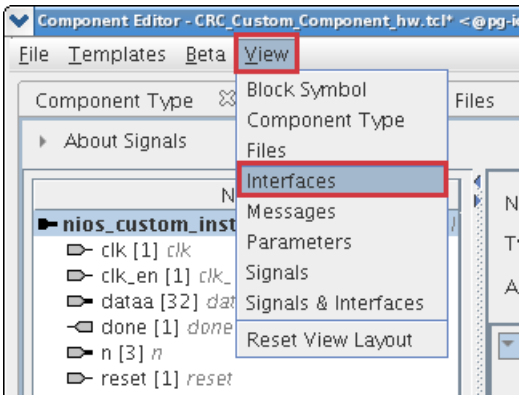
## Setting Up the Custom Instruction Interfaces

To set up the custom instruction interfaces, you use the **Interfaces** tab.

To set up the custom instruction interfaces, follow these steps:

1. In the View menu, click **Interfaces** to display the **Interfaces** tab.

**Figure 17: Opening the Interfaces Tab**




2. If the **Remove Interfaces With No Signals** button is active, click it.
3. Ensure that a single interface remains, with **Name** set to the name in the Signals tab. For the design example, maintain the interface name **nios_custom_instruction_slave.**
4. Ensure the **Type** for this interface is **Custom Instruction Slave**.
5. For **Clock Cycles**, type 0. This is the correct value for a variable multicycle type custom instruction, such as the CRC module in the design example. For other designs, use the correct number of clock cycles for your custom instruction logic.
6. For **Operands**, type `1`, because the CRC custom instruction has one operand. For other designs, type the number of operands used by your custom instruction.

**Note:** If you rename an interface by changing the value in the **Name** field, the **Signals** tab **Interface** column value changes automatically. The value shown in the block diagram updates when you change tabs and return to the **Interfaces** tab.

**Note:** If the interface includes a `done` signal and a `clk` signal, the component editor infers that the interface is a variable multicycle type custom instruction interface, and sets the value of **Clock Cycles** to `0`.

### Specifying Additional Interfaces

You can specify additional interfaces in the **Interfaces** tab.

You can specify additional interfaces if your custom instruction logic requires special interfaces, either to the Avalon-MM fabric or outside the Qsys system. The design example does not require additional interfaces.

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

The following instructions provide the information you need if a custom instruction in your own design requires additional interfaces. You do not need these steps if you are implementing the design example.

To specify additional interfaces on the **Interfaces** tab, follow these steps:

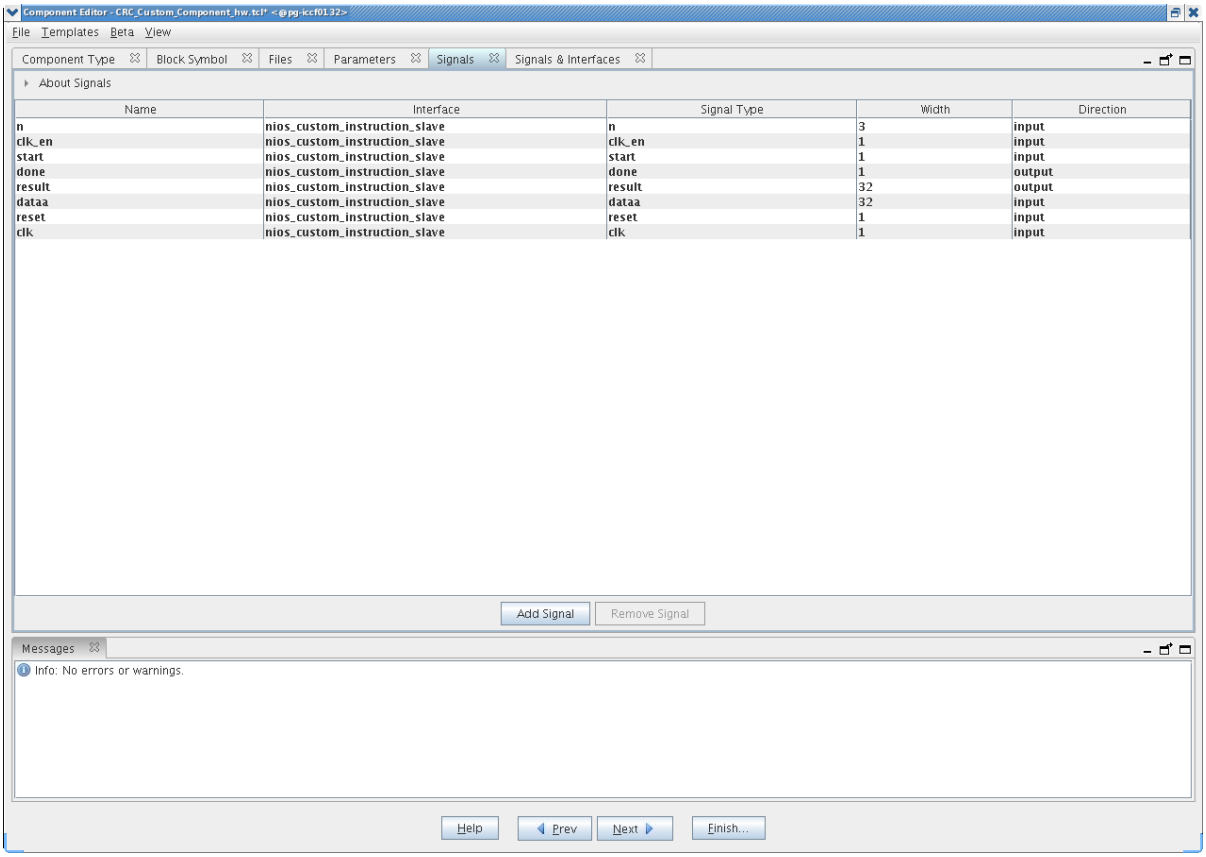
1. Click **Add Interface**. The new interface has **Custom Instruction Slave** interface type by default.
2. For **Type**, select the desired interface type.
3. Set the parameters for the newly created interface according to your system requirements.

## Configuring the Custom Instruction Signal Type

To configure the custom instruction signal type, follow these steps:

1. In the **View** menu, click **Signals** to open the **Signals** tab.

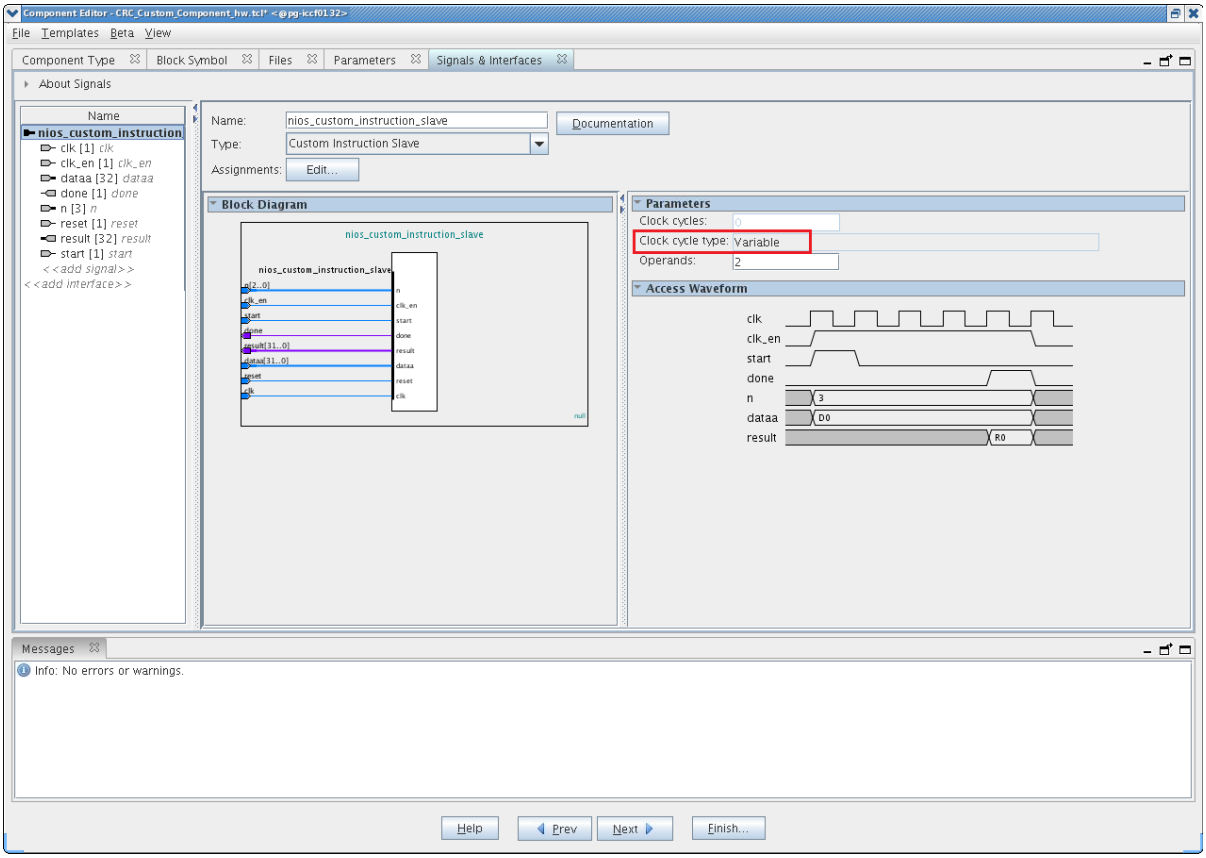
**Figure 18: Custom Instruction Signal Types**




2. For each signal in the list, follow these steps:
   a. Select the signal name.
   b. In the **Interface** column, select the name of the interface to which you want to assign the signal.

      In the design example, select **nios_custom_instruction_slave** for all signals. These selections ensure that the signals appear together on a single interface, and that the interface corresponds to the design example files in the **crc_hw** folder.
   c. In the **Signal Type** column, select one of the standard hardware ports listed in "Custom Instruction Types". In the design example, each signal must be mapped to the signal type of the same name.
3. Open the **Signals and Interfaces** tab.

**Figure 19: Signals and Interfaces**




The parameters for **Clock Cycle Type** automatically change to "Variable" because the design example builds a variable multicycle type custom instruction. For other designs, you enter the correct clock cycle type for your custom instruction design:

- "Variable" for a variable multicycle type custom instruction
- "Multicycle" for a fixed multicycle type custom instruction
- "Combinatorial" for a combinational type custom instruction.

If the interface does not include a `clk` signal, the component editor automatically infers that the interface is a combinational type interface. If the interface includes a `clk` signal, the component editor automatically infers that the interface is a multicycle interface. If the interface does not include a `done` signal, the component editor infers that the interface is a fixed multicycle type interface. If the interface includes a done signal, the component editor infers that the interface is a variable multicycle type interface.

**Related Information**

**Custom Instruction Types** on page 4
List of standard custom instruction hardware ports, to be used as signal types

## Saving and Adding the Custom Instruction

When your custom instruction is fully defined, you save it and add it to your Qsys system.

To save the custom instruction and add it to your Nios II processor, follow these steps:

1. Click **Finish**. A dialog box prompts you to save your changes before exiting.
2. Click **Yes, Save**. The new custom instruction appears in the Qsys Component Library.
3. In the Qsys Component Library, under Library, select **CRC**, the new custom instruction you created in the design example.
4. Click **Add** to add the new instruction to the Qsys system.
   Qsys automatically assigns an unused selection index to the new custom instruction. You can see this index in the **System Contents** tab, in the **Base** column, in the form "Opcode <*N*>". <*N*> is represented as a decimal number. The selection index is exported to **system.h** when you generate the system.
5. In the **Connections** panel, connect the new **CRC_0** component's **nios_custom_instruction_slave** interface to the **cpu** component's **custom_instruction_master** interface.
6. Optional: You can change the custom instruction's selection index in the **System Contents** tab. In the **Base** column across from the custom instruction slave, click on "Opcode <*N*>", and type the desired selection index in decimal.

### Generating the System and Compiling in the Quartus Prime Software

After you add the custom instruction logic to the system, you can generate the Qsys system and compile it in the Quartus Prime software.

To generate the system and compile, follow these steps:

1. In Qsys, on the **Generation** tab, turn on **Create HDL design files for synthesis**.
2. Click **Generate**. System generation may take several seconds to complete.
3. After system generation completes, on the File menu, click **Exit**.
4. In the Quartus Prime software, on the Project menu, click **Add/Remove Files in Project**.
5. Ensure that the **.qip** file in the **synthesis** subdirectory is added to the project.
6. On the Processing menu, click **Start Compilation**.

#### Related Information

- **Creating a System with Qsys (Pro Edition)**
  For detailed information about the Qsys component editor, refer to "Creating a System with Qsys" in the *Quartus Prime Pro Edition Handbook Volume 1: Design and Synthesis*.
- **Creating a System with Qsys (Standard Edition)**
  For detailed information about the Qsys component editor, refer to "Creating a System with Qsys" in the *Quartus Prime Standard Edition Handbook Volume 1: Design and Synthesis*.

## Accessing the Custom Instruction Example from Software

Next you create and build a new software project using the Nios II software build flow, and run the software that accesses the custom instruction.

The downloadable design files include the software source files. The following table lists the CRC application software source files and their corresponding descriptions.

**Table 7: CRC Application Software Source Files**

| File Name | Description |
|---|---|
| **crc_main.c** | 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. |

| File Name | Description |
|---|---|
| **crc.c** | Software CRC algorithm run by the Nios II processor. |
| **crc.h** | Header file for **crc.c**. |
| **ci_crc.c** | Program that accesses CRC custom instruction. |
| **ci_crc.h** | Header file for **ci_crc.c**. |

To run the application software, you must create an Executable and Linking Format File (**.elf**) first. To create the **.elf** file, follow the instructions in the "Nios II Software Build Flow" section in the **readme_qsys.txt** file in the extracted design files.

The application program runs three implementations of the CRC algorithm on the same pseudo-random input data: an unoptimized software implementation, an optimized software implementation, and the custom instruction CRC. The program calculates the processing time and throughput for each of the versions, to demonstrate the improved efficiency of a custom instruction compared to a software implementation.

## Custom Instruction Example Software Output

The following example shows the output from the application program run on a Cyclone V E FPGA Development Kit with a 5CEFA7F31I7N speed grade device. This example was created using the Quartus Prime software v15.1 and Nios II Embedded Design Suite (EDS) v15.1.

The output shows that the custom instruction CRC is 68 times faster than the unoptimized CRC calculated purely in software and is 39 times faster than the optimized version of the software CRC. The results you see using a different target device and board may vary depending on the memory characteristics of the board and the clock speed of the device, but these ratios are representative.

### Output of the CRC Design Example Software Run on a Cyclone V E FPGA Development Kit using the Quartus Prime Software v15.1.

```
********************************************************************************
Comparison between software and custom instruction CRC32
********************************************************************************
System specification
--------------------
System clock speed = 50 MHz
Number of buffer locations = 32
Size of each buffer = 256 bytes

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 = 34 ms
Optimized software CRC = 19 ms
Custom instruction CRC = 00 ms

Processing throughput for each implementation
---------------------------------------------
Software CRC = 2978 Mbps
Optimized software CRC = 32768 Mbps
Custom instruction CRC = 949 Mbps

Speedup ratio
-------------
Custom instruction CRC vs software CRC = 68
Custom instruction CRC vs optimized software CRC = 39
Optimized software CRC vs software CRC = 1
```

## Using the User-defined Custom Instruction Macro

The design example software uses a user-defined macro to access the CRC custom instruction.

The following example shows the macro that is defined in the **ci_crc.c** file.

```
#define CRC_CI_MACRO(n, A) \
__builtin_custom_ini(ALT_CI_CRC_CUSTOM_COMPONENT_0_N + (n & 0x7), (A))
```

This macro accepts a single `int` type input operand and returns an `int` type value. The CRC custom instruction has extended type; the `n` value in the macro `CRC_CI_MACRO()` indicates the operation to be performed by the custom instruction.

`ALT_CI_CRC_CUSTOM_COMPONENT_0_N` is the custom instruction selection index for the first instruction in the component. `ALT_CI_CRC_CUSTOM_COMPONENT_0_N` is added to the value of `n` to calculate the selection index for a specific instruction. The `n` value is masked because the `n` port of the custom instruction has only three bits.

To initialize the custom instruction, for example, you can add the initialization code in the following example to your application software.

```
/* Initialize the custom instruction CRC to the initial remainder value: */
CRC_CI_MACRO (0,0);
```

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

The examples above demonstrate that 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, and vary the extension index value, `n`. However, the macro definition and usage must be consistent with the port declarations of the custom instruction. For example, if you define the macro to return an `int` value, the custom instruction must have a `result` port.

**Related Information**

- **Implementing a Nios II Custom Instruction in Qsys** on page 17
  Step-by-step instructions for implementing a custom instruction

- **Software Interface** on page 12
  Information about the custom instruction software interface

# Custom Instruction Templates

The Nios II EDS includes VHDL and Verilog HDL custom instruction wrapper file templates that you can reference when writing custom instructions in VHDL and Verilog HDL.

Full sets of template files are available in the following directories:

- *<nios2eds installation directory>***/examples/verilog/custom_instruction_templates**
- *<nios2eds installation directory>***/examples/vhdl/custom_instruction_templates**

## VHDL Custom Instruction Template

The Nios II EDS includes a VHDL custom instruction template for an internal register type custom instruction.

```
-- 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 multicycle or extended multicycle)
  signal reset: in std_logic;
      -- CPU master asynchronous active high reset
      --  (required for multicycle or extended multicycle)
  signal clk_en: in std_logic;
      -- Clock-qualifier (required for multicycle or extended multicycle)
  signal start: in std_logic;
      -- Active high signal used to specify that inputs are valid
      --  (required for multicycle or extended multicycle)
  signal done: out std_logic;
      -- Active high signal used to notify the CPU that result is valid
      --  (required for variable multicycle or extended variable multicycle)
  signal n: in std_logic_vector(7 downto 0);
      -- N-field selector (required for extended);
      -- Modify width to match the number of unique operations in the 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 Custom Instruction Template Example

The Nios II EDS includes a Verilog HDL custom instruction template for an internal register type custom instruction.

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

module custominstruction(
  clk,      // CPU system clock (required for multicycle or extended multicycle)
  reset,    // CPU master asynchronous active high reset
            //  (required for multicycle or extended multicycle)
  clk_en,   // Clock-qualifier (required for multicycle or extended multicycle)
  start,    // Active high signal used to specify that inputs are valid
            //  (required for multicycle or extended multicycle)
  done,     // Active high signal used to notify the CPU that result is valid
            //  (required for variable multicycle or extended variable multicycle)
  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
input  clk;
input  reset;
input  clk_en;
input  start;
input  [7:0]  n;  // modify width to match the number of unique operations
                  // in the instruction
input  [4:0]  a;
input  [4:0]  b;
input  [4:0]  c;
input  readra;
input  readrb;
input  writerc;
input  [31:0]  dataa;
input  [31:0]  datab;

//OUTPUTS
output  done;
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
```

# Custom Instruction Built-in Functions

The Nios II GCC compiler, `nios2-elf-gcc`, is customized with built-in functions to support custom instructions.

Nios II custom instruction built-in functions have the following return types:

- `void`
- `int`
- `float`
- `void*`

**Related Information**

**https://gcc.gnu.org**
More information about GCC built-in functions

## Built-in Functions with No Return Value

The following built-in functions in the Nios II GCC compiler have no return value. `n` represents the custom instruction selection index, and `dataa` and `datab` represent the input arguments, if any.

- `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 that Return a Value of Type Int

The following built-in functions in the Nios II GCC compiler return a value of type `int`. `n` represents the custom instruction selection index, and `dataa` and `datab` represent the input arguments, if any.

- `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 that Return a Value of Type Float

The following built-in functions in the Nios II GCC compiler return a value of type `float`. `n` represents the custom instruction selection index, and `dataa` and `datab` represent the input arguments, if any.

- `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 that Return a Pointer Value

The following built-in functions in the Nios II GCC compiler return a pointer value. `n` represents the custom instruction selection index, and `dataa` and `datab` represent the input arguments, if any.

- `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);`

# Floating Point Custom Instructions

The Nios II EDS includes floating point custom instructions to access the Altera floating point hardware soft IP cores.

The Quartus Prime software offers two predefined floating point units:

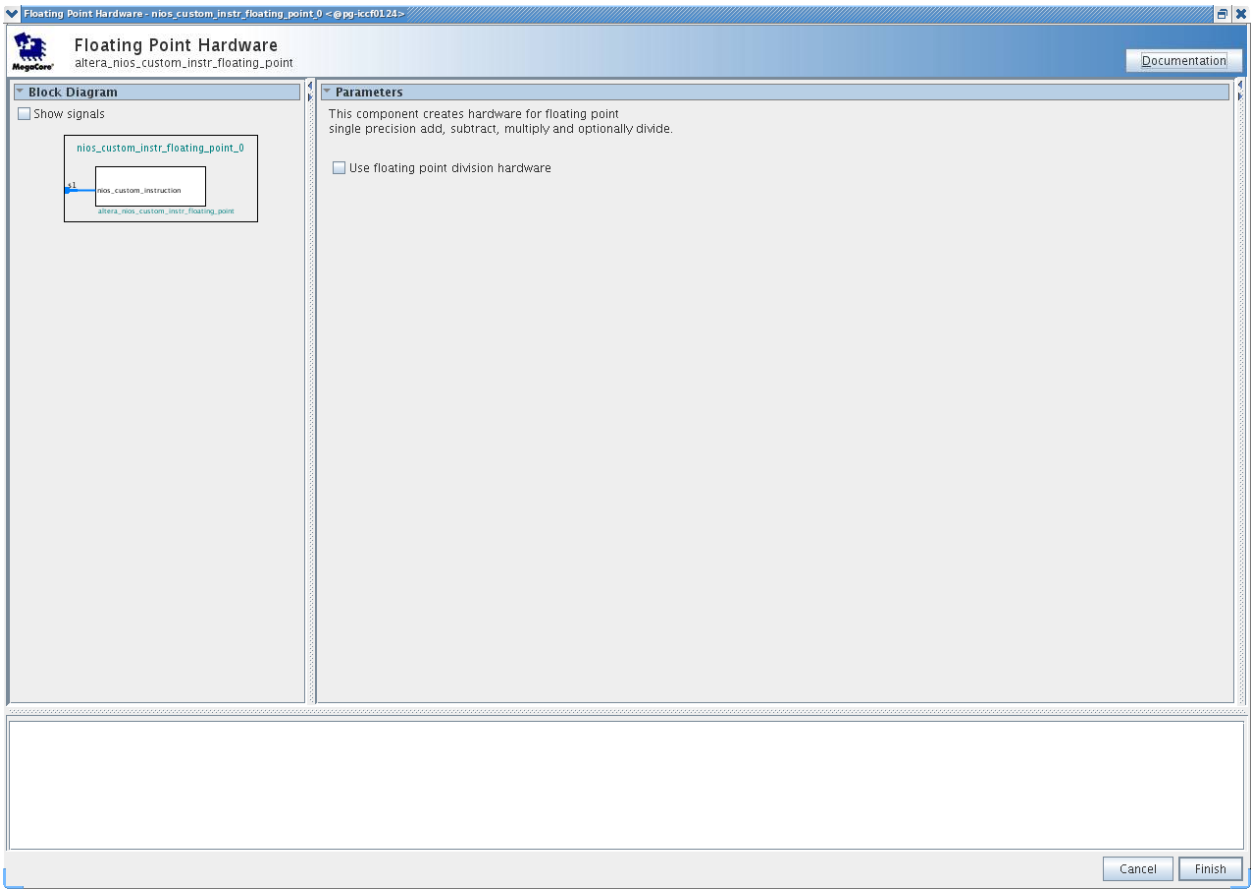
- Floating Point Hardware—Supports single precision floating point add, subtract, multiply, and divide instructions.
- Floating Point Hardware 2—Supports floating point add, subtract, multiply, divide, square root, floating point conversion and comparison, and several other useful operations, with improved performance and resource usage.

## Floating Point Hardware Component

The Floating Point Hardware component supports single-precision floating point hardware add, subtract, multiply, and optional divide instructions.

When you add the Floating Point Hardware component to your system, a parameter editor appears, and you can turn on an option to include a floating point divider.

**Figure 20: Floating Point Hardware Custom Instruction Parameter Editor**




**Note:** The selection indices must be as shown in "Supporting Double-Precision Constants". Qsys enforces this rule when you instantiate the component.

When you add a floating point custom instruction to your system, the Nios II software build tools add flags to the `nios2-elf-gcc` command line which are determined by specific custom instructions used by the software. These flags select the appropriate version of newlib to support the floating point operations that are in use, omitting code supporting unused operations.

The software build tools add one of the following compiler flags:

- `-mcustom-fpu-cfg=60-1`--system does not include a custom instruction floating point divider
- `-mcustom-fpu-cfg=60-2`--system includes a custom instruction floating point divider

The `-mcustom-fpu-cfg` option bundles several command-line options into one and allows the GCC linker to choose the precompiled newlib that supports the selected floating point operations.

**Related Information**

**Creating a Custom Version of newlib**
How to compile newlib for the Nios II Gen2 processor

## Supporting Double-Precision Constants

The `-mcustom-fpu-cfg` flag forces the use of single-precision constants. To allow double-precision constants, you must modify the `nios2-elf-gcc` command in your makefiles. You must remove the `-mcustom-fpu-cfg` flag and replace it with individual compiler flags.

To enable double-precision floating point constants, in the `nios2-elf-gcc` command line, replace the `-mcustom-fpu-cfg` option with the individual options shown in the following table.

**Table 8: Individual Floating Point Compiler Options**

| `-mcustom-fpu-cfg` Option | Individual Options |
|---|---|
| `-mcustom-fpu-cfg=60-1` | • `-mcustom-fmuls=252`<br>• `-mcustom-fadds=253`<br>• `-mcustom-fsubs=254` |
| `-mcustom-fpu-cfg=60-2` | • `-mcustom-fmuls=252`<br>• `-mcustom-fadds=253`<br>• `-mcustom-fsubs=254`<br>• `-mcustom-fdivs=255` |

For example, if you instantiated the floating point handware without the divider, the `nios2-elf-gcc` command in your makefiles resembles the following:

```
nios2-elf-gcc -mcustom-fpu-cfg=60-1 ...
```

To enable double-precision floating point constants, your modified command resembles the following:

```
nios2-elf-gcc -mcustom-fmuls=252 -mcustom-fadds=253 -mcustom-fsubs=254 ...
```

**Note:** Replace the `-mcustom-fpu-cfg` flag only if required. Replacing the `-mcustom-fpu-cfg` flag disables floating point custom instruction support in newlib functions, forcing the library to use the slower, software-emulated instructions.

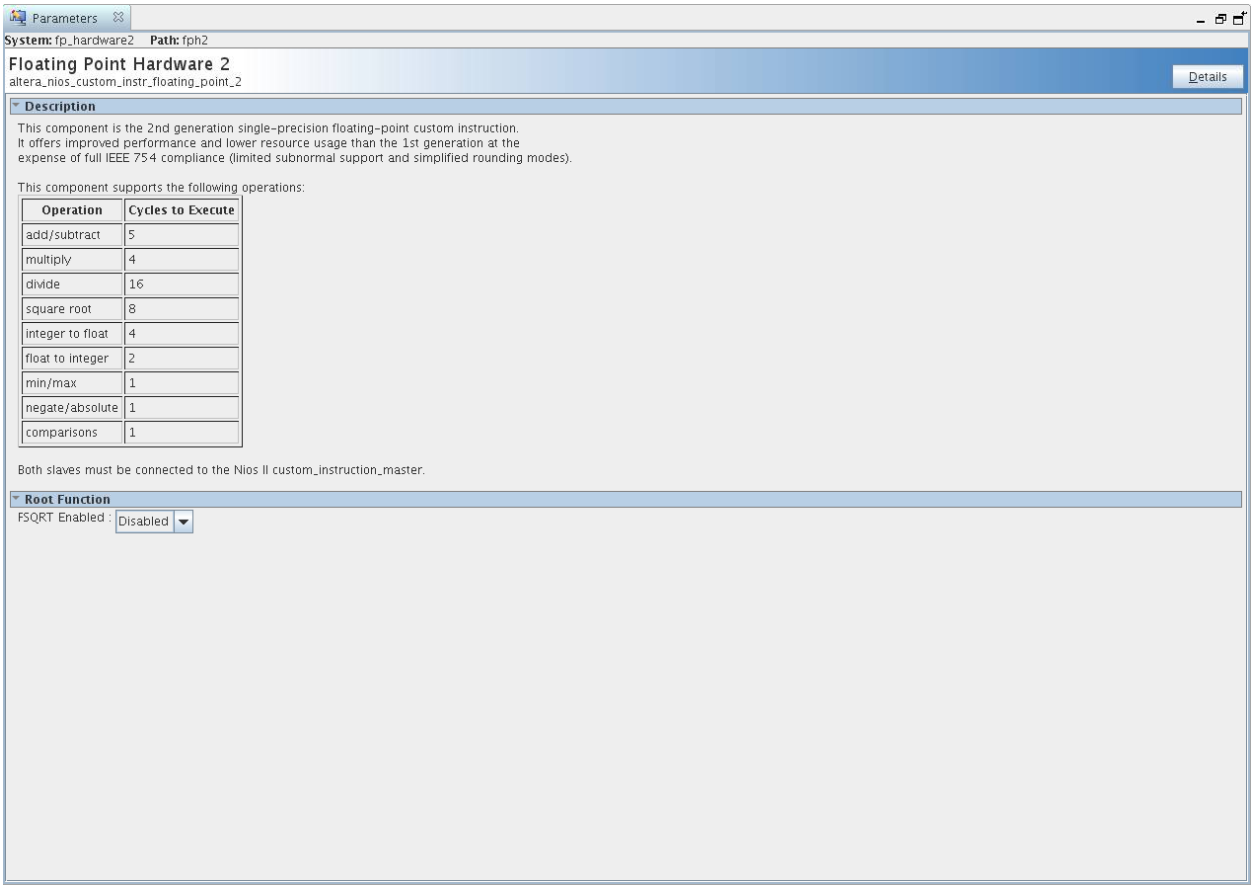
# Floating Point Hardware 2 Component

The Floating Point Hardware 2 component implements add, subtract, multiply, divide, square root, integer/floating point conversions, min, max, arithmetic invert, absolute value, and comparison instructions.

The Floating Point Hardware 2 component is the second generation of the floating point hardware, with better performance and lower resource usage than the Floating Point Hardware component.

When you add the Floating Point Hardware 2 component to your system, a parameter editor appears showing the operations that it supports and the number of cycles to execute each operation. You can choose to enable or disable the square root instruction.

**Figure 21: Floating Point Hardware 2 Custom Instruction Parameter Editor**




**Note:** The selection indices must be as shown the table below. Qsys enforces this rule when you instantiate the component.

With Floating Point Hardware 2, Altera recommends that you compile your newlib library from source code. In the `nios2-elf-gcc` command line, specify the hardware floating point operations that your code needs. This method allows newlib to take advantage of all Floating Point Hardware 2 functions required by your code, without incurring overhead for unnecessary functions.

The `-mcustom-fpu-cfg` option is available and compatible with the Floating Point Hardware 2 component, but it is not recommended.

**Table 9: Individual nios2-elf-gcc Command Line Options**

| Function | Command Line Option |
|---|---|
| Add | `-mcustom-fadds=253` |

| Function | Command Line Option |
|---|---|
| Subtract | `-mcustom-fsubs=254` |
| Multiply | `-mcustom-fmuls=252` |
| Divide | `-mcustom-fdivs=255` |
| Square root | `-mcustom-fsqrts=251` |
| Convert integer to floating point | `-mcustom-floatis=250` |
| Convert floating point to integer | `-mcustom-fixsi=249` |
| Minimum | `-mcustom-fmins=233` |
| Maximum | `-mcustom-fmaxs=232` |
| Arithmetic invert | `-mcustom-fnegs=225` |
| Absolute value | `-mcustom-fabss=224` |
| Compare a < b | `-mcustom-fcmplts=231` |
| Compare a <= b | `-mcustom-fcmples=230` |
| Compare a > b | `-mcustom-fcmpgts=229` |
| Compare a >= b | `-mcustom-fcmpges=228` |
| Compare a == b | `-mcustom-fcmpeqs=227` |
| Compare a != b | `-mcustom-fcmpnes=226` |

**Related Information**

- **Floating Point Hardware Component** on page 35
- **Creating a Custom Version of newlib**
  How to compile newlib for the Nios II Gen2 processor

## Document Revision History

The following table shows the revision history for this document.

| Date | Version | Changes |
|---|---|---|
| November 2015 | 2015.11.02 | • Updated for Quartus Prime software v15.1.<br>• Updated for Floating Point Custom Instructions 2<br>• Remove SOPC Builder system integration tool flow.<br>• Name change: the Quartus II software is now known as the Quartus Prime software |
| January 2011 | 2.0 | • Updated for Quartus II software v10.1.<br>• Updated for new Qsys system integration tool flow.<br>• Updated with formatting changes. |

| Date | Version | Changes |
|---|---|---|
| May 2008 | 1.5 | • Add new tutorial design.<br>• Describe new custom instruction import flow.<br>• Minor corrections to terminology and usage. |
| May 2007 | 1.4 | Add title and core version number to page footers. |
| May 2007 | 1.3 | • Describe new component editor import flow.<br>• Remove tutorial design.<br>• Minor corrections to terminology and usage. |
| December 2004 | 1.2 | Updates for Nios II processor vresion 1.1. |
| September 2004 | 1.1 | Updates for Nios II processor version 1.01. |
| May 2004 | 1.0 | Initial release. |