Using Nios II Floating-Point Custom Instructions

Tutorial

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

TU-N2FLTNGPNT-2.0

Contents

1. Floating-Point Custom Instructions

This tutorial guides you through the basics elements of floating-point custom instructions for use with the Nios® II embedded processor. Nios II floating-point custom instructions accelerate arithmetic functions executed on `float` variable types. The tutorial is a good starting point if you are considering floating-point custom instructions for inclusion in your own project. In this tutorial you add the floating-point custom instructions to a Nios II example design, and create a software program to demonstrate floating-point performance.

The tutorial covers the following topics:

- How to add floating-point custom instructions to a Nios II processor
- How to use floating-point custom instructions in a C program, and how the custom instructions work with the Nios II Embedded Design Suite (EDS)
- The advantages and disadvantages of floating-point custom instructions, and how best to use them in your own system

About the Floating-Point Custom Instructions

The floating-point custom instructions, optionally available on the Nios II processor, implement single-precision, floating-point arithmetic operations. You can use the custom instructions to accelerate floating-point operations in your Nios II C/C++ application program. The basic set of floating-point custom instructions is available on every Nios II core implementation and includes single precision floating-point addition, subtraction, and multiplication. Floating-point division is available as an extension to the basic instruction set.

Table 1–1 lists approximate acceleration factors that the floating-point custom instructions provide:

Table 1–1. Sample Floating-Point Custom Instruction Acceleration Factors *(Note 1)*

Target Device	Addition	Subtraction	Multiplication	Division
EP3C120	20 times	18 times	17 times	12 times
EP3SL150	18 times	19 times	12 times	13 times

Note to Table 1–1:

(1) These figures show typical speed increases over the equivalent software implementation. For each target device, these results were obtained using a Nios II /f core processor executing code from on-chip memory. You might see different acceleration results, depending on your hardware design and target device and on the details of your software application.

When the floating-point custom instructions are present in your target hardware, the Nios II Software Build Tools (SBT) for Eclipse compiles your code to use the custom instructions for floating-point operations, including the four primitive arithmetic operations (addition, subtraction, multiplication and division) and the ANSI C math library. Table 1–2 on page 1–8 lists the ANSI C math functions.

The floating point custom instructions substantially comply with the IEEE 754-1985 floating point standard. For details, refer to “Floating Point Instructions” in the *Processor Architecture* chapter of the *Nios II Processor Reference Handbook*.

Preparing Your System

The following sections provide information you need before you begin the tutorial.

Prerequisites

To make effective use of this tutorial, you should be familiar with the following topics:

- Defining and generating Nios II hardware systems with SOPC Builder
- Compiling Nios II hardware systems with the Quartus® II development software
- Creating, compiling, and running Nios II software projects

To learn about defining, generating, and compiling Nios II systems, refer to the *Nios II Hardware Development Tutorial*. To learn about Nios II software projects, refer to the *Getting Started with the Graphical User Interface* chapter of the *Nios II Software Developer’s Handbook*.

Software and Hardware Requirements

This tutorial requires the following software and hardware:

- Quartus II development software version 9.1 or later, installed on a Windows or Linux computer
- Nios II EDS version 9.1 or later
- A JTAG download cable compatible with your target hardware, for example, a USB-Blaster™ cable
- A development board that includes the following devices:
  - An Altera FPGA large enough to support the Nios II processor core, hold the target design, and leave enough unused logic elements to support the floating-point custom instructions. For information about device resource usage, refer to Table 1–3 on page 1–9.
  - An oscillator that drives a constant clock frequency to an FPGA pin. The maximum frequency depends on the speed grade of the FPGA.
  - A JTAG connection to the FPGA that provides a programming interface and communication link to the Nios II system. This connection can be either a dedicated 10-pin JTAG header for an Altera USB-Blaster download cable (revision B or higher) or a USB connection with USB-Blaster circuitry embedded on the board.

- A Nios II target design that includes the following components:
  - Nios II processor
  - JTAG UART
  - Performance counter with at least 2 simultaneously-measured sections
  - 128 KB of on-chip or external memory
  - System timer
  - System ID peripheral

Altera provides several working Nios II reference designs which you can use as a starting point for your own designs. After installing the Nios II EDS, refer to the <*Nios II EDS install path*>**/examples/verilog** or the <*Nios II EDS install path*>**/examples/vhdl** directory. Demonstration applications are also available in newer development kit installations.

Tutorial Software Files

The tutorial software files are available in a **.zip** file next to the link to this document on the Literature: Nios II Processor page of the Altera website. Download and unzip the file in a temporary directory. The **.zip** file includes the following files:

- **floating_point.c**—main program
- **floating_point.h**—global definitions
- **floating_point_CI.c**—functions to exercise the floating-point custom instructions
- **floating_point_SW.c**—functions to exercise the software-implemented floating-point operations

Building and Programming the Hardware

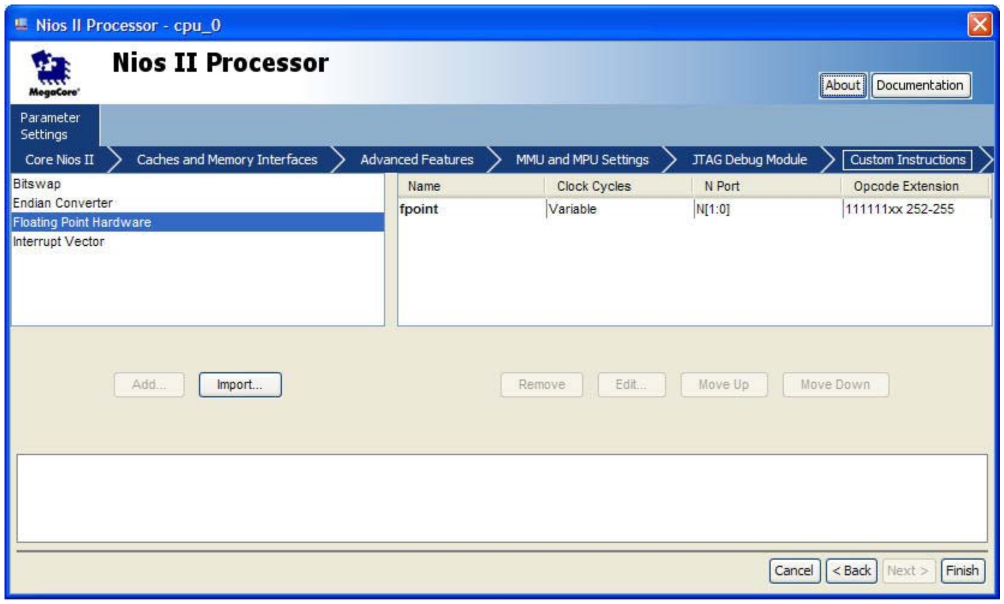
Perform the following steps to add the floating-point custom instructions to the Nios II processor in your target design:

1. Start the Quartus II development software and open a working copy of your target design.
2. Start SOPC Builder.
3. On the **System Contents** tab, double-click the Nios II processor component. The Nios II MegaWizard interface appears.
4. Select the **Custom Instructions** page.
5. Select **Floating Point Hardware** from the list and click **Add**. The **Nios II Floating Point Hardware** dialog box appears.
6. Turn on **Use floating point division hardware**.

   The floating-point division hardware is optional. For a discussion regarding the advantages and disadvantages of using the floating-point division hardware, refer to "Moving On to Your Own System" on page 1–7.

7. Click **Finish** to exit the **Nios II Floating Point Hardware** dialog box. Figure 1–1 shows the **Custom Instructions** page of the Nios II MegaWizard interface with the floating-point hardware inserted.

Figure 1–1. Hardware Setup Dialog Box



Bitswap
Endian Converter
Floating Point Hardware
Interrupt Vector

Name	Clock Cycles	N Port	Opcode Extension
fpoint	Variable	N[1:0]	111111xx 252-255

For further information about adding the floating-point custom instructions, refer to the *Instantiating the Nios II Processor in SOPC Builder* chapter of the *Nios II Processor Reference Handbook*.

8. Click **Finish** to exit the Nios II MegaWizard interface.
9. Generate the HDL for your SOPC Builder system. When the generation process is complete, exit SOPC Builder.
10. Compile the Quartus II project.
11. Program your target hardware with the resulting SRAM Object File (**.sof**).

Building and Running the Software

This section steps you through creating, building, running, and analyzing your software project.

Creating the Software Project

Perform the following steps to create the software project:

1. Start the Nios II SBT for Eclipse.
2. Create a new **Nios II Application and BSP from Template** based on the **Blank Project** template. Under **Target hardware information**, browse to locate the SOPC Information File (**.sopcinfo**) that you generated in "Building and Programming the Hardware" on page 1–3.

3. Import the tutorial software files described in "Tutorial Software Files" on page 1–3 into your Nios II application project. The easiest way to do this is to select the files in an application such as Windows Explorer, and drag them into the Nios II application project folder in the Project Explorer view of the Nios II SBT for Eclipse.
4. Adjust the compiler optimization settings to meet your needs. Access the settings through the **Properties** dialog boxes for your Nios II application and Nios II BSP projects.

Building and Running the Software and Analyzing the Results

Perform the following steps to analyze the results of the software project:

1. Build the software project. The Nios II SBT for Eclipse detects the presence of the floating-point custom instructions at build time, and uses them for all single precision floating-point arithmetic.
2. Run the software on your Nios II target design. The program runs four tests, one each for the add, subtract, multiply, and divide operations. In each test, the program carries out the floating-point operation on 1000 pairs of random operands. It executes both the floating-point custom instruction and the equivalent software implementation. Using the performance counter component, the tutorial software compares the hardware and software execution times.

   The following program output shows the results:

```
--Performance Counter Report--
Total Time: 0.01222420 seconds  (611210 clock-cycles)
+---------------+-----+-----------+---------------+-----------+
| Section       |  %  | Time (sec)|  Time (clocks)|Occurrences|
+---------------+-----+-----------+---------------+-----------+
|FP CI ADD      | 2.29|    0.00030|          14000|       1000|
+---------------+-----+-----------+---------------+-----------+
|FP SW ADD      | 50.2|    0.00610|         306640|       1000|
+---------------+-----+-----------+---------------+-----------+

--Performance Counter Report--
Total Time: 0.00987798 seconds  (493899 clock-cycles)
+---------------+-----+-----------+---------------+-----------+
| Section       |  %  | Time (sec)|  Time (clocks)|Occurrences|
+---------------+-----+-----------+---------------+-----------+
|FP CI SUBTRACT | 2.83|    0.00028|          14000|       1000|
+---------------+-----+-----------+---------------+-----------+
|FP SW SUBTRACT | 50.8|    0.00502|         250975|       1000|
+---------------+-----+-----------+---------------+-----------+
--Performance Counter Report--
Total Time: 0.0110131 seconds  (550654 clock-cycles)
+---------------+-----+-----------+---------------+-----------+
| Section       |  %  | Time (sec)|  Time (clocks)|Occurrences|
+---------------+-----+-----------+---------------+-----------+
|FP CI MULTIPLY | 2.18|    0.00024|          12000|       1000|
+---------------+-----+-----------+---------------+-----------+
|FP SW MULTIPLY |   59|    0.00650|         325076|       1000|
+---------------+-----+-----------+---------------+-----------+

--Performance Counter Report--
Total Time: 0.0142152 seconds  (710758 clock-cycles)
+---------------+-----+-----------+---------------+-----------+
| Section       |  %  | Time (sec)|  Time (clocks)|Occurrences|
+---------------+-----+-----------+---------------+-----------+
|FP CI DIVIDE   |  4.5|    0.00064|          32000|       1000|
+---------------+-----+-----------+---------------+-----------+
|FP SW DIVIDE   | 67.8|    0.00963|         481698|       1000|
+---------------+-----+-----------+---------------+-----------+
```

3. Analyze the results report for each test. In each report, the `FP CI <instruction>` entry lists the performance of the custom instruction, and the `FP SW <instruction>` entry lists the performance of the software implementation. The `Time (sec)` and `Time (clock)` columns represent the aggregate time spent executing the floating-point operations, in seconds and in Nios II clock cycles. `Total Time` represents the duration of the test, expressed both in seconds and in Nios II clock cycles. The `%` column represents the time spent executing the floating-point operation, as a percentage of the test total.

   ☞ You might have different speed results, depending on your target hardware and on the actual values of the random operands.

The tutorial software uses the Nios II performance counter component to collect timing information on the floating-point operations. For more information, refer to the *Performance Counter Core* chapter in volume 5 of the *Quartus II Handbook*.

Software Implementation

The tutorial software uses `#pragma` directives to compare hardware and software implementations of the floating-point instructions.

The following `#pragmas` direct the Nios II compiler to ignore the floating-point instructions and generate software implementations:

- `#pragma no_custom_fadds`—forces software implementation of floating-point add

- `#pragma no_custom_fsubs`—forces software implementation of floating-point subtract
- `#pragma no_custom_fmuls`—forces software implementation of floating-point multiply
- `#pragma no_custom_fdivs`—forces software implementation of floating-point divide

The scope of these #pragmas is the entire C file.

For more information, refer to "Floating Point Instructions" in the *Processor Architecture* chapter of the *Nios II Processor Reference Handbook*.

Moving On to Your Own System

Congratulations! You have successfully created a Nios II system that uses the floating-point custom instructions. Through this tutorial, you have familiarized yourself with the following steps for integrating the floating-point custom instructions into a Nios II system:

- Modifying and generating Nios II system hardware in SOPC Builder
- Compiling the Quartus II project
- Creating a new project in the Nios II SBT for Eclipse
- Compiling the project
- Running the software on the target hardware

This section helps you determine how to use the floating-point custom instructions in your own project.

Assessing Your Floating-Point Optimization Needs

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

- Have you identified your performance bottlenecks? Make sure your performance issues are caused by floating-point arithmetic before you try to fix them with floating-point acceleration.

  For detailed information about Nios II performance profiling, refer to *AN391: Profiling Nios II Systems*.

- Can you use integer arithmetic? While the floating-point custom instructions are faster than software-implemented floating-point, they are slower than integer arithmetic. A common integer technique is to represent numerical values with an implicit scaling factor. As a simple example, if you are calculating milliamperes, you might represent your values internally as microamperes.

- Are you taking full advantage of compiler optimization? You can increase the Nios II compiler optimization level through the **Properties** dialog box of your Nios II application and BSP projects.

  For information, refer to "Reducing Code Footprint" in the *Developing Programs Using the Hardware Abstraction Layer* chapter of the *Nios II Software Developer's Handbook.*

- Have you hand-optimized your mathematical operations? Numerical analysis textbooks offer simple, effective techniques for performing accurate calculations with the minimum number of floating-point operations.

If you have followed these suggestions, and you need further acceleration, the floating-point custom instructions are probably an appropriate solution.

Floating-Point Divide Considerations

The floating-point division hardware requires more resources than the other instructions, so you might opt to omit it if your Nios II application does not make heavy use of floating-point division.

In some cases, you can rewrite your code to minimize or even eliminate divide operations. For example, if your algorithm requires division by a constant value, you can precalculate its inverse and use a multiply operation in the speed-critical section of your code.

Table 1–2 indicates which math library functions use floating-point, and of those, which use floating-point division. If a function uses floating-point, it runs faster with floating-point hardware. If a function uses floating-point division, it runs even faster with floating-point division hardware.

Table 1–2. Math Library Floating-Point Usage

Math Function	Uses Floating-Point	Uses Floating-Point Division	Math Function	Uses Floating-Point	Uses Floating-Point Division
acos()	Yes	Yes	frexp()	Yes	
asin()	Yes	Yes	ldexp()	Yes	
atan()	Yes	Yes	log()	Yes	Yes
atan2()	Yes	Yes	log10()	Yes	Yes
cos()	Yes		modf()	Yes	
cosh()	Yes	Yes	pow()	Yes	Yes
sin()	Yes		sqrt()	Yes	Yes
sinh()	Yes	Yes	ceil()	Yes	
tan()	Yes	Yes	fabs()		
tanh()	Yes	Yes	floor()	Yes	
exp()	Yes	Yes	fmod()	Yes	Yes

When you omit the floating-point divide instruction, the Nios II SBT for Eclipse implements floating-point division in software.

For information about selecting the floating-point division hardware with the Nios II MegaWizard interface, refer to the *Instantiating the Nios II Processor in SOPC Builder* chapter of the *Nios II Processor Reference Handbook*.

Simulation

You can use the floating-point custom instructions with the ModelSim hardware simulator.

Device Resource Usage

The floating-point custom instructions are available on all Altera devices that support the Nios II processor. Table 1–3 shows approximate resource usage in each supported device.

If the target device includes on-chip multiplier elements, the floating-point hardware incorporates them as needed. If there are no on-chip multiplier elements, the floating-point custom instructions are implemented entirely with general-purpose logic elements.

Table 1–3. Approximate Device Resource Usage

Target Device Family	**Logic Elements (LE) or Adaptive Look-up Tables (ALUT)**		**Multiplier Elements** *(1)*
	Without Divide	**With Divide**	
Cyclone III	1500 LEs	7200 LEs	7
Stratix III	750 ALUTs	4200 ALUTs	4

Note to Table 1–3:

(1) In Cyclone III devices, the multiplier element is an embedded multiplier 9-bit element. In Stratix III devices, the multiplier element is a DSP 18-bit element.

Depending on the Quartus II routing and fitting, resource usage in your project might differ considerably from the values shown in Table 1–3.

Document Revision History

The following table shows the revision history for this document.

Date	Version	Changes
February 2010	2.0	Revised for Nios II Software Build Tools for Eclipse.
May 2006	1.0	Initial release.

How to Contact Altera

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

Contact (1)	Contact Method	Address
Technical support	Website	www.altera.com/support
Technical training	Website	www.altera.com/training
	Email	custrain@altera.com
Product literature	Website	www.altera.com/literature
Non-technical support (General)	Email	nacomp@altera.com
(Software Licensing)	Email	authorization@altera.com

Note to Table:

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

Typographic Conventions

The following table shows the typographic conventions this document uses.

Visual Cue	Meaning
Bold Type with Initial Capital Letters	Indicates command names, dialog box titles, dialog box options, and other GUI labels. For example, **Save As** dialog box. For GUI elements, capitalization matches the GUI.
bold type	Indicates directory names, project names, disk drive names, file names, file name extensions, software utility names, and GUI labels. For example, **\qdesigns** directory, **d:** drive, and **chiptrip.gdf** file.
Italic Type with Initial Capital Letters	Indicates document titles. For example, *AN 519: Stratix IV Design Guidelines*.
Italic type	Indicates variables. For example, $n + 1$. Variable names are enclosed in angle brackets (< >). For example, <*file name*> and <*project name*>**.pof** file.
Initial Capital Letters	Indicates keyboard keys and menu names. For example, the Delete key and the Options menu.

Visual Cue	Meaning
"Subheading Title"	Quotation marks indicate references to sections within a document and titles of Quartus II Help topics. For example, "Typographic Conventions."
`Courier type`	Indicates signal, port, register, bit, block, and primitive names. For example, `data1`, `tdi`, and `input`. The suffix `n` denotes an active-low signal. For example, `resetn`. Indicates command line commands and anything that must be typed exactly as it appears. For example, `c:\qdesigns\tutorial\chiptrip.gdf`. Also indicates sections of an actual file, such as a Report File, references to parts of files (for example, the AHDL keyword `SUBDESIGN`), and logic function names (for example, `TRI`).
1., 2., 3., and a., b., c., and so on	Numbered steps indicate a list of items when the sequence of the items is important, such as the steps listed in a procedure.
■ ■ ■	Bullets indicate a list of items when the sequence of the items is not important.
☞	The hand points to information that requires special attention.
⚠ CAUTION	A caution calls attention to a condition or possible situation that can damage or destroy the product or your work.
⚠ WARNING	A warning calls attention to a condition or possible situation that can cause you injury.
↵	The angled arrow instructs you to press the Enter key.
👣	The feet direct you to more information about a particular topic.