Nios II Floating Point Hardware 2 Component User Guide

2016.05.03

ug-1175

Subscribe

Send Feedback

Introduction

The Floating Point Hardware (FPH1)[1] provides substantial performance improvement over floating-point software emulation by providing custom instruction implementations of single-precision add, sub, multiply, and divide operations. Software implements all other floating-point operations (including double-precision operations) by emulating FPH1 functionality.

The Floating Point Hardware 2 (FPH2)[2] achieves even higher levels of performance by providing lower cycle count implementations of add, sub, multiply, and divide operations, and by providing custom instruction implementations of additional floating-point operations.

You must be familiar with the following items to fully understand this document:

- FPH1
- Nios® II Gen2 (or Classic) Processor Reference Handbook
- Nios II Gen2 (or Classic) Software Developer's Handbook
- Nios II Custom Instruction User Guide
- Using Nios II Floating-Point Custom Instructions Tutorial

Related Information

- **Nios II Classic Processor Reference Handbook**
- **Nios II Gen2 Processor Reference Handbook**
- **Nios II Classic Software Developer's Handbook**
- **Using Nios II Floating-Point Custom Instructions Tutorial**
- **GCC Floating-point Custom Instruction Support Overview**
- **GCC Single-precision Floating-point Custom Instruction Command Line**
- **Nios II Gen2 Software Developer's Handbook**
- **Nios II Custom Instruction User Guide**

Overview

The following figure shows the structure of the FPH2 component, with the display name **Floating Point Hardware 2** and the IP name **altera_nios_custom_instr_floating_point_2**. **Floating Point Hardware 2**

[1] First generation.
[2] Second generation.

© 2016 Altera Corporation. All rights reserved. ALTERA, ARRIA, CYCLONE, ENPIRION, MAX, MEGACORE, NIOS, QUARTUS and STRATIX words and logos are trademarks of Altera Corporation and registered in the U.S. Patent and Trademark Office and in other countries. All other words and logos identified as trademarks or service marks are the property of their respective holders as described at www.altera.com/common/legal.html. Altera warrants performance of its semiconductor products to current specifications in accordance with Altera's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Altera assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Altera. Altera customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.

ISO 9001:2008 Registered

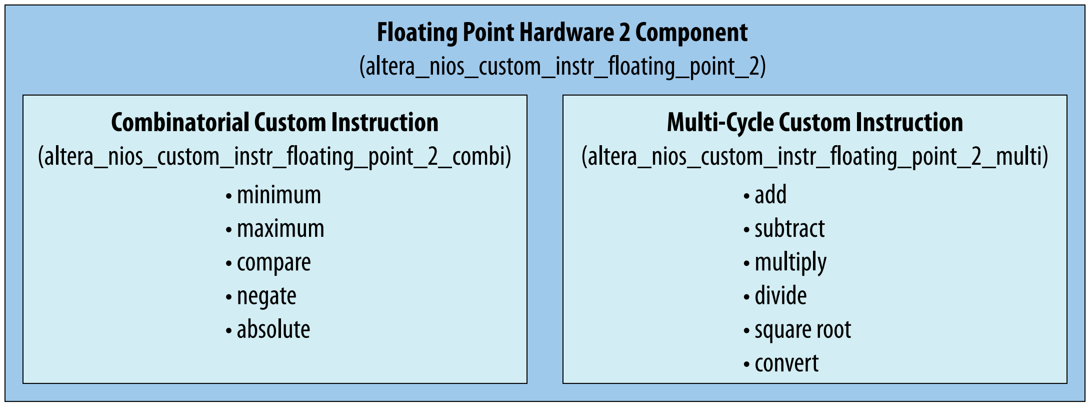
now part of Intel

packages all the floating point functions in a single Qsys component, consisting of the following subcomponents:

- **altera_nios_custom_instr_floating_point_2_combi**
- **altera_nios_custom_instr_floating_point_2_multi**

Figure 1: Custom Instruction Implementation

This figure lists the floating-operations implemented by each custom instruction.


Floating Point Hardware 2 Component
(altera_nios_custom_instr_floating_point_2)
Combinatorial Custom Instruction
(altera_nios_custom_instr_floating_point_2_combi)
• minimum
• maximum
• compare
• negate
• absolute
Multi-Cycle Custom Instruction
(altera_nios_custom_instr_floating_point_2_multi)
• add
• subtract
• multiply
• divide
• square root
• convert

The characteristics of the FPH2 are:

- Supports FPH1 operations (add, sub, multiply, divide) and adds support for square root, comparisons, integer conversions, minimum, maximum, negate, and absolute
- Single-precision floating-point values are stored in the Nios II general purpose registers
- VHDL only
- Qsys support only
- Single-precision only
- Optimized for FPGAs with 4-input LEs and 18-bit multipliers
- GCC and Nios II SBT (Software Build Tools) software support
- IEEE 754-2008 compliant except for:
  - Simplified rounding
  - Simplified NaN handling
  - No exceptions
  - No status flags
  - Subnormal supported on a subset of operations
- Binary-compatibility with FPH1
  - FPH1 implements Round-To-Nearest rounding. Because FPH2 implements different rounding, results might be subtly different between the two generations

FPH2 Resource Usage

When added to a typical system, the FPH2 consumes the following resources:

- Approximately 2500 4-input LEs
- Nine 9-bit multipliers
- Three M9K memories or larger

Floating Point Hardware 2 Benchmarking

The floating point hardware 2 component is benchmarked in the following hardware configuration:

- Cyclone V 5CGXFC7D6F31C6 device
- Nios II processor
- On-chip memory
- Pipeline bridge
- Timer
- JTAG UART
- Floating point hardware 2

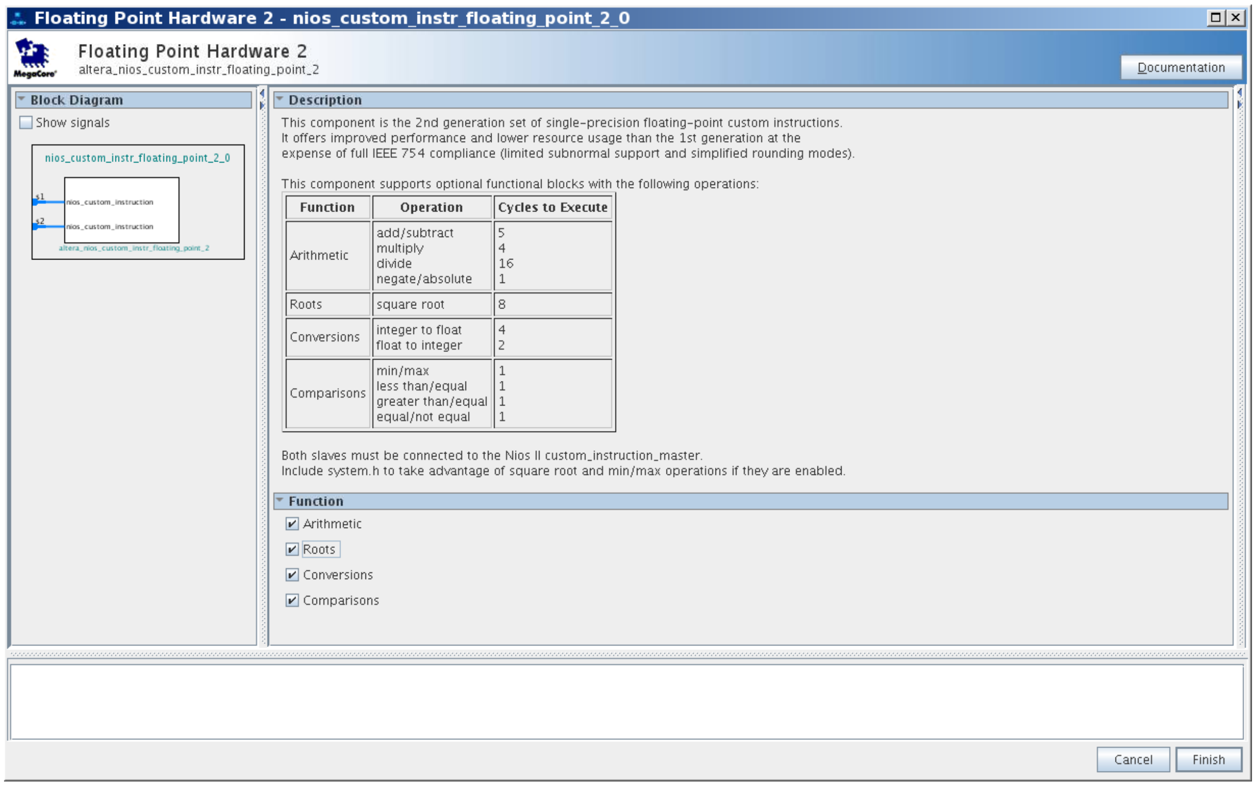
This hardware design achieves F_{MAX}=125 MHz.

Using the FPH2 in Qsys

To instantiate the FPH2 component in your system, in Qsys, locate the **Floating Point Hardware 2** component in the Project area of the Component Library. The FPH2 component is located under the “Embedded Processors” group in the Component Library.

The floating point hardware 2 component editor, shown in the figure below, allows you to selectively enable any of several groups of floating point custom instructions. By default, all instructions are enabled.

Figure 2: Floating Point Hardware 2 Component Editor


Floating Point Hardware 2 - nios_custom_instr_floating_point_2_0
Floating Point Hardware 2
altera_nios_custom_instr_floating_point_2
Documentation
Block Diagram
Show signals
nios_custom_instr_floating_point_2_0
s1
s2
nios_custom_instruction
nios_custom_instruction
altera_nios_custom_instr_floating_point_2
Description
Function
Cancel
Finish

This component is the 2nd generation set of single-precision floating-point custom instructions.
It offers improved performance and lower resource usage than the 1st generation at the
expense of full IEEE 754 compliance (limited subnormal support and simplified rounding modes).

This component supports optional functional blocks with the following operations:

Function	Operation	Cycles to Execute
Arithmetic	add/subtract multiply divide negate/absolute	5 4 16 1
Roots	square root	8
Conversions	integer to float float to integer	4 2
Comparisons	min/max less than/equal greater than/equal equal/not equal	1 1 1 1

Both slaves must be connected to the Nios II custom_instruction_master.
Include system.h to take advantage of square root and min/max operations if they are enabled.

- [✓] Arithmetic
- [✓] Roots
- [✓] Conversions
- [✓] Comparisons

In most cases, you should leave all floating point custom instructions enabled. However, for the MAX 10 device family in certain configurations, you might need to disable the **Roots** group.

MAX 10 devices cannot support the FPH2 square root instruction in the following configurations:

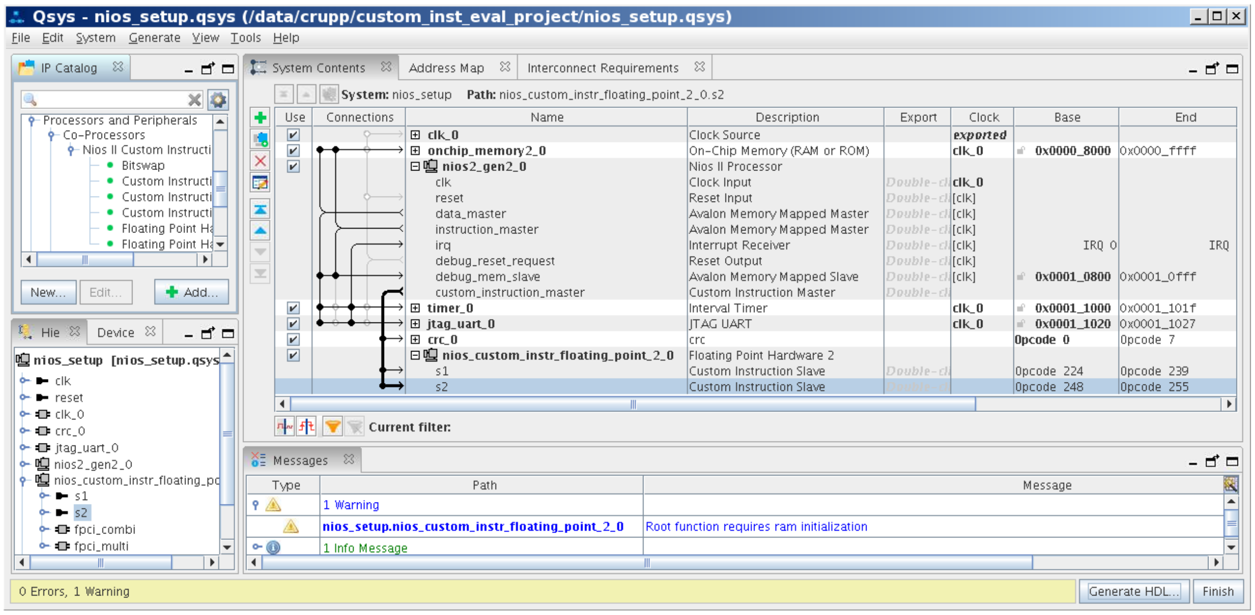
- Dual configuration mode
- Compressed configuration mode
- External RAM initialization disabled

The square root instruction uses a lookup table, requiring initialization that the MAX 10 cannot support in these configurations. Turn off the **Roots** option if you are targeting a MAX 10 device in one of these configurations.

When you disable one of the floating point instruction groups, software must implement the functions in that group (in this case, square root) if they are required. The BSP generator automatically creates this support. Refer to "Nios II SBT" for details.

The figure below shows Qsys with Nios II connected to the FPH2. The FPH2 has two slaves (s1 and s2). One slave is for the combinatorial custom instruction and the other slave is for the multi-cycle custom instruction. Connect both slaves to the Nios II custom_instruction_master by clicking the dot in the connections patch panel. The following figure shows how the connection should look.

Figure 3: Floating Point Hardware 2 Component in Qsys



Qsys - nios_setup.qsys (/data/crupp/custom_inst_eval_project/nios_setup.qsys)

File Edit System Generate View Tools Help

IP Catalog

- Processors and Peripherals
  - Co-Processors
    - Nios II Custom Instructi
      - Bitswap
      - Custom Instructi
      - Custom Instructi
      - Custom Instructi
      - Floating Point Ha
      - Floating Point Ha

New... Edit... Add...

Hie Device

- nios_setup [nios_setup.qsys
  - clk
  - reset
  - clk_0
  - crc_0
  - jtag_uart_0
  - nios2_gen2_0
  - nios_custom_instr_floating_pc
    - s1
    - s2
    - fpci_combi
    - fpci_multi

System Contents | Address Map | Interconnect Requirements

System: nios_setup **Path:** nios_custom_instr_floating_point_2_0.s2

Use	Connections	Name	Description	Export	Clock	Base	End
✓		⊞ **clk_0**	Clock Source		*exported*		
✓		⊞ **onchip_memory2_0**	On-Chip Memory (RAM or ROM)		clk_0	**0x0000_8000**	0x0000_ffff
✓		⊟ **nios2_gen2_0**	Nios II Processor				
		clk	Clock Input	Double-cl	clk_0		
		reset	Reset Input	Double-cl	[clk]		
		data_master	Avalon Memory Mapped Master	Double-cl	[clk]		
		instruction_master	Avalon Memory Mapped Master	Double-cl	[clk]		
		irq	Interrupt Receiver	Double-cl	[clk]	IRQ 0	IRQ
		debug_reset_request	Reset Output	Double-cl	[clk]		
		debug_mem_slave	Avalon Memory Mapped Slave	Double-cl	[clk]	**0x0001_0800**	0x0001_0fff
		custom_instruction_master	Custom Instruction Master	Double-cl			
✓		⊞ **timer_0**	Interval Timer		clk_0	**0x0001_1000**	0x0001_101f
✓		⊞ **jtag_uart_0**	JTAG UART		clk_0	**0x0001_1020**	0x0001_1027
✓		⊞ **crc_0**	crc			**Opcode 0**	Opcode 7
✓		⊟ **nios_custom_instr_floating_point_2_0**	Floating Point Hardware 2				
		s1	Custom Instruction Slave	Double-cl		Opcode 224	Opcode 239
		s2	Custom Instruction Slave	Double-cl		Opcode 248	Opcode 255

Current filter:

Messages

Type	Path	Message
⚠	1 Warning	
⚠	**nios_setup.nios_custom_instr_floating_point_2_0**	Root function requires ram initialization
ⓘ	1 Info Message	

0 Errors, 1 Warning

Generate HDL... Finish

The example in the figure above targets a MAX 10 device. Note the warning message, reminding you that there could be an issue with RAM initialization for the square root function.

After connecting the FPH2 to the Nios II, generate your system in Qsys as you normally would. Then use the Quartus® Prime software to compile the generated RTL, or use an RTL simulator, like Modelsim™, to perform simulations.

Note: If you use the Nios II software build tools (SBT) to create your software projects, the BSP generator creates a custom Newlib library for your floating point hardware. If you modify your floating point hardware configuration, you must regenerate and rebuild your BSP to ensure that Newlib is built correctly. For details, refer to "Nios II SBT".

Related Information

- **Nios II SBT** on page 18
- **Quartus Prime Standard Edition Handbook Volume 1: Design and Synthesis**
- **Newlib Library** on page 17

Floating Point Background

Related Information

- **GCC Floating-point Custom Instruction Support Overview**
- **GCC Single-precision Floating-point Custom Instruction Command Line**

IEEE 754 Format

The figure below shows the fields in an IEEE 754 32-bit single-precision value. The table below provides a description of the fields. Normal single-precision floating-point numbers have the value `(-1)`^{S} `* 1.FRAC * 2`$^{EXP -127}$.

Figure 4: Single-Precision Format

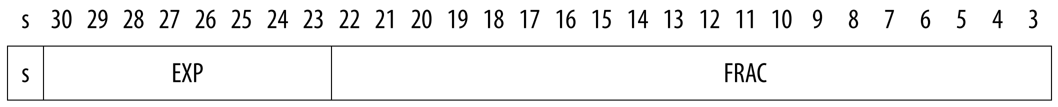

s 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3
s
EXP
FRAC

Table 1: Single-Precision Field Descriptions

Mnemonic	Name	Description
FRAC	Fraction	Specifies the fractional portion (right of the binary point) of the mantissa. The integer value (left of the binary point) is always assumed to be 1 for normal values so it is omitted. This omitted value is called the hidden bit. The mantissa ranges from ≥1.0 to <2.0.
EXP	Biased Exponent	Contains the exponent biased by the value 127. The biased exponent value 0x0 is reserved for zero and subnormal values. The biased exponent value 0xff is reserved for infinity and NaN. The biased exponent ranges from 1 to 0xfe for normal numbers (-126 to 127 when the bias is subtracted out).
S	Sign	Specifies the sign. 1 = negative, 0 = positive. Normal values, zero, infinity, and subnormals are all signed. NaN has no sign, so the S field is ignored.

The IEEE 754 standard provides the following special values:

- Zero (EXP=0, FRAC=0)
- Subnormal (EXP=0, FRAC≠0)
- Infinity (EXP=255, FRAC=0)
- NaN (EXP=255, FRAC≠0)

Note: Zero, subnormal, and infinity are signed as specified by the S field. NaN has no sign so the S field is ignored.

Unit in the Last Place

Unit in the last place (ULP) represents the value 2^{-23}, which is approximately `1.192093e-07`. The ULP is the distance between the closest straddling floating-point numbers a and b ($a \leq x \leq b$, $a \neq b$), assuming that the exponent range is not upper-bounded. The IEEE Round-to-Nearest modes produce results with a maximum error of one-half ULP. The other IEEE rounding modes (Round-to-Zero, Round-to-Positive-Infinity, and Round-to-Negative-Infinity) produce results with a maximum error of one ULP.

Encoding of Values

The table below shows how single-precision values are encoded across the 32-bit range from `0x0000_0000` to `0xffff_ffff`. Single-precision floating point numbers have the following characteristics:

- Precision (ρ) = 24 bits (23 bits in FRAC plus one hidden bit)
- Radix (β) = 2
- e_{min} = -126
- e_{max} = 127

The most-significant bit of FRAC is 0 for signaling NaNs (sNaN) and 1 for quiet NaNs (qNaN).

Table 2: Encoding of Values

Hexadecimal Value	Name	S	EXP	FRAC	Value (Decimal)
`0x0000_0000`	+0	0	`0x00`	`0x00_0000`	0.0
`0x0000_0001`	min pos subnormal	0	`0x00`	`0x00_0001`	1.40129846e–45 ($\beta^{emin-\rho+1} = 2^{-126-24+1} = 2^{-149}$)
`0x007f_ffff`	max pos subnormal	0	`0x00`	`0x7f_ffff`	1.1754942e-38
`0x0080_0000`	min pos normal	0	`0x01`	`0x00_0000`	1.17549435e–38 ($\beta^{emin} = 2^{-126}$)
`0x3f80_0000`	1	0	`0x7f`	`0x00_0000`	1.0 ($1.0x2^{0}$)
`0x4000_0000`	2	0	`0x80`	`0x00_0000`	2.0 ($1.0x2^{1}$)
`0x7f7f_ffff`	max pos normal	0	`0xfe`	`0x7f_ffff`	3.40282347e+38 ($(\beta - \beta^{1-\rho}) * 2^{emax} = (2 - 2^{1-24}) * 2^{127} = (2 - 2^{23}) * 2^{127}$)
`0x7f80_0000`	+∞	0	`0xff`	`0x00_0000`	
`0x7f80_0001`	min sNaN (pos sign)	0	`0xff`	`0x00_0001`	
`0x7fdf_ffff`	max sNaN (pos sign)	0	`0xff`	`0x3f_ffff`	
`0x7fe0_0000`	min qNaN (pos sign)	0	`0xff`	`0x40_0000`	
`0x7fff_ffff`	max qNaN (pos sign)	0	`0xff`	`0x7f_ffff`	
`0x8000_0000`	-0	1	`0x00`	`0x00_0000`	-0.0
`0x8000_0001`	max neg subnormal	1	`0x00`	`0x00_0001`	-1.40129846e–45
`0x807f_ffff`	min neg subnormal	1	`0x00`	`0x7f_ffff`	-1.1754942e-38
`0x8080_0000`	max neg normal	1	`0x01`	`0x00_0000`	-1.17549435e–38
`0xff7f_ffff`	min neg normal	1	`0xfe`	`0x7f_ffff`	-3.40282347e+38
`0xff80_0000`	-∞	1	`0xff`	`0x00_0000`	
`0xff80_0001`	max sNaN (neg sign)	1	`0xff`	`0x00_0001`	

Hexadecimal Value	Name	S	EXP	FRAC	Value (Decimal)
`0xffdf_ffff`	min sNaN (neg sign)	1	`0xff`	`0x3f_ffff`	
`0xffe0_0000`	max qNaN (neg sign)	1	`0xff`	`0x40_0000`	
`0xffff_ffff`	min qNaN (neg sign)	1	`0xff`	`0x7f_ffff`	

Rounding Schemes

When the exact result of a floating-point operation cannot be exactly represented as a floating-point value, it must be rounded.

The IEEE 754-2008 standard defines the default rounding mode to be “Round-to-Nearest RoundTiesTo-Even”. In the IEEE 754-1985 standard, this is called “Round-to-Nearest-Even”. Both standards also define additional rounding modes called “Round-to-Zero”, “Round-to-Negative-Infinity”, and “Round-to-Positive-Infinity”. The IEEE 754-2008 standard introduced a new optional rounding mode called “Round-to-Nearest RoundTiesAway”.

The FPH2 operations either support Nearest Rounding (RoundTiesAway), Truncation Rounding, or Faithful Rounding. The type of rounding is a function of the operation and is specified in Table 4-2. Because the software emulation library (used when FPH operations aren’t provided) and FPH1 implement Round-to-Nearest RoundTiesToEven, there can be differences in the results between FPH2 and these other solutions.

Nearest Rounding

Nearest Rounding corresponds to the IEEE 754-2008 “Round-to-Nearest RoundTiesAway” rounding mode. Nearest Rounding rounds the result to the nearest single-precision number. When the result is halfway between two single-precision numbers, the rounding chooses the upper number (larger values for positive results, smaller value for negative results).

Nearest Rounding has a maximum error of one-half ULP. Errors are not evenly distributed, because nearest rounding chooses the upper number more often than the lower number when results are randomly distributed.

Truncation Rounding

Truncation Rounding corresponds to the IEEE 754-2008 “Round-To-Zero” rounding mode. Truncation Rounding rounds results to the lower nearest single-precision number.

Truncation Rounding has a maximum error of one ULP. Errors are not evenly distributed.

Faithful Rounding

Faithful Rounding rounds results to either the upper or lower nearest single-precision numbers. Therefore, Faithful Rounding produces one of two possible values. The choice between the two is not defined.

Faithful Rounding has a maximum error of one ULP. Errors are not guaranteed to be evenly distributed.

Note: Faithful Rounding mode is not defined by IEEE 754.

Rounding Examples

The table below shows examples of the supported rounding schemes for decimal values assuming rounded normalized decimal values with two digits of precision, like one-digit integer or one-digit fraction.

Table 3: Decimal Rounding Examples

Unrounded Value	Nearest Rounding	Truncation Rounding	Faithful Rounding
3.34	3.3	3.3	3.3 or 3.4
6.45	6.5	6.4	6.4 or 6.5
2.00	2.0	2.0	2.0 or 2.1
8.99	9.0	8.9	8.9 or 9.0
-1.24	-1.2	-1.2	-1.2 or -1.3
-3.78	-3.8	-3.7	-3.7 or -3.8

Special Cases

The table below lists the results of some IEEE 754 special cases. The *x* represents a normal value. The FPH2 are compliant for all of these cases.

Results are assumed to be correctly signed so signs are omitted when they are not important. When the sign is relevant, signs are shown with extra parenthesis around the value such as (+∞). The value *x* in the table represents any non-NaN value.

Comparisons ignore the sign of zero for equality. For example, (-0) == (+0) and (+0) ≤ (-0). Comparisons that don't include equality, like > and <, don't consider -0 to be less than +0. Comparisons return false if either or both inputs are NaN. The min and max operations return the non-NaN input if one of their inputs is NaN and the other is non-NaN. Other operations that produce floating-point results return NaN if any or all of their inputs are NaN.

Table 4: Special Cases

Operation	Special Cases			
fdivs	0/0=NaN	∞/∞=NaN	0/∞=0, ∞/0=∞	NaN/*x*=NaN, *x*/NaN=NaN, NaN/NaN=NaN
fsubs	(+∞)-(+∞)=NaN	(-∞)-(-∞)=NaN	(-0)-(-0)=+0	NaN-*x*=NaN, *x*-NaN=NaN, NaN-NaN=NaN
fadds	(+∞)+(-∞)=NaN	(-∞)+(+∞)=NaN	(+0)+(-0)=+0, (-0)+(+0)=+0	NaN+*x*=NaN, *x*+NaN=NaN, NaN+NaN=NaN
fmuls	0*∞=NaN	∞*0=NaN		NaN*x*=NaN, *x**NaN=NaN, NaN*NaN=NaN
fsqrts	sqrt(-0) =-0	sqrt(*x*) =NaN, *x*<-0		sqrt(NAN) =NaN
fixsi & round	int(>2^{31}-1)= 0x7fffffff, int(+∞) =0x7fffffff	int(<-2^{31})= x80000000, int(-∞)=0x80000000		int(NaN)=undefined

Operation	Special Cases			
fmins	min((+0),(-0))=(-0)	min((-0),(+0))=(-0)		min(NaN,n)=*x*, min(*x*,NaN)=*x*, min(NaN,NaN)=NaN, min(+∞,*x*)=*x*, min(-∞,*x*)=-∞
fmaxs	max((+0),(-0))=(+0)	max((-0),(+0))=(+0)		max(NaN,*x*)=*x*, max(*x*,NaN)=*x*, max(NaN,NaN)=NaN, max(+∞,*x*)=+∞, max(-∞,*x*)=*x*
fcmplts (<)	(+∞)<(+∞)=0	(-∞)<(-∞)=0	(-0)<(+0)=0, (+0)<(-0)=0	NaN<*x*=0, *x*<NaN=0, NaN<NaN=0
fcmples (≤)	(+∞)≤(+∞)=1	(-∞)≤(-∞)=1	(+0)≤(-0)=1, (-0)≤(+0)=1	NaN≤*x*=0, *x*≤NaN=0, NaN≤NaN=0
fcmpgts (>)	(+∞)>(+∞)=0	(-∞)>(-∞)=0	(-0)>(+0)=0, (+0)>(-0)=0	NaN>*x*=0, *x*>NaN=0, NaN>NaN=0
fcmpges (≥)	(+∞)≥(+∞)=1	(-∞)≥(-∞)=1	(-0)≥(+0)=1, (+0)≥(-0)=1	NaN≥*x*=0, *x*≥NaN=0, NaN≥NaN=0
fcmpeqs (=)	(+∞)=(+∞)=1	(-∞)=(-∞)=1	(-0)=(+0)=1	(NaN==*x*)=0, (*x*==NaN)=0, (NaN==NaN)=0
fcmpnes (≠)	(+∞)≠ (+∞)=0	(-∞)≠ (-∞)=0	(-0)≠(+0)=0	NaN≠*x*=0, *x*≠NaN=0, NaN≠NaN=0

Feature Description

The FPH2 are implemented with one combinatorial custom instruction and one multi-cycle custom instruction. The combinatorial custom instruction implements the comparison, minimum, maximum, negate, and absolute operations. The multi-cycle custom instruction implements the add, subtract, multiply, divide, square root, and conversion operations.

Note: All operations are required. There are no configurable options.

IEEE 754 Compliance

Floating point hardware 2 operations are compliant with the IEEE 754-2008 standard, except for the following:

- No traps/exceptions.
- No status flags.
- Remainder and conversions between binary and decimal operations are not supported. These are provided by the software emulation library.
- No support for round-to-nearest-even mode. Nearest Rounding, Truncation Rounding, or Faithful Rounding is used, depending on the operator.
- Subnormals are not supported by the add, subtract, multiply, divide, and square root operations. Subnormal inputs are treated as signed zero and subnormal outputs are never created (result is signed zero instead). This treatment of subnormal values called flush-to-zero.[3]

[3] Subnormals are supported by comparison, minimum, maximum, float-to-integer, negate, and absolute operations, so these operations are IEEE 754-2008 compliant.

- Subnormals cannot be created by the integer2float conversion operation. This behavior is IEEE 754 compliant.
- No distinction between signaling and quiet NaNs as input operands. Any result that produces a NaN may produce either a signaling or quiet NaN.
- A NaN result with one or more NaN input operands is not guaranteed to return any of the input NaN values; the NaN result can be a different NaN than the input NaNs.

Exception Handling

The FPH2 component does not support exceptions. Instead, it creates a specific result. The following table shows the FPH2 results created for operations that would trigger an IEEE 754 exception.

Table 5: IEEE 754 Exception Cases

IEEE 754 Exception	FPH2 Result
Invalid	NaN
Division by zero	Signed infinity
Overflow	Signed infinity
Underflow	Signed zero
Inexact	Normal number

Operations

The table below provides a detailed summary of the FPH2 operations. The values “a” and “b” are assumed to be single-precision floating-point values. The following list provides detailed information about each column:

- **Operation**(4)—Provides the name of the floating-point operation. The names match the names of the corresponding GCC floating point hardware command-line options except for “round”, which has no GCC support.
- **N**—Provides the 8-bit fixed custom instruction N value for the operation. FPH2 component uses fixed N values that occupy the top 32 Nios II custom instruction N values (224 to 255). The FPH1 also use fixed N values (252 to 255) and the FPH2 assign the same operations to those N values to maintain compatibility.
- **Cycle**(5)—Specifies the number of cycles it takes to execute the instruction. A combinatorial custom instruction takes 1 cycle. A multi-cycle custom instruction always requires at least 2 cycles. An N-cycle custom instruction has N-2 register stages inside the custom instruction because the Nios II registers the result from the custom instruction and also allows another cycle for g wire delays in the source operand bypass multiplexers. The **Cycle** column does not include the extra cycles (maximum of 2) required because the Nios II/f processor stalls the instruction following the multi-cycle custom instruction if that instruction uses the result within 2 cycles. These extra cycles are required because multi-cycle instructions are late-result instructions.
- **Result**—Describes the computation performed by the operation.

(4) For more information, refer to "-mcustom-<operation>".
(5) For more information, refer to one of the Nios II Processor Reference Handbooks.

- **Subnormal**—Describes how the operation treats subnormal inputs and subnormal outputs.
- **Rounding**[6]—Describes how the FPH2 component rounds the result. The possible choices are Nearest, Truncation, Faithful, and none.
- **GCC Inference**—Shows the C code from which GCC infers the custom instruction operation.

Table 6: FPH2 Operation Summary

Operation	N	Cycles	Result	Subnormal	Rounding	GCC Inference
fdivs	255	16	a/b	flush-to-0	Nearest	a/b
fsubs	254	5	a-b	flush-to-0	Faithful	a-b
fadds	253	5	a+b	flush-to-0	Faithful	a+b
fmuls	252	4	a*b	flush-to-0	Faithful	a*b
fsqrts	251	8	sqrt(a)	flush-to-0	Faithful	sqrtf()
floatis	250	4	int_to_float(a)	Does not apply	Does not apply	Casting
fixsi	249	2	float_to_int(a)	flush-to-0	Truncation	Casting
round	248	2	float_to_int(a)	flush-to-0	Nearest	lroundf() [7]
reserved	234 to 247	Undefined	undefined			
fmins	233	1	(a<b) ? a : b	supported	None	fminf()[7]
fmaxs	232	1	(a<b) ? b : a	supported	None	fmaxf()[7]
fcmplts	231	1	(a<b) ? 1 : 0	supported	None	a<b
fcmples	230	1	(a≤b) ? 1 : 0	supported	None	a<=b
fcmpgts	229	1	(a>b) ? 1 : 0	supported	None	a>b
fcmpges	228	1	(a≥b) ? 1 : 0	supported	None	a>=b
fcmpeqs	227	1	(a=b) ? 1 : 0	supported	None	a==b
fcmpnes	226	1	(a≠b) ? 1 : 0	supported	None	a!=b
fnegs	225	1	-a	supported	None	-a
fabss	224	1	|a|	supported	None	fabsf()

Related Information

- Rounding Schemes on page 8
- -mcustom-<operation> on page 14
- Nios II Classic Processor Reference Handbook
- Nios II Gen2 Processor Reference Handbook

[6] For more information, refer to "Rounding Schemes". A rounding of "none" means that the result does not need to be rounded.
[7] Nios II GCC cannot reliably replace calls to these Newlib floating-point functions with the equivalent custom instruction. For information about using these functions, refer to "C Macros for round(), fmins() , and fmaxs()".

- C Macros for round(), fmins(), and fmaxs() on page 18
- GCC Command Line Options
- Newlib Documentation page
- GCC Floating-point Custom Instruction Support Overview
- GCC Single-precision Floating-point Custom Instruction Command Line
- Nios II Custom Instruction User Guide

Software Issues

Nios II GCC

The Nios II Embedded Design Suite includes the Nios II GCC. The FPH2 component is supported by Nios II EDS versions 15.1 and higher (GCC v4.7.3 and higher).

Related Information

- GCC Command Line Options
- GCC Floating-point Custom Instruction Support Overview
- GCC Single-precision Floating-point Custom Instruction Command Line

Inference

The GCC compiler infers most FPH2 operations from C source code. The table in "Operations" lists all the operations and shows how the FPH2 are inferred.

Note: GCC does not infer Newlib math functions. These functions can be replaced with their equivalent custom instruction using the __builtin_custom_* facility of GCC.

The **system.h** header file provides a C `#define` macro declaration that re-defines the required Newlib math functions to use the corresponding custom instruction instead.

Related Information

- C Macros for round(), fmins(), and fmaxs() on page 18
- Operations on page 11
- Newlib Documentation page

Conversions

The FPH2 component provides functions for conversion between signed integer types (C `short`, `int` and `long` types) and 32-bit single-precision floating point types (C `float` type). The Nios II GCC compiler infers these hardware functions when compiled code converts data between these types, for example in C casting.

The FPH2 component does not provide functions for conversion between unsigned integer types and floating point. When converting between unsigned integer types and float types, the compiler implements software emulation. Therefore conversion to and from unsigned integers is much slower than conversion to and from signed integers.

If you do not need the extra range of positive values obtained when converting a float to an unsigned integer directly, you can use the FPH2 and avoid using the software emulation if you modify your C code to first cast the float type to an int type or long type and then cast to the desired unsigned integer type. For example, instead of:

```
float f;
unsigned int s = (unsigned int)f; // Software emulation
```

use:

```
float f;
unsigned int s = (unsigned int)(int)f; // FPH2
```

The FPH2 provides two operations for converting single-precision floating-point values to signed integer values:

- `fixsi`
- `round`

The `fixsi` operation performs truncation when converting a float to a signed integer. For example, `fixsi` converts 4.8 to 4 and -1.5 to -1. GCC follows the C standard and invokes the `fixsi` operation whenever source code uses a cast or any time that C automatically converts a float to a signed integer.

The `round` operation performs Nearest Rounding (tie-rounds-away) when converting a float to a signed integer. For example, `round` converts 4.8 to 5 and -1.5 to -2. Software can invoke the `round` operation by calling the custom instruction directly, or by using the `#define` provided in **system.h**, which replaces the Newlib `lroundf()` function.

Related Information

Newlib Documentation page

Nios II Floating-Point Options

GCC options that are only provided by the Nios II port of GCC are described below.

-mcustom-<operation>

The `-mcustom-<operation>` command-line option instructs GCC to call custom instructions instead of emulating the specified operation. The syntax of the `-mcustom-<operation>` is as follows:

```
-mcustom-<operation>=N
```

N custom instruction value, an unsigned decimal. For a complete list of the operations and their N values, refer to the table in "Operations".

By default, the compiler implements all floating point operations in software. You can also specify software emulation for an individual instruction with the `-mno-custom-<operation>` command-line option.

Note: The command line can specify multiple `-mcustom-` switches. If there is a conflict, the last switch on the command line takes effect.

The following command-line options should be passed to GCC to instruct it to use all operations provided by the FPH2 that can be inferred by GCC. For more information, refer to "Inference".

For users of the Nios II SBT, these command-line arguments are automatically added to the invocation of GCC by the generated makefiles. For more information, refer to "Nios II SBT".

```
-mcustom-fabss=224
-mcustom-fnegs=225
-mcustom-fcmpnes=226
```

```
-mcustom-fcmpeqs=227
-mcustom-fcmpges=228
-mcustom-fcmpgts=229
-mcustom-fcmples=230
-mcustom-fcmplts=231
-mcustom-fmins=232
-mcustom-fmaxs=233
-mcustom-round=248
-mcustom-fixsi=249
-mcustom-floatis=250
-mcustom-fmuls=252
-mcustom-fadds=253
-mcustom-fsubs=254
-mcustom-fdivs=255
```

Note: There is no command-line option for the round operation.

Related Information

- Inference on page 13
- Operations on page 11
- Nios II SBT on page 18

pragmas

GCC supports pragmas located in source code files to override the `-mcustom` command-line options. The pragmas affect the entire source file.

The following pragma tells GCC to call custom instruction N (where N is a decimal integer from 0 to 255) to implement the specified floating-point operation:

```
#pragma GCC target(“custom-<operation>=N”)
```

The following pragma tells GCC to use the software emulation instead of the custom instruction to implement the specified floating-point operation:

```
#pragma GCC targer(“no-custom-<operation>”)
```

Note: There is no pragma support for the round operation.

-mcustom-fpu-cfg

If you specify the `-mcustom-fpu-cfg` option on the GCC linker command line, it chooses a precompiled Newlib library with floating-point support. The precompiled libraries only use operations (add, subtract, multiply, and divide) supported by FPH1.

Note: For FPH2, Altera does not recommend using the `-mcustom-fpu-cfg` option.

Related Information

- Newlib Documentation page
- Newlib Library on page 17
- "Nios II Options" in GCC Command Options (gcc.gnu.org)

Generic Floating-Point Options

There are options provided by GCC and are not only provided by Nios II GCC. However, these options have Nios II specific behaviors in some cases.

-fno-math-errno

From the GCC documentation:

```
“Do not set ERRNO after calling math functions that are executed with a single
instruction, e.g., sqrt. A program that relies on IEEE exceptions for math error
handling may want to use this flag for speed while maintaining IEEE arithmetic
compatibility.”
```

If you specify `-fno-math-errno` on the GCC command line, the compiler maps calls to `sqrtf()` directly to the `fsqrts` custom instruction. Otherwise, by default GCC adds several instructions after the `fsqrts` custom instruction to check for a NaN result, indicating an attempt to take the square root of a negative number. If `fsqrts` returns NaN, the code calls the Newlib `sqrtf()` function to set the C `errno` variable.

Typically, this overhead is undesirable. Altera recommends that you enable `-fno-math-errno` to eliminate the overhead of calling `sqrtf()`.

If you use the Nios II SBT, the generated makefiles set `-fno-math-errno` by default. You can override this behavior by setting `-fmath-errno` in the `CPPFLAGS` make variable.

The `-ffinite-math-only` option also eliminates the overhead of checking for NaN result for square root. However, this option also has other effects. Refer to "-ffinite-math-only" for details about this option.

Related Information

- Nios II SBT on page 18
- -ffinite-math-only on page 17
- Newlib Documentation page

-fsingle-precision-constant

From the GCC documentation:

```
“Treat floating-point constants as single-precision constants instead of implicitly
converting them to double-precision constants.”
```

For FPH2, the Nios II SBT omits `-fsingle-precision-constant` from the makefile GCC command line by default. This behavior contrasts with SBT support for FPH1, which sets this option with `-mcustom-fpu-cfg`. The SBT does not use `-fsingle-precision-constant` for FPH2 because it can cause problems for double-precision code.

You can enable `-fsingle-precision-constant` if you are sure it will not cause problems for your code. In general, it is better to cast floating-point constants to the `float` type, or use the 'f' suffix (for example `3.14f`), because these approaches are localized and independent of compiler options.

Related Information

Nios II SBT on page 18

-funsafe-math-optimizations

From the GCC documentation:

```
“Allow optimizations for floating-point arithmetic that (a) assume that arguments
and results are valid and (b) may violate IEEE or ANSI standards. When used at link-
time, it may include libraries or startup files that change the default FPU control
word or other similar optimizations.”
```

The `-funsafe-math-optimizations` option is not required, because FPH2 does not implement transcendental functions (`sin()`, `cos()`, `tan()`, `atan()`, `exp()`, and `log()`).

This option would be required if the floating point hardware implemented the transcendental functions. GCC requires this option to ensure that application code does not inadvertently use hardware accelerators that might be problematic.

-ffinite-math-only

From the GCC documentation:

```
“Allow optimizations for floating-point arithmetic that assume that arguments and
results are not NaNs or +-Infs.”
```

Programmers are recommended to experiment with this option to determine how it affects their code.

The `-ffinite-math-only` option also eliminates the GCC overhead created on calls to `sqrtf()` like `–fno-math-errno`.

Related Information

-fno-math-errno on page 16

-fno-trapping-math

From the GCC documentation:

```
“Compile code assuming that floating-point operations cannot generate user-visible
traps. These traps include division by zero, overflow, underflow, inexact result
and invalid operation. This option implies -fno-signaling-nans. Setting this option
may allow faster code if one relies on “non-stop” IEEE arithmetic, for example.”
```

Programmers are recommended to experiment with this option to determine how it affects their code.

-frounding-math

From the GCC documentation:

```
“Disable transformations and optimizations that assume default floating point
rounding behavior. This is round-to-zero for all floating point to integer
conversions, and round-to-nearest for all other arithmetic truncations. This option
should be specified for programs that change the FP rounding mode dynamically, or
that may be executed with a non-default rounding mode. This option disables
constant folding of floating point expressions at compile-time (which may be
affected by rounding mode) and arithmetic transformations that are unsafe in the
presence of sign-dependent rounding modes.”
```

Programmers are recommended to experiment with this option to determine how it affects their code.

Newlib Library

The Nios II SBT include the Newlib library (C and math) in precompiled and source versions. However, the precompiled Newlib libraries are not recommended for FPH2.

You should compile Newlib from source code with individual `–mcustom-<operation>` options, selected to match your hardware configuration. This allows Newlib to incorporate the benefits of all FPH2 operations that can be inferred by GCC. If you use the Nios II software build tools, the BSP generator takes care of this for you.

The Newlib `isgreater()`, `isgreaterequal()`, `isless()`, `islessequal()`, and `islessgreater` macros defined in **math.h** use the normal comparison operators (such as. < and >=), so these macros automatically use the FPH2 comparison operations.

The Newlib `fmaxf()` and `fminf()` functions return the maximum or minimum numeric value of their arguments. NaN arguments are treated as missing data: if one argument is a NaN and the other numeric, then the functions return the numeric value. The FPH2 fmaxs/fmins operations match this behavior.

Note: If you modify your floating point hardware configuration, you must regenerate and rebuild your BSP to ensure that Newlib is built correctly. For details, refer to "Nios II SBT".

Related Information

- **Nios II SBT** on page 18
- **-mcustom-<operation>** on page 14
- **pragmas** on page 15
- **Newlib Documentation page**
- **GCC Floating-point Custom Instruction Support Overview**
  For more information about the *GCC Floating-point Custom Instruction Support Overview*.

C Macros for round(), fmins(), and fmaxs()

Nios II GCC cannot reliably replace calls to the following Newlib floating-point functions with the equivalent custom instruction, even though it has `-mcustom-<operation>` command-line options and pragma support for them:

- `round()`
- `fmins()`
- `fmaxs()`

Instead, these custom instructions must be invoked directly using the __builtin_custom_* facility of GCC. **system.h** provides the required `#define` macros to invoke the custom instructions directly. The Nios II Software Build Tools automatically include this header file in your C source files. For information about built-in functions, refer to the *Nios II Custom Instruction User Guide*.

Related Information

- **GCC Command Line Options**
- **Newlib Documentation page**
- **Nios II Custom Instruction User Guide**

Nios II SBT

The Software Build Tools (SBT) are tools used to create Altera HAL-based Board Support Packages (BSP) and application and library makefiles for embedded software running on a Nios II. These tools come in command-line and Eclipse GUI-based forms.

For more information about the SBT, refer to one of the Nios II Software Developer's Handbooks.

When these tools are used to generate a BSP for a Nios II with the FPH2 component connected to that Nios II, the **sw.tcl** file in the component causes the BSP and any applications or libraries that use that BSP to be aware of the presence of the FPH2. In particular, **sw.tcl** performs the following functions:

- Examines the system you created in Qsys, and determines the correct GCC flags for your floating point hardware.
- Creates makefile rules to pass the `-mcustom-<operation>` options to GCC, so it knows to use the available FPH2 operations instead of the software emulation code to implement the specified floating-point operations.
- Creates makefile rules to pass the `-fno-math-errno` option to GCC, to eliminate the overhead of detecting NaN results and setting the `errno` variable for calls to `sqrtf()`.
- Adds `#define` macro declarations to **system.h** for the Newlib math library routines that GCC does not reliably replace with custom instructions. For more information, refer to "C Macros for float(), fmins(), and fmaxs()".
- Creates makefile rules to generate a correct version of Newlib. Uses the GCC flags determined from your hardware system.

Note: If you modify your floating point hardware configuration, you must regenerate and rebuild your BSP to ensure that Newlib is built correctly.

Related Information

- **C Macros for round(), fmins(), and fmaxs()** on page 18
- **Operations** on page 11
- **Nios II Classic Software Developer's Handbook**
- **GCC Floating-point Custom Instruction Support Overview**
  For more information about the *GCC Floating-point Custom Instruction Support Overview*.
- **Nios II Gen2 Software Developer's Handbook**

Document Revision History for Nios II Floating Point Hardware 2 Component User Guide

Date	Version	Changes
May 2016	2016.05.03	Enhanced Qsys component editor and software build tools, allowing selective implementation of floating point functions.
May 2015	2015.05.22	Initial release.