Nios II Floating Point Hardware 2 Component User Guide

2015.05.22

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. Other floating-point operations (including all double-precision operations) are implemented by software emulation.

The Floating Point Hardware 2 (FPH2)[2] has 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 Classic Processor Reference Handbook
- Nios II Gen2 Processor Reference Handbook
- Nios II Classic Software Developer's Handbook
- Nios II Gen2 Software Developer's Handbook
- Nios II Custom Instruction User Guide
- Using Nios II Floating-Point Custom Instructions Tutorial

Note: The following topic discusses single-precision and floating-point custom instructions. For more information, refer to the *GCC Floating-point Custom Instruction Support Overview* and *GCC Single-precision Floating-point Custom Instruction Command Line* webpages.

Related Information

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

(1) First generation.
(2) Second generation.

- GCC Single-precision Floating-point Custom Instruction Command Line

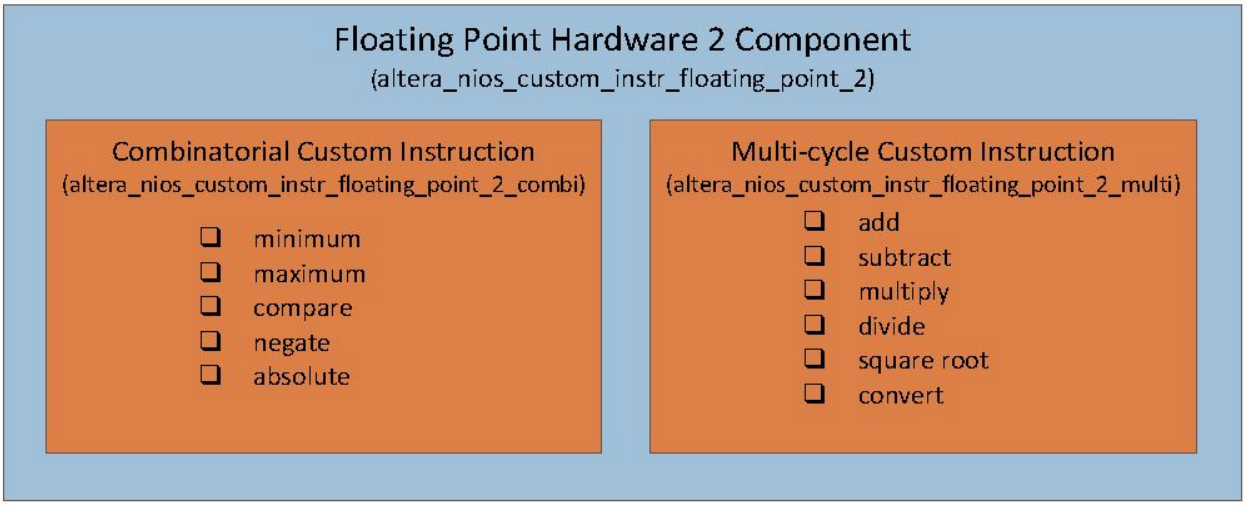
Overview

The "Custom Instruction Implementation" figure shows the structure of the FPH2 component. The component is packaged as one Qsys component with the display name **Floating Point Hardware 2** and the name **altera_nios_custom_instr_floating_point_2**. The component is composed of the following two custom instruction components:

- **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.



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

Resources

The resource utilization by the FPH2 are as follows:

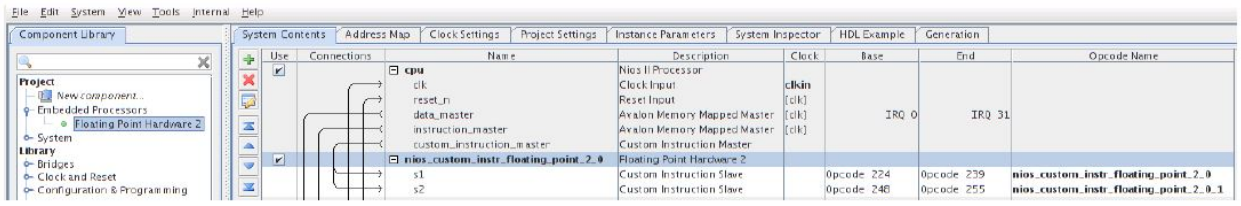
- ~2500 4-input LEs
- Nine 9-bit multipliers
- Three M9K memories or larger

Using the FPH2 in Qsys

To use the FPH2 in Qsys, you must find the component called “Floating Point Hardware 2” in the Project area of the Component Library. The FPH2 component is located under the “Embedded Processors” group in the Component Library.

The "Qsys Screenshot" figure shows a screenshot of 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 it should look.

Figure 2: Qsys Screenshot



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

Related Information

Quartus II Handbook Volume 1: Design and Synthesis

Floating-Point Background

For a better understanding of the FPH2, this section provides background information about floating-point.

Note: The following topic discusses single-precision and floating-point custom instructions. For more information, refer to the *GCC Floating-point Custom Instruction Support Overview* and *GCC Single-precision Floating-point Custom Instruction Command Line* webpages.

Related Information

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

IEEE 754 Format

The "Single-Precision Format" figure shows the fields in an IEEE 754 32-bit single-precision value and the "Single-Precision Field Descriptions" table provides a description of the fields. Normal single-precision floating-point numbers have the value $(-1)^{S} * 1.FRAC * 2^{EXP\ -127}$.

Figure 3: Single-Precision Format

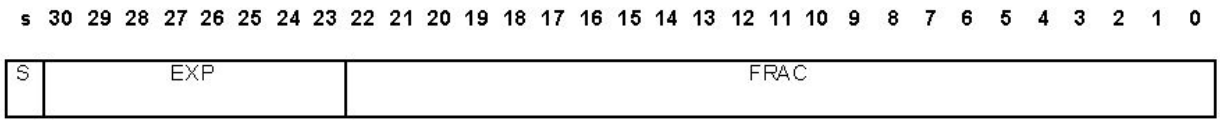


Table 1: Single-Precision Field Descriptions

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 either 0 or 1 and 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

A unit in the last place (ulp) is 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 "Encoding of Values" table shows how single-precision values are encoded across the 32-bit range from `0x0000_0000` to `0xffff_ffff`. The precision ρ for single precision is 24 (23 bits in FRAC plus one hidden bit), the radix β is 2, e_{min} is -126, e_{max} is 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	Decimal Value
`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`	

Hexadecimal Value	Name	S	EXP	FRAC	Decimal Value
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	
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	

In summary the order of values from 0x0000_0000 to 0xffff_ffff is as follows:

- +0
- +subnormal
- +normal
- +∞
- +NaN
- -0
- -subnormal
- -normal
- -∞
- -NaN

Rounding Schemes

The result of an operation on floating-point numbers may not be exactly represented as a floating-point value so it has to 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 was 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).

The maximum error of Nearest Rounding is one-half `ulp`. Errors are not evenly distributed because the upper number is chosen more often than the lower number for randomly distributed results.

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.

The maximum error of Truncation Rounding is one `ulp`. Errors are not evenly distributed.

Faithful Rounding

Faithful Rounding is not an IEEE 754 defined rounding mode. Faithful Rounding rounds results to either the upper or lower nearest single-precision numbers. So, the result produced is either one of two possible values and the choice between the two is not defined.

The maximum error of Faithful Rounding is one ulp. Errors are not guaranteed to be evenly distributed.

Rounding Examples

The "Decimal Rounding Examples" table 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 "Special Cases" table lists the results of some IEEE 754 special cases. The value ‘x’ is 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 is 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
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

Operation	Special Cases			
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

The FPH2 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.
- Round-to-nearest-even not supported. Instead, Nearest Rounding, Truncation Rounding, or Faithful Rounding (depending on the operator) is supported.
- Subnormals are not supported by add, subtract, multiply, divide, square root, and float2integer conversion operations. Subnormal inputs are treated as signed zero and subnormal outputs are never created (result is signed zero instead). This treatment of subnormal values is known in the industry as flush-to-zero.(3)
- Subnormals cannot be created by the integer2float conversation operation. This behavior is IEEE 754 compliant and is listed here for completeness.
- No distinction is made 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 does not support exceptions. Instead, a specific result is created. The "IEEE 754 Exception Cases" 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

(3) Subnormals are supported by comparison, minimum, maximum, comparison, negate, and absolute operations so these are IEEE 754-2008 compliant.

IEEE 754 Exception	FPH2 Result
Underflow	Signed zero
Inexact	Normal number

Operations

The "FPH2 Operation Summary" table 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 FPH command-line options except for “round” which has no GCC support.
- **N**—Provides the 8-bit fixed custom instruction N value for the operation. The FPH2 use 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 is always at least 2 cycles. An N cycle multi-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 doesn’t include the extra cycles (maximum of 2) that an instruction following the multi-cycle custom instruction is stalled by the Nios II/f if the instruction uses the result within 2 cycles. These extra cycles are because multi-cycle instructions are late result instructions.
- **Result**—Describes the computation performed by the operation.
- **Subnormal**—Describes how the operation treats subnormal inputs and subnormal outputs.
- **Rounding**[6]—Describes how the result is rounded. The possible choices are Nearest, Truncation, Faithful, and none.
- **GCC Inference**—Shows the C code that 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()

(4) For more information, refer to the "-mcustom-<operation>" chapter.
(5) For more information, refer to one of the Nios II Processor Reference Handbooks.
(6) For more information, refer to the "Rounding Schemes" chapter. A rounding of “none” means that the result doesn’t have to be rounded.
(7) Nios II GCC 4.7.3 is not able to reliably replace calls to Newlib floating-point functions with the equivalent custom instruction even though it has –mcustom-<operation> command-line options and pragma support for these. Instead, the custom instruction must be invoked directly using the __builtin_custom_* facility of GCC. The FPH2 include a C header file that provides the required #define macros to invoke the custom instruction directly. You should include this header file in your C source files.

Operation	N	Cycles	Result	Subnormal	Rounding	GCC Inference
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()

Note: The following topic discusses single-precision and floating-point custom instructions. For more information, refer to the *GCC Floating-point Custom Instruction Support Overview* and *GCC Single-precision Floating-point Custom Instruction Command Line* webpages.

Related Information

- **Rounding Schemes** on page 6
- **-mcustom-<operation>** on page 13
- **Nios II Classic Processor Reference Handbook**
- **Nios II Gen2 Processor Reference Handbook**
- **GCC 4.7.3 Command Line Options**
- **Newlib Documentation page**
- **GCC Floating-point Custom Instruction Support Overview**
- **GCC Single-precision Floating-point Custom Instruction Command Line**

Software Issues

Nios II GCC

The version of the Nios II GCC is 4.7.3 and it is used with the FPH2.

Note: The following topic discusses single-precision and floating-point custom instructions. For more information, refer to the *GCC Floating-point Custom Instruction Support Overview* and *GCC Single-precision Floating-point Custom Instruction Command Line* webpages.

Related Information

- **GCC 4.7.3 Command Line Options**
- **GCC Floating-point Custom Instruction Support Overview**
- **GCC Single-precision Floating-point Custom Instruction Command Line**

Inference

GCC is able to infer most FPH2 operations from C source code. The "FPH2 Operation Summary" table in the "Operations" chapter 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 `altera_nios_custom_instr_floating_point_2.h` C header file provides a `#define` macro declaration that re-defines the required `Newlib ()` math functions to use the corresponding custom instruction instead.

Related Information

- **Operations** on page 10
- **altera_nios_custom_instr_floating_point_2.h** on page 17
- **Newlib Documentation page**

Conversions

The FPH2 provide conversion operations between a signed integer type (C “short”, “int” and “long” types) and a 32-bit single-precision (C “float” type). These conversions are inferred when GCC needs to convert between these types, like casting. When converting between unsigned integer types and a float type, the software emulation is used instead of the FPH2 so it is much slower.

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. The `round` operation is invoked by calling the custom instruction directly or using the provided `#define` that 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 using the software emulation to execute the specified operation. The `-mcustom-<operation>` is followed by an equals sign and the custom instruction N value as an unsigned decimal value. The operations and their N values are listed in the "FPH2 Operation Summary" table in the "Operations" chapter.

The `-mno-custom-<operation>` command-line option instructs GCC to use the software emulation to implement the specified operation (this is the default).

Note: There is no command-line option for the round operation. These switches may be specified multiple times on the command-line; the last one is the one that 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" chapter.

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 the "Nios II SBT" chapter.

```
-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-fixsi=249
-mcustom-floatis=250
-mcustom-fmuls=252
-mcustom-fadds=253
-mcustom-fsubs=254
-mcustom-fdivs=255
```

Related Information

- **Inference** on page 12
- **Operations** on page 10
- **Nios II SBT** on page 17

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

This option was created for FPH1. It bundles up several command-line options into one and allows the GCC linker to choose the pre-compiled Newlib libraries that match the bundled options.

For FPH2, the `-mcustom-fpu-cfg` option is not recommended. Instead it is recommended that the individual options be used and the Newlib library be compiled from source code instead of using the pre-compiled Newlib libraries. This allows Newlib to obtain the benefits of all FPH2 operations that can be inferred by GCC. However, the -mcustom-fpu-cfg option is still available and compatible with the FPH2.

The `-mcustom-fpu-cfg=60-1` option is used when the FPH1 optional division is absent and is equivalent to setting:

```
-mcustom-fmuls=252
-mcustom-fadds=253
-mcustom-fsubs=254
-fsingle-precision-constant
```

The `-mcustom-fpu-cfg=60-2` option is used when the FPH1 optional division is present and is equivalent to setting:

```
-mcustom-fmuls=252
-mcustom-fadds=253
-mcustom-fsubs=254
-mcustom-fdivs=255
-fsingle-precision-constant
```

Related Information

Newlib Documentation page

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 `-fno-math-errno` is specified, GCC maps calls to `sqrtf()` directly to the `fsqrts` custom instruction. Otherwise, GCC adds several instructions after the `fsqrts` custom instruction to check for a NaN result and call the Newlib `sqrtf()` function if the custom instruction returns a NaN result. A NaN result occurs when attempting the square root of a number less than -0. The Newlib sqrtf() function is called just to set the C language errno variable. Typically embedded developers will not want this overhead on calls to `sqrtf()` so it is recommended that `-fno-math-errno` be enabled.

For users of the Nios II SBT, the `-fno-math-errno` option is set by default in the generated makefiles. This can be overridden 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. This option also has other effects and is described in the "-ffinite-math-only" chapter.

Related Information

- **Nios II SBT** on page 17
- **-ffinite-math-only** on page 15
- **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 users of the Nios II SBT, `-fsingle-precision-constant` is not set by default in the generated makefiles. This is contrast to the FPH1 that set this option with the `-mcustom-fpu-cfg` option. The `-fsingle-precision-constant` is not set by default in FPH2 because it can break double-precision code.

Programmers can turn on this switch if it is safe for their code. In general, casting floating-point constants to the "float" type or appending an "f", like `3.14f` are better solutions because they are localized and independent of compiler options.

Related Information

Nios II SBT on page 17

-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."
```

This option is only required if custom instructions are used to implement the transcendental functions (i.e. sin, cos, tan, atan, exp, and log). This option is GCC's way of saying, "I can't imagine that your transcendental floating-point hardware is IEEE compliant, and I want you to admit this before I'll use it." Because transcendental functions are not included with the FPH2, the `-funsafe-math-optimizations` option is not required.

-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 14

-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 Newlib library (C and math) is provided in pre-compiled and source versions. If the GCC linker is passed the `-mcustom-fpu-cfg` option, it will chose a pre-compiled Newlib library with floating-point support. The pre-compiled versions only use a subset of the FPH2 operations (namely add, subtract, multiply, and divide) because the pre-compiled libraries were created for FPH1.

If users can meet their requirements using the available pre-compiled Newlib libraries, they are free to use them. If not, it is recommended that users compile the Newlib libraries from source code with the same `-mcustom-<operation>` options as the rest of their software.

The Newlib `isgreater()`, `isgreaterequal()`, `isless()`, `islessequal()`, and `islessgreater` macros defined in math.h use the normal comparison operators (e.g. <, >=) 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: The following topic discusses floating-point custom instructions. For more information, refer to the *GCC Floating-point Custom Instruction Support Overview* webpage.

Related Information

- -mcustom-<operation> on page 13
- pragmas on page 13
- Newlib Documentation page

- **GCC Floating-point Custom Instruction Support Overview**
  For more information about the *GCC Floating-point Custom Instruction Support Overview*.

altera_nios_custom_instr_floating_point_2.h

This C header file provides `#define` macro declarations for the Newlib math library routines not reliably replaced with custom instructions by GCC 4.7.3. C source code should include this header file to use the custom instruction implementation of these operations.

Related Information

- **GCC 4.7.3 Command Line Options**
- **Newlib Documentation page**

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, the `sw.tcl` performs the following functions:

- Generated makefiles 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.
- Generated makefiles 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 the file `altera_nios_custom_instr_floating_point_2.h` to the BSP include directory. For more information, refer to the "altera_nios_custom_instr_floating_point_2.h" chapter.

Note: The following topic discusses floating-point custom instructions. For more information, refer to the *GCC Floating-point Custom Instruction Support Overview* webpage.

Related Information

- **altera_nios_custom_instr_floating_point_2.h** on page 17
- **Nios II Classic Software Developer's Handbook**
- **Nios II Gen2 Software Developer's Handbook**
- **GCC Floating-point Custom Instruction Support Overview**
  For more information about the *GCC Floating-point Custom Instruction Support Overview*.

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

Date	Version	Changes
May 2015	2015.05.22	Initial release.