Floating-Point Operator v7.1

LogiCORE IP Product Guide

Vivado Design Suite

PG060 December 16, 2020

XILINX®

Table of Contents

Chapter 6: Test Bench

Appendix A: Upgrading

Appendix B: Debugging

Appendix C: Additional Resources and Legal Notices

IP Facts

Introduction

The Xilinx® Floating-Point Operator core provides you with the means to perform floating-point arithmetic on an FPGA. The core can be customized for operation, wordlength, latency and interface.

Features

- Supported operators:
    - Multiply
    - Add/subtract
    - Accumulator
    - Fused multiply-add
    - Divide
    - Square-root
    - Comparison
    - Reciprocal
    - Reciprocal square root
    - Absolute value
    - Natural logarithm
    - Exponential
    - Conversion from floating-point to fixed-point
    - Conversion from fixed-point to floating-point
    - Conversion between floating-point types
    - Unfused multiply-add
    - Unfused multiply-accumulator
    - Accumulator primitive
- Compliance with *IEEE-754 Standard* [Ref 1] (with only minor documented deviations)
- Parameterized fraction and exponent wordlengths for most operators
- Optimizations for speed and latency
- Fully synchronous design using a single clock

LogiCORE IP Facts Table	
Core Specifics	
Supported Device Family[1]	UltraScale+™ UltraScale™ Versal™ ACAP Zynq®-7000 SoC 7 Series
Supported User Interfaces	AXI4-Stream
Resources	Performance and Resource Utilization web page
Provided with Core	
Design Files	Encrypted RTL
Example Design	Not Provided
Test Bench	VHDL
Constraints File	Not Provided
Simulation Model	Encrypted VHDL, C Model
Supported S/W Driver	N/A
Tested Design Flows[2]	
Design Entry	Vivado® Design Suite System Generator for DSP
Simulation	For supported simulators, see the Xilinx Design Tools: Release Notes Guide.
Synthesis	Vivado Synthesis
Support	
Release Notes and Known Issues	AR: 54504
All Vivado IP Change Logs	Master Vivado IP Change Logs: 72775
Xilinx Support web page	

Notes:

1. For a complete listing of supported devices, see the Vivado IP catalog.
2. For the supported versions of third-party tools, see the Xilinx Design Tools: Release Notes Guide.

Chapter 1

Overview

Navigating Content by Design Process

Xilinx® documentation is organized around a set of standard design processes to help you find relevant content for your current development task. This document covers the following design processes:

- **Hardware, IP, and Platform Development:** Creating the PL IP blocks for the hardware platform, creating PL kernels, subsystem functional simulation, and evaluating the Vivado timing, resource and power closure. Also involves developing the hardware platform for system integration. Topics in this document that apply to this design process include:
    - Port Descriptions
    - Clocking
    - Resets
    - Customizing and Generating the Core
    - C Model

Core Overview

The Xilinx Floating-Point Operator core allows a range of floating-point arithmetic operations to be performed on FPGA. The operation is specified when the core is generated, and each operation variant has a common interface. This interface is shown in Figure 2-1.

Unsupported Features

See Standards.

Licensing and Ordering

This Xilinx LogiCORE™ IP module is provided at no additional cost with the Xilinx Vivado® Design Suite under the terms of the Xilinx End User License. Information about this and other Xilinx LogiCORE IP modules is available at the Xilinx Intellectual Property page. For information about pricing and availability of other Xilinx LogiCORE IP modules and tools, contact your local Xilinx sales representative.

Chapter 2

Product Specification

Standards

IEEE-754 Support

The Xilinx® Floating-Point Operator core complies with much of the *IEEE-754 Standard* [Ref 1]. The deviations generally provide a better trade-off of resources against functionality. Specifically, the core deviates in the following ways:

- Non-Standard Wordlengths
- Denormalized Numbers
- Rounding Modes
- Signaling and Quiet NaNs

Non-Standard Wordlengths

The Xilinx Floating-Point Operator core supports a different range of fraction and exponent wordlength than defined in the *IEEE-754 Standard*.

Basic Formats:

- **binary16 (Half Precision Format)** – Uses 16 bits, with an 11-bit fraction and 5-bit exponent.
- **binary32 (Single Precision Format)** – Uses 32 bits, with a 24-bit fraction and 8-bit exponent.
- **binary64 (Double Precision Format)** – Uses 64 bits, with a 53-bit fraction and 11-bit exponent.
- **binary128 (Quadruple Format)** – not supported

Extendable Precision Formats (not available on all operators):

- Uses up to 80 bits.
- Exponent width of 4 to 16 bits.

- Fraction width of 4 to 64 bits

  Note: Limitations apply based on exponent width. See the Vivado® Integrated Design Environment for actual ranges.

Denormalized Numbers

The exponent limits the size of numbers that can be represented. It is possible to extend the range for small numbers using the minimum exponent value (0) and allowing the fraction to become denormalized. That is, the hidden bit b_0 becomes zero such that $b_0.b_1b_2\ldots b_{p-1} < 1$. Now the value is given by:

$$v = (-1)^s 2^{-\left(2^{w_e - 1} - 2\right)} 0.b_1b_2\ldots b_{w_f - 1}$$

These denormalized numbers are extremely small. For example, with single precision the value is bounded $|v| < 2^{-126}$. As such, in most practical calculation they do not contribute to the end result. Furthermore, as the denormalized value becomes smaller, it is represented with fewer bits and the relative rounding error introduced by each operation is increased.

The Xilinx Floating-Point Operator core does not support denormalized numbers for most operators. In FPGAs, the dynamic range can be increased using fewer resources by increasing the size of the exponent (and a 1-bit increase for single precision increases the range by 2^{256}). If necessary, the overall wordlength of the format can be maintained by an associated decrease in the wordlength of the fraction.

To provide robustness, the core treats denormalized operands as zero with a sign taken from the denormalized number. Results that would have been denormalized are set to an appropriately signed zero.

The exception to the above rules is the absolute value operator, which propagates denormalized operands to the output.

The support for denormalized numbers cannot be switched off on some processors. Therefore, there might be very small differences between values generated by the Floating-Point Operator core and a program running on a conventional processor when numbers are very small. If such differences must be avoided, the arithmetic model on the conventional processor should include a simple check for denormalized numbers. This check should set the output of an operation to zero when denormalized numbers are detected to correctly reflect what happens in the FPGA implementation.

Rounding Modes

Only the default rounding mode, Round to Nearest (as defined by the *IEEE-754 Standard* [Ref 1]), is supported on most operators. This mode is often referred to as Round to Nearest Even, as values are rounded to the nearest representable value, with ties rounded to the nearest value with a zero least significant bit. The accumulator operator only supports

Round Towards Zero. The float-to-fixed operator uses Round to Nearest which differs from the behavior of the C language when casting floating-point values to integers.

Signaling and Quiet NaNs

The *IEEE-754 Standard* requires provision of Signaling and Quiet NaNs. However, the Xilinx Floating-Point Operator core treats all NaNs as Quiet NaNs. When any NaN is supplied as one of the operands to the core, the result is a Quiet NaN, and an invalid operation exception is not raised (as would be the case for signaling NaNs). The exceptions to this rule are floating-point to fixed-point conversion and the absolute value operator. For detailed information of the floating-point to fixed-point conversion, see the behavior of INVALID_OP. For the absolute value operator, Signaling NaNs are propagated from input to output.

Accuracy of Results

Compliance to the *IEEE-754 Standard* requires that elementary arithmetic operations produce results accurate to half of one Unit in the Last Place (ULP). The Xilinx Floating-Point Operator satisfies this requirement for the multiply, add/subtract, fused multiply-add, divide, square-root and conversion operators.

- The reciprocal, reciprocal square-root, logarithm and exponential operators produce results which are accurate to one ULP. The accuracy of the accumulator operator is variable. See Accumulator Design Guidelines. For half precision format, the reciprocal and reciprocal square root operators are accurate to one half ULP.
- The unfused multiply-add and unfused multiply-acc implementations using DSP58 incur rounding after both the mult and add/acc stages and therefore have a minimum accuracy of 1 ULP.

Performance

Latency

The latency of most operators can be set between 0 and a maximum value that is dependent upon the parameters chosen. The maximum latency of the Floating-Point Operator core for all operators can be found on the Vivado Integrated Design Environment (IDE).

Note: The accumulator operator has a minimum latency of 1 clock cycle.

The maximum latency of the divide and square root operations is Fraction Width + 4, and for compare operation it is two cycles. The float-to-float conversion operation is three cycles when either fraction or exponent width is being reduced; otherwise it is two cycles. It is two cycles, even when the input and result widths are the same, as the core provides conditioning in this situation. For more information, see Cycles per Operation.

Resource Utilization

For details about performance, visit Performance and Resource Utilization.

Port Descriptions

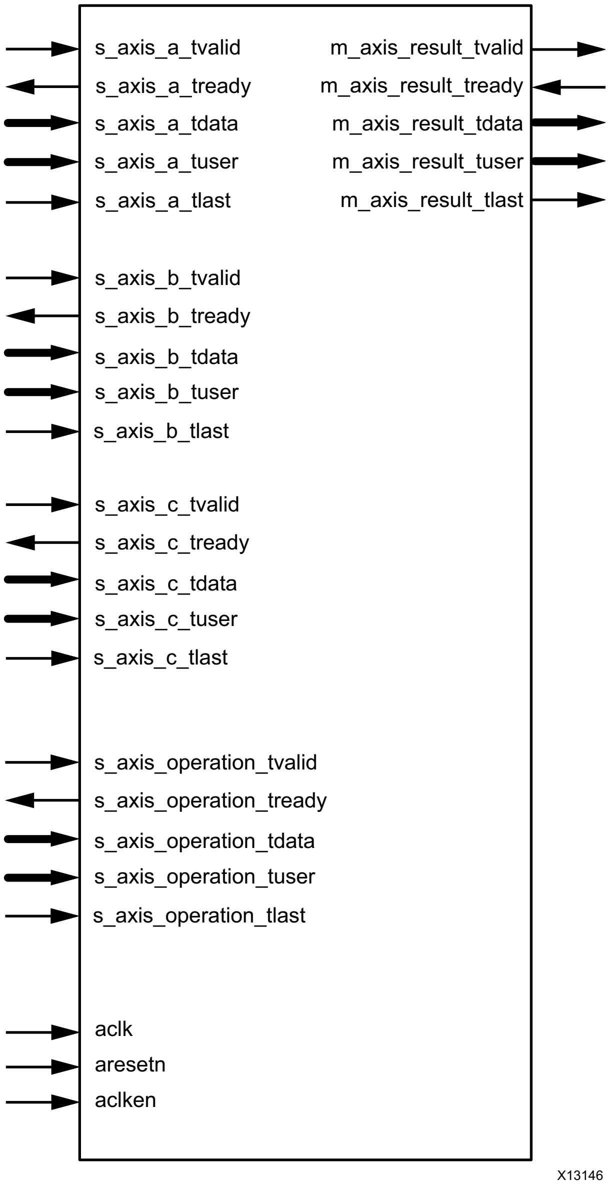

Figure 2-1: **Core Schematic Symbol**

The ports employed by the core are shown in Figure 2-1. They are described in more detail in Table 2-1. All control signals are active-High with the exception of `aresetn`.

Table 2-1: **Core Signal Pinout**

Name	Direction	Description
aclk	Input	Rising-edge clock
aclken	Input	Active-High clock enable (optional)

Table 2-1: **Core Signal Pinout *(Cont'd)***

Name	Direction	Description
aresetn	Input	Active-Low synchronous clear (optional), always takes priority over `aclken`). This signal must be asserted for a minimum of 2 clock cycles.
s_axis_a_tvalid	Input	TVALID for channel A
s_axis_a_tready	Output	TREADY for channel A
s_axis_a_tdata	Input	TDATA for channel A. See TDATA Packing for internal structure
s_axis_a_tuser	Input	TUSER for channel A
s_axis_a_tlast	Input	TLAST for channel A
s_axis_b_tvalid	Input	TVALID for channel B
s_axis_b_tready	Output	TREADY for channel B
s_axis_b_tdata	Input	TDATA for channel B. See TDATA Packing for internal structure
s_axis_b_tuser	Input	TUSER for channel B
s_axis_b_tlast	Input	TLAST for channel B
s_axis_c_tvalid	Input	TVALID for channel C
s_axis_c_tready	Output	TREADY for channel C
s_axis_c_tdata	Input	TDATA for channel C. See TDATA Packing for internal structure
s_axis_c_tuser	Input	TUSER for channel C
s_axis_c_tlast	Input	TLAST for channel C
s_axis_operation_tvalid	Input	TVALID for channel OPERATION
s_axis_operation_tready	Output	TREADY for channel OPERATION
s_axis_operation_tdata	Input	TDATA for channel OPERATION. See TDATA Packing for internal structure
s_axis_operation_tuser	Input	TUSER for channel OPERATION
s_axis_operation_tlast	Input	TLAST for channel OPERATION
m_axis_result_tvalid	Output	TVALID for channel RESULT
m_axis_result_tready	Input	TREADY for channel RESULT
m_axis_result_tdata	Output	TDATA for channel RESULT. See TDATA Subfield for internal structure
m_axis_result_tuser	Output	TUSER for channel RESULT
m_axis_result_tlast	Output	TLAST for channel RESULT

All AXI4-Stream port names are lower case, but for ease of visualization, upper case is used in this document when referring to port name suffixes, such as TDATA or TLAST.

A Channel (s_axis_a_tdata)

Operand A input.

B Channel (s_axis_b_tdata)

Operand B input.

C Channel (s_axis_c_tdata)

Operand C input.

aclk

All signals are synchronous to the aclk input.

aclken

When aclken is deasserted, the clock is disabled, and the state of the core and its outputs are maintained.

Note: aresetn takes priority over aclken.

aresetn

When aresetn is asserted, the core control circuits are synchronously set to their initial state. Any incomplete results are discarded, and m_axis_result_tvalid is not generated for them. While aresetn is asserted m_axis_result_tvalid is synchronously deasserted. The core is ready for new input one cycle after aresetn is deasserted, at which point slave channel tvalids are asserted. aresetn takes priority over aclken. If aresetn is required to be gated by aclken, then this can be done externally to the core.

Note: See the warning described in Non-Blocking Mode.

IMPORTANT: *aresetn must be driven low for a minimum of two clock cycles to reset the core.*

Operation Channel (s_axis_operation_tdata)

The operation channel is present when add and subtract operations are selected together, or when a programmable comparator is selected. The operations are binary encoded as specified in Table 2-2.

Table 2-2: **Encoding of s_axis_operation_tdata**

FP Operation		s_axis_operation_tdata(5:0)
Add		000000
Subtract		000001
Compare (Programmable)	Unordered[(1)]	000100
	Less Than	001100
	Equal	010100
	Less Than or Equal	011100
	Greater Than	100100
	Not Equal	101100
	Greater Than or Equal	110100

Notes:

1. An unordered comparison returns TRUE when either (or both) of the operands are NaN, indicating that the operands' magnitudes cannot be put in size order.

Result Channel (m_axis_result_tdata)

If the operation is compare, then the valid bits within the result depend upon the compare operation selected. If the compare operation is one of those listed in Table 2-2, then only the least significant bit of the result indicates whether the comparison is TRUE or FALSE. If the operation is condition code, then the result of the comparison is provided by 4 bits using the encoding summarized in Table 2-3.

Table 2-3: **Condition Code Summary**

Compare Operation	m_axis_result_tdata(3:0)				Result
	3	**2**	**1**	**0**	
Programmable				0	A OP B = FALSE
				1	A OP B = TRUE
Condition Code	Unordered	>	<	EQ	Meaning
	0	0	0	1	A = B
	0	0	1	0	A < B
	0	1	0	0	A > B
	1	0	0	0	A, B or both are NaN.

The following flag signals provide exception information. Additional detail on their behavior can be found in the *IEEE-754 Standard*. The exception flags are not presented as discrete signals in Floating-Point Operator v7.1, but instead are provided in the RESULT channel `m_axis_result_tuser` subfield. For more details, see Output Result Channel.

The accumulator operator adds two non-standard exception flags: Accumulator Input Overflow, and Accumulator Overflow. For more information about these flags, see Accumulator Design Guidelines.

UNDERFLOW

Underflow is signaled when the operation generates a non-zero result which is too small to be represented with the chosen precision. The result is set to zero. Underflow is detected after rounding.

Note: A number that becomes denormalized before rounding is set to zero and underflow signaled.

OVERFLOW

Overflow is signaled when the operation generates a result that is too large to be represented with the chosen precision. For most operators, the output is set to a correctly signed ∞.

Due to its different rounding mode, the accumulator operator sets the output to the target format's largest finite number with the sign of the pre-rounded result.

INVALID_OP

Invalid general-computational or signaling-computational operations are signaled when the operation performed is invalid. According to the *IEEE-754 Standard* [Ref 1], the following are invalid operations:

1. Any operation on a signaling NaN. (This is not relevant to the core as all NaNs are treated as Quiet NaNs).
2. Addition or subtraction of infinite values where the sign of the result cannot be determined. For example, magnitude subtraction of infinities such as $(+\infty) + (-\infty)$.
3. Multiplication, or fused multiply-add, where $0 \times \infty$.
4. Division where 0/0 or ∞/∞.
5. Square root if the operand is less than zero. A special case is sqrt(-0), which is defined to be -0 by the *IEEE-754 Standard*.
6. When the input of a conversion precludes a faithful representation that cannot otherwise be signaled (for example NaN or infinity).
7. Logarithm if the input is less than 0. A special case is log(-0) which is defined to be $-\infty$.

When an invalid operation occurs, the associated result is a Quiet NaN. In the case of floating-point to fixed-point conversion, NaN and infinity raise an invalid operation exception. If the operand is out of range, or an infinity, then an overflow exception is raised. By analyzing the two exception signals it is possible to determine which of the three types of operand was converted. (See Table 2-4.)

Table 2-4: **Invalid Operation Summary**

Operand	Invalid Operation	Overflow	Result
+ Out of Range	0	1	011...11
- Out of Range	0	1	100...00
+ Infinity	1	1	011...11
- Infinity	1	1	100...00
NaN	1	0	100...00

When the operand of a Floating-point to fixed-point conversion is a NaN, the result is set to the most negative representable number. When the operand is infinity or an out-of-range floating-point number, the result is saturated to the most positive or most negative number, depending upon the sign of the operand.

Note: Floating-point to fixed-point conversion does not treat a NaN as a Quiet NaN, because NaN is not representable within the resulting fixed-point format, and so can only be indicated through an invalid operation exception.

The absolute value operator does not signal an invalid operation when a Signaling NaN is input, as it is not a general computational or a signaling computational operation.

Note: The fused multiply-add operator does not signal an invalid operation when $0 \times \infty$ + Quiet NaN is performed.

DIVIDE_BY_ZERO

DIVIDE_BY_ZERO is asserted when a divide operation is performed where the divisor is zero and the dividend is a finite non-zero number. The result in this circumstance is a correctly signed infinity.

DIVIDE_BY_ZERO is asserted when a logarithm operation is performed where the operand is zero. The result in this circumstance is negative infinity.

Designing with the Core

This chapter includes guidelines and additional information to make designing with the core easier.

General Design Guidelines

The floating-point and fixed-point representations employed by the core are described in Floating-Point Number Representation and Fixed-Point Number Representation.

Floating-Point Number Representation

The core employs a floating-point representation that is a generalization of the *IEEE-754 Standard* [Ref 1] to allow for non-standard sizes. When standard sizes are chosen, the format and special values employed are identical to those described by the *IEEE-754 Standard*.

Two parameters have been adopted for the purposes of generalizing the format employed by the Floating-Point Operator core. These specify the total format width and the width of the fractional part. For standard single precision types, the format width is 32 bits and fraction width 24 bits. In the following description, these widths are abbreviated to w and w_f, respectively.

A floating-point number is represented using a sign, exponent, and fraction (which are denoted as 's,' 'E,' and $b_0.b_1b_2...b_{w_f-1}$, respectively).

The value of a floating-point number is given by: $v = (-1)^s 2^E b_0.b_1b_2...b_{w_f-1}$

The binary bits, b_i, have weighting 2^{-i}, where the most significant bit b_0 is a constant 1. As such, the combination is bounded such that $1 \leq b_0.b_1b_2...b_{p-1} < 2$ and the number is said to be normalized. To provide increased dynamic range, this quantity is scaled by a positive or negative power of 2 (denoted here as E). The sign bit provides a value that is negative when $s = 1$, and positive when $s = 0$.

The binary representation of a floating-point number contains three fields as shown in Figure 3-1.

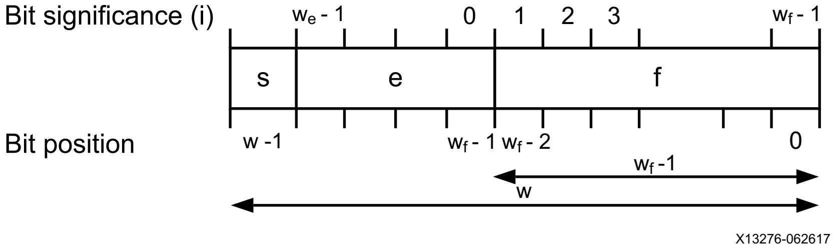

Figure 3-1: **Bit Fields within the Floating-Point Representation**

As b_0 is a constant, only the fractional part is retained, that is, $f = b_1 \ldots b_{w_f - 1}$. This requires only $w_f - 1$ bits. Of the remaining bits, one bit is used to represent the sign, and $w_e = w - w_f$ bits represent the exponent.

The exponent field, e, employs a biased unsigned integer representation, whose value is given by:

$$e = \sum_{i=0}^{w_e - 1} e_i 2^i$$

The index, i, of each bit within the exponent field is shown in Figure 3-1.

The signed value of the exponent, E, is obtained by removing the bias, that is,

$$E = e - (2^{w_e - 1} - 1).$$

In reality, w_f is not the wordlength of the fraction, but the fraction with the hidden bit, b_0, included. This terminology has been adopted to provide commonality with that used to describe fixed-point parameters (as employed by Xilinx® System Generator™ for DSP).

Special Values

Several values for s, e and f have been reserved for representing special numbers, such as Not a Number (NaN), Infinity (∞), Zero (0), and denormalized numbers (see Denormalized Numbers for an explanation of the latter). These special values are summarized in Table 3-1.

Table 3-1: **Special Values**

Symbol for Special Value	s Field	e Field	f Field
NaN	don't care	$2^{w_e - 1}$ -1 (that is, e = 11...11)	Any non-zero field. For results that are NaN the most significant bit of fraction is set (that is, f = 10...00)
±∞	sign of ∞	$2^{w_e - 1}$ -1 (that is, e = 11...11)	Zero (that is, f = 00...00)

Table 3-1: **Special Values *(Cont'd)***

Symbol for Special Value	s Field	e Field	f Field
± 0	sign of 0	0	Zero (that is, $f = 00...00$)
denormalized	sign of number	0	Any non-zero field

In Table 3-1 the sign bit is undefined when a result is a NaN. The core generates NaNs with the sign bit set to 0 (that is, positive). Also, infinity and zero are signed. Where possible, the sign is handled in the same way as finite non-zero numbers. For example, $-0 + (-0) = -0$, $-0 + 0 = 0$ and $-\infty + (-\infty) = -\infty$. A meaningless operation such as $-\infty + \infty$ raises an invalid operation exception and produces a NaN as a result.

Fixed-Point Number Representation

For the purposes of fixed-point to floating-point conversion, a fixed-point representation is adopted that is consistent with the signed integer type used by Xilinx System Generator for DSP. Fixed-point values are represented using a twos complement number that is weighted by a fixed power of 2. The binary representation of a fixed-point number contains three fields as shown in Figure 3-2 (although it is still a weighted twos complement number).

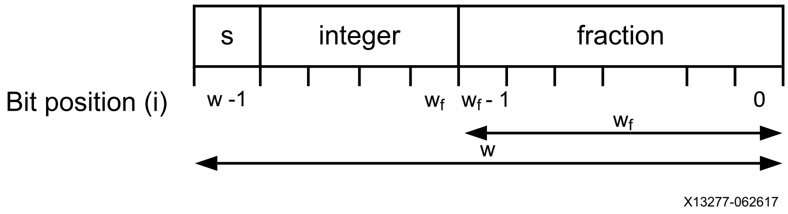

Figure 3-2: **Bit Fields within the Fixed-Point Representation**

In Figure 3-2, the bit position has been labeled with an index i. Based upon this, the value of a fixed-point number is given by:

$$v = -(b_{w-1})2^{w-1-w_f} + b_{w-2}\ldots b_{w_f}.b_{w_f-1}\ldots b_1 b_0$$

$$= -(b_{w-1})2^{w-1-w_f} + \sum_{i=0}^{w-2} 2^{i-w_f} b_i$$

For example, a 32-bit signed integer representation is obtained when a total width of 32 and a fraction width of 0 are specified. Round to Nearest is employed within the conversion operations. To provide for the sign bit, the width of the integer field must be at least 1, requiring that the fractional width be no larger than w – 1.

The fixed-to-float operator also has the option to perform 32-bit and 64-bit signed and unsigned integer conversions to convert standard software integer data formats to floating point.

Accumulator Design Guidelines

Configuring the Accumulator

The accumulator operator has been implemented as a floating-point wrapper around a fixed point accumulator to reduce resources and to allow a throughput of one sample per clock cycle. Refer to *An FPGA-specific Approach to Floating-Point Accumulation and Sum-of-Products* [Ref 17] for more information. Three parameters are required to configure the accumulator:

- **Input MSB** – The MSB of the largest number that can be accepted.
- **LSB** – The LSB of the smallest number that can be accepted. It is also the LSB of the accumulated result.
- **MSB** – The MSB of the largest result. It can be up to 54 bits greater than the Input MSB.

These values can be set to fully support the chosen IEEE-754 format, as shown in Table 3-2, allowing the accumulator to process any floating-point number and accumulate without introducing any round-off error. For example, to fully accommodate Single Precision, the Input MSB should be set to 127 and the LSB set to -149.

Table 3-2: **MSB and LSB of the Largest and Smallest IEEE-754 Floating-Point Numbers**

Format	MSB	LSB
Single	2^{127}	2^{-149}
Double	2^{1023}	2^{-1074}
Custom	$2^{(exponent\_width-1)}-1$	$2^{(1-MSB)-(fractional\_width-1)}$

Resource usage can be reduced if these parameters are set to match the bounds on the dataset that is used with the accumulator. For example, if the largest value that will be accumulated is 100,000 then the MSB can be set to 17, substantially reducing the width of the accumulator.

The LSB controls the accuracy of the accumulator. Input values with an LSB smaller than the LSB of the accumulator are truncated (Round Towards Zero) introducing a maximum error of 2^{LSB-1} per accumulation. In the worst case, the lower $log_2(n)$ bits of the accumulator are incorrect after n such accumulations. If accuracy to 2^x is required after n accumulations then the LSB needs to be set to $x-\lceil \log_2(N) \rceil$. For example, if accuracy to 2^{-16} is required after 1000 accumulations, the LSB needs to be set to $-16-\lceil \log_2(1000) \rceil = -26$.

The MSB of the accumulator sets the maximum value that can be accumulated. The set value can be up to 54 bits greater than the Input MSB, which allows one number to be accumulated every clock cycle for one year at 400 MHz. If the MSB is set to be greater than the maximum value of the IEEE-754 format then the result can cause an IEEE-754 overflow unless sufficient subtractions occur to bring it back into range (this is for a positive accumulated value. For a negative accumulated value, sufficient additions need to occur.)

Denormalized Numbers

The accumulator is consistent with the other operators in its handling of denormalized values. Denormalized numbers seen on the input are flushed to zero. Denormalized numbers on the output are flushed to zero and the Underflow flag is set. However, denormalized numbers generated in the accumulator are retained within the accumulator to maintain accuracy.

Exceptions

The accumulator handles exceptions in order shown in Table 3-3. All exceptions except for OVERFLOW and UNDERFLOW are unrecoverable.

Note: OVERFLOW and UNDERFLOW represent the state of the accumulated value after it has been converted to a floating-point number. Following operations might bring the accumulator back into a valid range so these exceptions are handled on a per-output basis. All other exceptions represent the state of the accumulator itself and recovery is only possible by ending the accumulation.

After an unrecoverable exception occurs, the output value and flags remain set until a new accumulation is started. The output value and flags can subsequently change if an exception above it in the table occurs.

Table 3-3: **Accumulator Exception Handling**

Exception	Output Value	Flags	Notes
NaN Summand	NaN	None	The output is a Quiet NaN, with the sign bit set to 0.
Summand MSB > Input MSB	NaN	ACCUM_INPUT_OVERFLOW	Infinities cannot trigger this exception
Accumulated value MSB > MSB	NaN	ACCUM_OVERFLOW	
+∞ and -∞	NaN	INVALID_OP	
+∞ or -∞	+∞ or -∞	None	
Result larger than IEEE-754 format can represent	±MAX	OVERFLOW	As truncation is used (Round To 0) then the maximum value for the floating-point format is returned, not ∞.
Denormalized result	±0	UNDERFLOW	

Example:

- An +∞ summand is seen on the A channel.

- The accumulator now outputs +∞ with no flags until a new accumulation starts, unless a higher priority exception occurs.
- From this point on OVERFLOW and UNDERFLOW are impossible, but any of the exceptions above it in the table can still occur.

- Some time later, but within the same block, a -∞ summand is seen on the A channel.
    - The output value now changes to NaN and INVALID_OP is set as both +∞ and -∞ have been accumulated.
- Some time later, but within the same block, a NaN summand is seen on the A channel.
    - The output value remains as NaN but INVALID_OP is cleared.
- No other exceptions are possible.

Starting a New Accumulation

A floating-point number on the A channel is the first in a new accumulation when either of the following conditions are true:

- It is the first summand after `aresetn` has been asserted and released.
- It is the first summand after `s_axis_a_tlast` has been asserted on a valid AXI transfer.

When a new accumulation starts, the exceptions are cleared, the accumulator register is set to zero, and the new summand is combined (added or subtracted) with zero.

Accumulator Primitive

The Accumulator Primitive operation is an alternative accumulator operation. The Accumulator Primitive can be considered as an Add/Subtract operator with the result fed back such that the output P = P +/- A. Due to the difference in internal precision, this implementation can yield different results to the Accumulator operator and so is considered to be a separate operator rather than as an alternative use of resources.

Clocking

The Floating-Point Operator core uses a single clock, called `aclk`. All input and output interfaces and internal state are subject to this single clock.

Resets

The Floating-Point Operator core uses a single, optional, reset input called `aresetn`. This signal is active-Low and must be asserted for a minimum of two clock cycles to ensure correct operation. `aresetn` is a global synchronous reset which resets all control states in the core; all data in transit through the core is lost when `aresetn` is asserted.

Protocol Description

AXI4-Stream Considerations

The conversion to AXI4-Stream interfaces brings standardization and enhances interoperability of Xilinx IP LogiCORE™ solutions. Other than general control signals such as `aclk`, `aclken` and `aresetn`, all inputs and outputs to and from the Floating-Point Operator core are conveyed using AXI4-Stream channels. A channel consists of TVALID and TDATA always, plus several optional ports and fields. In the Floating-Point Operator, the optional ports supported are TREADY, TLAST and TUSER. Together, TVALID and TREADY perform a handshake to transfer a message, where the payload is TDATA, TUSER and TLAST. The Floating-Point Operator operates on the operands contained in the TDATA fields and outputs the result in the TDATA field of the output channel. The Floating-Point Operator does not use TUSER and TLAST inputs as such, but the core provides the facility to convey these fields with the same latency as for TDATA. This facility is expected to ease use of the Floating-Point Operator in a system. For example, the Floating-Point Operator might be operating on streaming packetized data. In this example, the core could be configured to pass the TLAST of the packetized data channel, thus saving the system designer the effort of constructing a bypass path for this information. For further details on AXI4-Stream interfaces see [Ref 15] and [Ref 16].

Note: The accumulator does use TLAST as an input. For more information, see **TLAST in the Accumulator and Accumulator Primitive Operators**.

Basic Handshake

Figure 3-3 shows the transfer of data in an AXI4-Stream channel. TVALID is driven by the source (master) side of the channel and TREADY is driven by the receiver (slave). TVALID indicates that the value in the payload fields (TDATA, TUSER and TLAST) is valid. TREADY indicates that the slave is ready to receive data. When both TVALID and TREADY are TRUE in a cycle, a transfer occurs. The master and slave set TVALID and TREADY respectively for the next transfer appropriately.

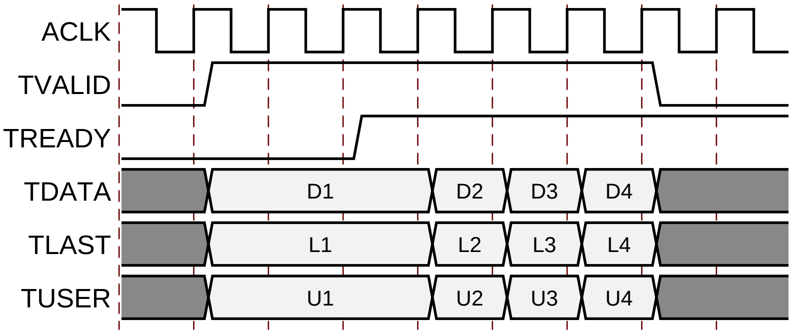

Figure 3-3: **Data Transfer in an AXI4-Stream Channel**

Non-Blocking Mode

The term Non-Blocking means that lack of data on one input channel does not block the execution of an operation if data is received on another input channel. The full flow control of AXI4-Stream is not always required. Blocking or Non-Blocking behavior is selected using the Flow Control parameter or Vivado® Integrated Design Environment field. The core supports a Non-Blocking mode in which the AXI4-Stream channels do not have TREADY, that is, they do not support back pressure. The choice of Blocking or Non-Blocking applies to the whole core, not each channel individually. Channels still have the non-optional TVALID signal, which is analogous to the New Data (ND) signal on many cores prior to the adoption of AXI4-Stream interfaces. Without the facility to block dataflow, the internal implementation is much simplified, so fewer resources are required for this mode.

RECOMMENDED: *This mode is recommended when moving to this core version from a pre-AXI4-Stream core with minimal change.*

When all of the present input channels receive an active TVALID, an operation is validated and the output TVALID (suitably delayed by the latency of the core) is asserted to qualify the result. Operations occur on every enabled clock cycle and data is presented on the output channel payload fields regardless of TVALID. This is to allow a minimal migration from previous core versions. Figure 3-4 shows the Non-Blocking behavior for a case of an adder with latency of one cycle.

IMPORTANT: *For performance, `aresetn` is registered internally, which delays its action by a clock cycle. The effect of this is that any transaction input in the cycle following the de-assertion of `aresetn` is reset by the action of `aresetn`, resulting in an output data value of zero. `m_axis_result_tvalid` is also inactive for this cycle.*

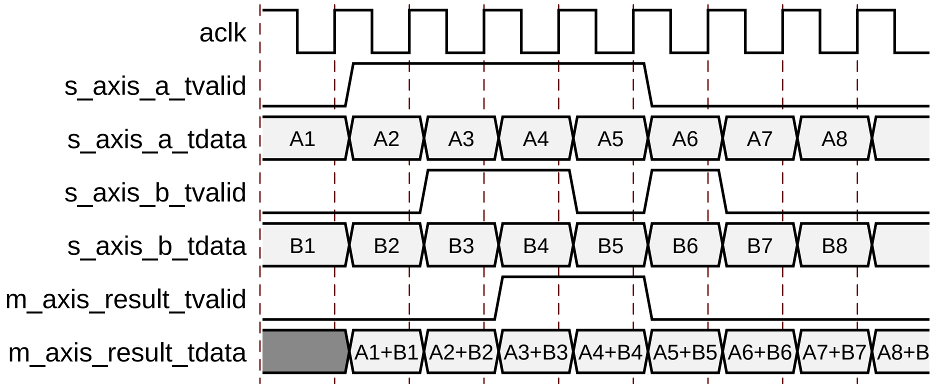

Figure 3-4: **Non-Blocking Mode**

Blocking Mode

The term Blocking means that operation execution does not occur until fresh data is available on all input channels. The full flow control of AXI4-Stream aids system design because the flow of data is self-regulating. Data loss is prevented by the presence of back pressure (TREADY), so that data is only propagated when the downstream datapath is ready to process the data.

The Floating-Point Operator has one, two or three input channels and one output channel. When all input channels have validated data available, an operation occurs and the result becomes available on the output. If the output is prevented from off-loading data because TREADY is low then data accumulates in the output buffer internal to the core. When this output buffer is nearly full the core stops further operations. This prevents the input buffers from off-loading data for new operations so the input buffers fill as new data is input. When the input buffers fill, their respective TREADYs are deasserted to prevent further input. This is the normal action of back pressure.

The inputs are tied in the sense that each must receive validated data before an operation is prompted. Therefore, there is an additional blocking mechanism, where at least one input channel does not receive validated data while others do. In this case, the validated data is stored in the input buffer of the channel.

After a few cycles of this scenario, the buffer of the channel receiving data fills and TREADY for that channel is deasserted until the starved channel receives some data. Figure 3-5 shows both blocking behavior and back pressure for the case of an adder. The first data on channel A is paired with the first data on channel B, the second with the second and so on. This demonstrates the 'blocking' concept. The diagram further shows how data output is delayed not only by latency, but also by the handshake signal `m_axis_result_tready`. This is 'back pressure'. Sustained back pressure on the output along with data availability on the inputs eventually leads to a saturation of the core buffers, leading the core to signal that it can no longer accept further input by deasserting the input channel TREADY signals. The minimum latency in this example is 2 cycles, but it should be noted that in Blocking operation latency is not a useful concept. Instead, as the diagram shows, the important idea is that each channel acts as a queue, ensuring that the first, second, third data samples on each channel are paired with the corresponding samples on the other channels for each operation.

Note: The core buffers have a greater capacity than implied by the diagram.

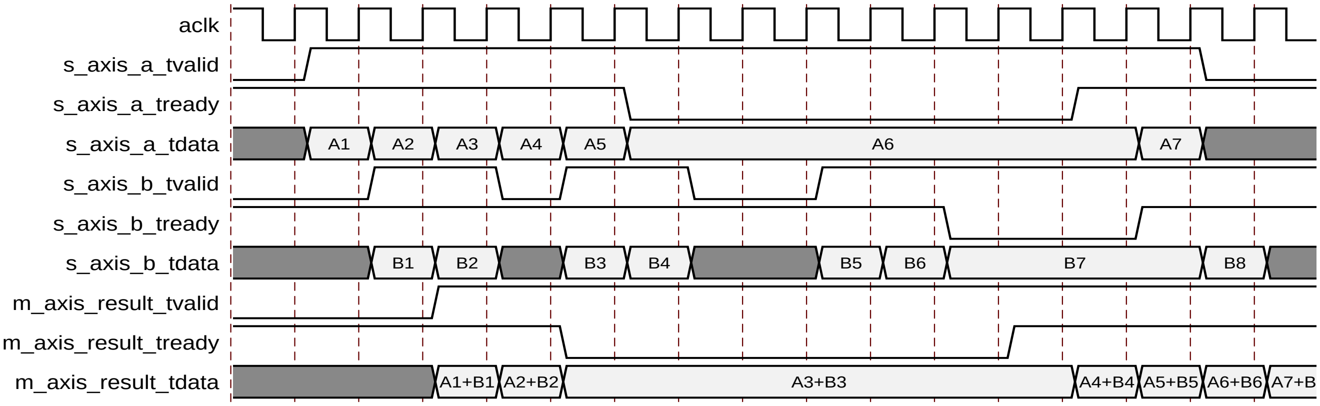

Figure 3-5: **Blocking Mode**

TDATA Packing

Fields within an AXI4-Stream interface are not given arbitrary names. Normally, information pertinent to the application is carried in the TDATA field. To ease interoperability with byte-oriented protocols, each subfield within TDATA which could be used independently is first extended, if necessary, to fit a bit field which is a multiple of 8 bits. For example, say the Floating-Point Operator is configured to have an A operand with a custom precision of 11 bits (5 exponent and 6 mantissa bits). The operand would occupy bits (10:0). Bits (15:11) would be ignored. The bits added by byte orientation are ignored by the core and do not result in additional resource use.

A, B, and C Input Channels

TDATA Structure for A, B, and C Channels

Input channels A, B, and C carry data for use in calculations in their TDATA fields. See Figure 3-6.

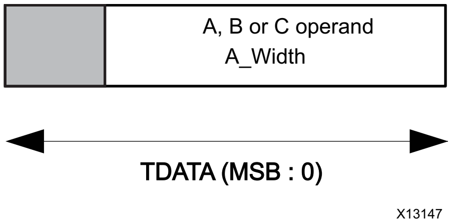

Figure 3-6: **TDATA Structure for A, B, and C Channels**

Figure 3-7 illustrates how the previous example of a custom precision input with 11 bits maps to the TDATA channel.

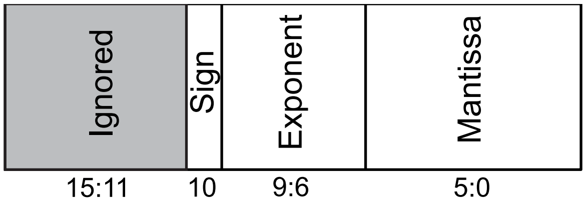

Figure 3-7: **Custom Precision Input (11 bits) Mapped to TDATA Channel**

TDATA Structure for OPERATION Channel

The OPERATION channel exists only when add and subtract operations are selected together, or when a programmable comparator is selected. The binary encoded operation code, as specified in Table 2-2, are 6 bits in length. However, due to the byte-oriented nature of TDATA, this means that TDATA has a width of 8 bits.

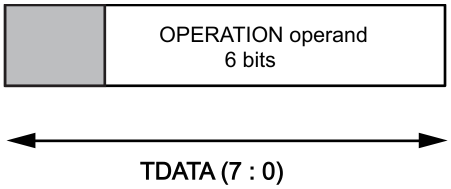

Figure 3-8: **TDATA Structure for OPERATION Channel**

TLAST and TUSER Handling

This section covers TLAST and TUSER handling in a variety of scenarios.

TLAST in All Operators Apart from the Accumulator and Accumulator Primitive Operators

TLAST in AXI4-Stream is used to denote the last transfer of a block of data. The Floating-Point Operator core operates on a per-sample basis where each operation is independent of any other before or after. Because of this, there is no need for TLAST on a Floating-Point Operator core.

The TLAST signal is supported on each channel purely as an optional aid to system design for the scenario in which the data stream being passed through the Floating-Point Operator core does indeed have some packetization, but which is not relevant to the core operation. The facility to pass TLAST removes the burden of matching latency to the TDATA path, which can be variable, through the Floating-Point Operator core.

TLAST in the Accumulator and Accumulator Primitive Operators

TLAST is used in the accumulator to signal the last sample in a block of data. The next sample received after the one with TLAST asserted is loaded into the accumulator to start a fresh accumulation.

On the result channel, TLAST is used to signal the last result in a block of data. The result with TLAST asserted represents the final accumulation of all of the data in the block.

TUSER

TUSER is for ancillary information that qualifies or augments the primary data in TDATA. The TUSER signal is supported on each channel purely as an optional aid to system design for the scenario in which the data stream being passed through the Floating-Point Operator core does indeed have some ancillary field, but which is not relevant to the core operation. The facility to pass TUSER removes the burden of matching latency to the TDATA path, which can be variable, through the Floating-Point Operator core.

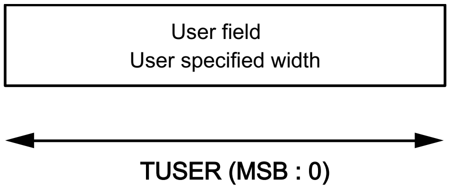

Figure 3-9: **TUSER Structure for A, B, C and OPERATION Channels**

TLAST Options

All Operators Apart from the Accumulator Operator

TLAST for each input channel is optional. When present, each input channel can be passed through the Floating-Point Operator core. When more than one channel has TLAST enabled, each input channel can pass a logical AND or logical OR of the TLASTs input. When no TLASTs are present on any input channel, the output channel does not have TLAST either.

Accumulator Operator

The accumulator has no TLAST options because TLAST is not optional.

TUSER Options

TUSER for each input channel is optional. Each has user-selectable width. These fields are concatenated, without any byte-orientation or padding, to form the output channel TUSER field. The TUSER field from channel A forms the least significant portion of the concatenation, then TUSER from channel B, TUSER from channel C, and TUSER from channel OPERATION.

For example, if channels A and OPERATION both have TUSER subfields with widths of 5 and 8 bits respectively, and no exception flag signals (for example, underflow) are selected, the output TUSER is a suitably delayed concatenation of A and OPERATION TUSER fields, 13 bits wide, with A in the least significant 5 bit positions (4 downto 0).

Output Result Channel

TDATA Subfield

The internal structure of the RESULT channel TDATA subfield depends on the operation performed by the core.

For numerical operations (for example, add, multiply) TDATA contains the numerical result of the operation and is a single floating-point or fixed-point number. The result width is sign-extended to a byte boundary if necessary. This is shown in Figure 3-10.

For Comparator operations, the result is either a 4-bit field (Condition Code) or a single bit indicating TRUE or FALSE. In both cases, the result is zero-padded to a byte boundary, as shown in Figure 3-11.

TUSER Subfield

The TUSER subfield is present if any of the input channels have an (optional) TUSER subfield, or if any of the exception flags (underflow, overflow, invalid operation, divide by zero, Accumulator Input Overflow and Accumulator Overflow) have been selected. The formatting of the TUSER fields is shown in Figure 3-12.

If any field of TUSER is not present, fields in more significant bit positions move down to fill the space. For example, if the overflow exception flag is selected, but the underflow exception flag is not, the overflow exception flag result moves to the least-significant bit position in the TUSER subfield.

No byte alignment is performed on TUSER fields. All fields present are immediately adjacent to one another with no padding between them or at the most significant bit.

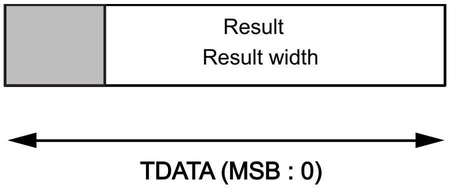

Figure 3-10: **TDATA Structure for Numerical Result Channel**

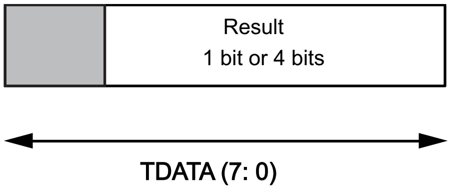

Figure 3-11: **TDATA Structure for Comparator Result Channel**

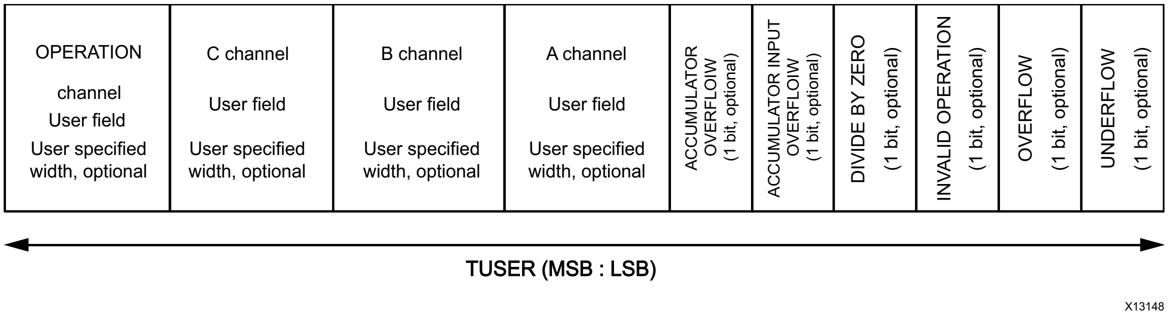

Figure 3-12: **TUSER Structure for Result Channel**

Chapter 4

Design Flow Steps

This chapter describes customizing and generating the core, constraining the core, and the simulation, synthesis and implementation steps that are specific to this IP core. More detailed information about the standard Vivado® design flows and the IP integrator can be found in the following Vivado Design Suite user guides:

- *Vivado Design Suite User Guide: Designing IP Subsystems using IP Integrator* (UG994) [Ref 8]
- *Vivado Design Suite User Guide: Designing with IP* (UG896) [Ref 6]
- *Vivado Design Suite User Guide: Getting Started* (UG910) [Ref 7]
- *Vivado Design Suite User Guide: Logic Simulation* (UG900) [Ref 10]

Customizing and Generating the Core

This section includes information about using Xilinx® tools to customize and generate the core in the Vivado Design Suite.

If you are customizing and generating the core in the Vivado IP Integrator, see the *Vivado Design Suite User Guide: Designing IP Subsystems using IP Integrator* (UG994) [Ref 8] for detailed information. IP Integrator might auto-compute certain configuration values when validating or generating the design. To check whether the values change, see the description of the parameter in this chapter. To view the parameter value, run the `validate_bd_design` command in the Tcl console.

All fields are visible in the IP Integrator. These fields are set automatically:

- A Precision Type
- Exponent Width
- Fraction Width

You can customize the IP for use in your design by specifying values for the various parameters associated with the IP core using the following steps:

1. Select the IP from the IP catalog.
2. Double-click the selected IP or select the Customize IP command from the toolbar or right-click menu.

For details, see the *Vivado Design Suite User Guide: Designing with IP* (UG896) [Ref 6] and the *Vivado Design Suite User Guide: Getting Started* (UG910) [Ref 7].

If you are customizing and generating the core in the Vivado IP integrator, see *Vivado Design Suite User Guide: Designing IP Subsystems Using IP Integrator* (UG994) [Ref 8] for detailed information. Vivado Integrated Design Environment (IDE) might auto-compute certain configuration values when validating or generating the design, as noted in this section. You can view the parameter value after successful completion of the `validate_bd_design` command.

The Floating-Point Operator core provides several tabs with fields to set the parameter values for the particular instantiation required. This section provides a description of each field.

The Vivado IDE allows configuration of the following:

- Core operation
- Wordlength
- Implementation optimizations, such as use of slices
- Optional pins

All Configuration Tabs

All configuration tabs allow the Component Name to be specified.

Component Name

The component name is used as the base name of the output files generated for the core. Names must start with a letter and be composed using the following characters: a to z, 0 to 9, and "_".

Operation Selection Tab

The floating-point operation can be one of the following:

- Add/Subtract
- Accumulators
- Multiply
- Fused Multiply-Add
- Divide
- Reciprocal
- Square-root

- Reciprocal square root
- Absolute value
- Logarithm
- Exponential
- Compare
- Fixed-to-float
- Float-to-fixed
- Float-to-float
- Unfused multiply-add
- Unfused multiply accumulator
- Accumulator primitive

When Add/Subtract, Accumulators, Unfused Multiply-Add, Accumulator Primitive, Unfused Multiply-Accumulator or Fused Multiply-Add is selected, it is possible for the core to perform both operations, or just add or subtract. When both are selected, the operation performed on a particular set of operands is controlled by the `s_axis_operation` channel (with encoding defined in Table 2-2).

When Add/Subtract, Accumulator, Multiply, Fused Multiply-Add, Logarithm or Exponential is selected, the level of slice usage can be specified according to FPGA family as described in the AXI4-Stream Channel Options section.

When Compare is selected, the compare operation can be programmable or fixed. If programmable, then the compare operation performed should be supplied through the `s_axis_operation` channel (with encoding defined in Table 2-2). If a fixed operation is required, then the operation type should be selected.

When Float-to-float conversion is selected, and exponent and fraction widths of the input and result are the same, the core provides a means to condition numbers, that is, convert denormalized numbers to zero, and signaling NaNs to quiet NaNs.

Precision of Inputs and Precision of Results Tabs

These tabs let you specify the precision of the operand and the precision of the result. Availability of the precision of results depends on the configuration selected on the Operation Selection Tab. (The Precision of Results tab is available only when performing an operand conversion: Fixed-to-Float, Float-to-Fixed and Float-to-Float.)

When targeting a Versal ACAP device, unfused multiply-add and unfused multiply-acc operators using the DSP Engines are available. The Multiply and Add/Subtract operators may also target the native floating-point support of the DSP Engines.

The parameters in these tabs define the number of bits used to represent quantities. The type of the operands and results depend on the operation requested. For fixed-point conversion operations, either the operand or result is fixed-point. For all other operations, the output is specified as a floating-point type.

Note: For the condition-code compare operation, `m_axis_result_tdata(3:0)` indicates the result of the comparison operation. For other compare operations `m_axis_result_tdata(0:0)` provides the result.

Table 4-1 defines the general limits of the format widths.

Table 4-1: **General Limits of Width and Fraction Width**

Format Type	Fraction Width		Exponent/Integer Width		Width	
	Min	Max	Min	Max	Min	Max
Floating-Point	4	64	4	16	4	64
Fixed-Point	0	63	1	64	4	64

There are also some further limits for specific cases which are enforced by the Vivado IDE:

- The exponent width (that is, Total Width-Fraction Width) should be chosen to support normalization of the fractional part. This can be calculated using:

  Minimum Exponent Width = ceil [$\log_2$(Fraction Width+3)] + 1

  For example, a 24-bit fractional part requires an exponent of at least 6 bits (for example, {ceil [$\log_2$ (27)]+1}).

- For conversion operations, the exponent width of the floating-point input or output is also constrained by the Total Width of the fixed-point input or output to be a minimum of:

  Minimum Exponent Width = ceil [$\log_2$(Total Width+3)] + 1

  For example, a 32-bit integer requires a minimum exponent of 7 bits.

A summary of the width limits imposed by exponent width is provided in Table 4-2.

Table 4-2: **Summary of Exponent Width Limits**

Floating-Point Fraction Width or Fixed-Point Total Width	Minimum Exponent Width
4 to 5	4
6 to 13	5
14 to 29	6
30 to 61	7
61 to 64	8

Internal Precision Tab

The Internal Precision tab is used only for the Accumulator. The values in these fields may be dynamically updated by the Vivado IDE when other operators are selected, but they may be safely ignored because these values are used only by the Accumulator operator. The tab allows you specify the following:

- Accumulator MSB
- Accumulator LSB
- Input MSB

See Configuring the Accumulator for more information.

Optimizations Tab

Architecture Optimizations

For multiplication (double precision), addition/subtraction and Accumulator operations, it is possible to specify a latency optimized architecture, or speed optimized architecture. The latency optimized architecture offers reduced latency at the expense of increased resources.

Implementation Optimizations

- DSP Slice Usage allows the level of slice multiplier use to be specified.
- Block Memory Usage allows the level of Block Memory use to be specified.

DSP Slice Usage

The level and type of multiplier usage depends upon the operation. See the Vivado IDE for details on how many DSP Slices are used for each configuration.

Block Memory Usage

Block memory usage can be specified for the Exponential operator. Table 4-3 summarizes the options for block memory.

Table 4-3: **Block Memory Usage for Exponential Operator**

Block Memory Usage	Single	Double
No usage	Distributed memory	Distributed memory
Full usage	1 RAMB36	5 RAMB18

A single block memory is also used for the half precision square root, reciprocal and reciprocal square root implementations.

Interface Options Tab

Flow Control Options

These parameters allow the AXI4-Stream interface to be optimized to suit the surrounding system.

- **Flow Control**
  - **Blocking** – When the core is configured to a Blocking interface, it waits for valid data to be available on all input channels before performing a calculation. Back pressure from downstream modules is possible.
  - **NonBlocking** – When the core is configured to use a NonBlocking interface, a calculation is performed on each cycle where all input channel TVALIDs are asserted. Back pressure from downstream modules is not possible.
- **Optimize Goal**
  - **Resources** – This option reduces the logic resources required by the AXI4-Stream interface, at the expense of maximum achievable clock frequency.
  - **Performance** – This option allows maximum performance, at the cost of additional logic required to buffer data in the event of back pressure from downstream modules.
- **RESULT channel has TREADY**
  - Unchecking this option removes TREADY signals from the RESULT channel, disabling the ability for downstream modules to signal back pressure to the Floating-Point Operator core and upstream modules.

Latency and Rate Configuration

This parameter describes the number of cycles between an operand input and result output. The latency of all operators can be set between 0 and a maximum value that is dependent upon the parameters chosen. IP cores that are intended for many applications, such as the Floating-Point Operator core, are designed to run as fast as the DSP primitives. To achieve this, the fabric logic must be heavily pipelined, leading to relatively high latency. For designs which run at a relatively low frequency, the latency can be reduced while timing can still be met. For instance, if the fully pipelined design with latency=12 can meet timing at 400 MHz, then if the system clock in the user design is 70MHz, for example, then latency can likely be reduced to six and timing can still be met. However, the relationship between latency and achieveable clock speed is not linear. This is because the amount of logic between register stages is roughly the same, so if one register is removed, the achieveable clock frequency drops considerably because one register to register path now has almost double the amount of logic as other register-to-register delays. The advantage of reducing latency when clock speed allows is that the result appears sooner. There is no significant gain in resources other than an inevitable reduction in registers for which there is an ample supply. For the previous example, Xilinx recommends using latency=12 if the system clock is > 0.5 * value in the resource and performance section on the Xilinx webpage for the IP core. Other latency values of use are 6, 3, 2 and 1, though minimum may apply due to FIFOs and feedback stages. Therefore, there is not much use in setting latency to 11 for this reason.

Note: The Accumulator operator has a minimum latency of 1.

Cycles per Operation

The Cycles per Operation Vivado IDE parameter describes the minimum number of cycles that must elapse between inputs. This rate can be specified. A value of 1 allows operands to be applied on every clock cycle, and results in a fully-parallel circuit. A value greater than 1 enables hardware reuse. The resources consumed by the core reduces as the number of cycles per operation is increased. A value of 2 approximately halves the resources used. A fully sequential implementation is obtained when the value is equal to Fraction Width+1 for the square-root operation, and Fraction Width+2 for the divide operation.

Control Signals

Pins for the following global signals are optional:

- **ACLKEN** – Active-High clock enable.
- **ARESETn** – Active-Low synchronous reset. Must be driven low for a minimum of two clock cycles to reset the core.

Optional Output Fields

The following exception signals are optional and are added to `m_axis_result_tuser` when selected:

- UNDERFLOW, OVERFLOW, INVALID_OPERATION, DIVIDE_BY_ZERO, ACCUM_OVERFLOW (accumulator only) and ACCUM_INPUT_OVERFLOW (accumulator only)
- See TLAST and TUSER Handling for information on the internal packing of the exception signals in `m_axis_result_tuser`.

AXI4-Stream Channel Options

The following sections allow configuration of additional AXI4-Stream channel features:

- **A Channel Options**
  - Enables TLAST and TUSER input fields for the A operand channel, and allows definition of the TUSER field width.
- **B Channel Options**

    - Enables TLAST and TUSER input fields for the B operand channel (when present), and allows definition of the TUSER field width.
- **C Channel Options**
    - Enables TLAST and TUSER input fields for the C operand channel (when present), and allows definition of the TUSER field width.
- **OPERATION Channel Options**
    - Enables TLAST and TUSER input fields for the OPERATION channel (when present), and allows definition of the TUSER field width.
- **Output TLAST Behavior**
    - When at least one TLAST input is present on the core, this option defines how the `m_axis_result_tlast` signal should be generated. Options are available to pass any of the input TLAST signals without modification, or to logically OR or AND all input TLASTs.

User Parameters

Table 4-4 shows the relationship between the fields in the Vivado IDE and the User Parameters (which can be viewed in the Tcl Console).

Table 4-4: **Vivado IDE Parameter to User Parameter Relationship**

Vivado IDE Parameter/Value[1]	User Parameter/Value[1]	Default Value
Operation Selection	operation_type	Add_Subtract
Absolute Value	Absolute	
Accumulator	Accumulator	
Add/Subtract	Add_Subtract	
Compare	Compare	
Divide	Divide	
Exponential	Exponential	
Fixed-to-float	Fixed_to_float	
Float-to-fixed	Float_to_fixed	
Float-to-float	Float_to_float	
Fused Multiply-Add	FMA	
Logarithm	Logarithm	
Multiply	Multiply	
Reciprocal	Reciprocal	
Reciprocal Square Root	Rec_Square_Root	
Square-root	Square_root	
Unfused Multiply-Add	Unfused_Multiply_Add	

Table 4-4: **Vivado IDE Parameter to User Parameter Relationship *(Cont'd)***

Vivado IDE Parameter/Value[1]	User Parameter/Value[1]	Default Value
Unfused Multiply Accumulator	Unfused_Multiply_Accumulator	
Accumulator Primitive	Accumulator_Primitive	
Add/Subtract and FMA Operator options	add_sub_value	Both
Compare Operator Options	c_compare_operation	Programmable
Precision of Inputs: A Precision Type	a_precision_type	Single
Precision of Inputs: Exponent Width	c_a_exponent_width	8
Precision of Inputs: Fraction Width	c_a_fraction_width	24
Result Precision Type	result_precision_type	Single
Precision of Result: Exponent Width	c_result_exponent_width	8
Precision of Result: Fraction Width	c_result_fraction_width	24
Accumulator MSB	c_accum_msb	32
Accumulator LSB	c_accum_lsb	-31
Input MSB	c_accum_input_msb	32
Architecture Optimizations	c_optimization	Speed_optimized
High_Speed	Speed_Optimised	
Low_Latency	Low_Latency	
DSP Slice Usage	c_mult_usage	No_Usage
Block Memory Usage	c_bram_usage	No_Usage
Flow Control	flow_control	Blocking
Optimize Goal	axi_optimize_goal	Resources
RESULT channel has TREADY	has_result_tready	True
Use Maximum Latency	maximum_latency	True
Latency	c_latency	12
Cycles/Operation	c_rate	1
ACLKEN	has_aclken	False
ARESETn	has_aresetn	False
UNDERFLOW	c_has_underflow	False
OVERFLOW	c_has_overflow	False
INVALID OP	c_has_invalid_op	False
DIVIDE BY ZERO	c_has_divide_by_zero	False
ACCUM OVERFLOW	c_has_accum_overflow	false
ACCUM INPUT OVERFLOW	c_has_accum_input_overflow	False
Channel A: Has TLAST	has_a_tlast	False
Channel A: Has TUSER	has_a_tuser	False
Channel A: TUSER Width	a_tuser_width	1

Table 4-4: **Vivado IDE Parameter to User Parameter Relationship** ***(Cont'd)***

Vivado IDE Parameter/Value[1]	User Parameter/Value[1]	Default Value
Channel B: Has TLAST	has_b_tlast	False
Channel B: Has TUSER	has_b_tuser	False
Channel B: TUSER Width	b_tuser_width	1
Channel C: Has TLAST	has_c_tlast	False
Channel C: Has TUSER	has_c_tuser	False
Channel C: TUSER Width	c_tuser_width	1
Channel OPERATION: Has TLAST	has_operation_tlast	False
Channel OPERATION: Has TUSER	has_operation_tuser	False
Channel OPERATION: TUSER Width	operation_tuser_width	1
TLAST Behavior	result_tlast_behv	Null

Notes:

1. Parameter values are listed in the table where the Vivado IDE parameter value differs from the user parameter value. Such values are shown in this table as indented below the associated parameter.

Using the Floating-Point Operator IP Core

Core Use through Vivado Design Suite

The Vivado Design Suite performs error-checking on all input parameters. Resource estimation and latency information are also available. Several files are produced when a core is generated, and customized instantiation templates for Verilog and VHDL design flows are provided in the .veo and .vho files, respectively. For detailed instructions, see the *Vivado Design Suite User Guide: Designing with IP* (UG896) [Ref 6].

Core Use through System Generator for DSP

The Floating-Point Operator core is available through Xilinx System Generator, a DSP design tool that enables the use of The Mathworks model-based design environment Simulink® for FPGA design. The Floating-Point Operator is used within DSP math building blocks provided in the Xilinx blockset for Simulink. The blocks that provide floating-point operations using the Floating-Point Operator core are:

- AddSub
- Accumulator
- Mult
- Fused Multiply-Add
- CMult (Constant Multiplier)
- Divide

- Reciprocal
- SquareRoot
- Reciprocal SquareRoot
- Absolute
- Logarithm
- Exponential
- Relational (provides compare operations)
- Convert (provides fixed to float, float to fixed, float to float)
- Accumulator Primitive
- Unfused Multiply-Add
- Unfused Multiply-Accumulate

See the *System Generator for DSP User Guide* (UG640) [Ref 9] for more information.

Output Generation

For details, see the *Vivado Design Suite User Guide: Designing with IP* (UG896) [Ref 6].

Constraining the Core

This section contains information about constraining the core in the Vivado Design Suite.

Required Constraints

This section is not applicable for this IP core.

Device, Package, and Speed Grade Selections

This section is not applicable for this IP core.

Clock Frequencies

This section is not applicable for this IP core.

Clock Management

This section is not applicable for this IP core.

Clock Placement

This section is not applicable for this IP core.

Banking

This section is not applicable for this IP core.

Transceiver Placement

This section is not applicable for this IP core.

I/O Standard and Placement

This section is not applicable for this IP core.

Simulation

For comprehensive information about Vivado simulation components, as well as information about using supported third party tools, see the *Vivado Design Suite User Guide: Logic Simulation* (UG900) [Ref 10].

Synthesis and Implementation

For details about synthesis and implementation, see the *Vivado Design Suite User Guide: Designing with IP* (UG896) [Ref 6].

Chapter 5

C Model

The Xilinx® LogiCORE™ IP Floating-Point Operator core bit accurate C model is a self-contained, linkable, shared library that models the functionality of this core with finite precision arithmetic. This model provides a bit accurate representation of the various modes of the Floating-Point Operator v7.1 core, and it is suitable for inclusion in a larger framework for system-level simulation or core-specific verification.

The C model is an optional output of the Vivado® Design Suite. For information about generating IP source outputs, see *Vivado Design Suite User Guide: Using the Vivado IDE* (UG893) [Ref 12].

Features

- Bit accurate with Floating-Point Operator core
- Available for 64-bit Linux and Windows platforms
- Supports all features of the Floating-Point Operator core
- Designed for integration into a larger system model
- Example C code showing how to use the C model functions

Overview

This product guide provides information about the Xilinx LogiCORE IP Floating-Point Operator v7.1 bit accurate C model for 64-bit Linux and Windows platforms.

The model consists of a set of C functions that reside in a shared library. Example C code is provided to demonstrate how these functions form the interface to the C model. Full details of this interface are given in C Model Interface.

The model is bit accurate but not cycle-accurate; it performs exactly the same operations as the core. However, it does not model the core latency or its interface signals.

Unpacking and Model Contents

There are separate ZIP files containing all the files necessary for use with a specific computing platform. Each ZIP file contains:

- The C model shared library
- Multiple Precision Integers and Rationals (MPIR) [Ref 3] and Multiple Precision Floating-point Reliable (MPFR) [Ref 4] shared libraries
- The C model header file
- The example code showing customers how to call the C model

Note: The C model uses MPIR and MPFR libraries. MPIR is an interface-compatible version of the GNU Multiple Precision (GMP) [Ref 2] library, with greater support for Windows platforms. MPIR has been compiled using its GMP compatibility option, so the MPIR library and header file use GMP file names. MPFR uses GMP, but here has been configured to use MPIR instead. Source code for the MPIR library, the MPFR library, and the Visual Studio project files for MPFR can be obtained from https://www.xilinx.com/products/design-tools/guest-resources.html.

Table 5-1: **Example C Model ZIP File Contents – Linux**

File	Description
floating_point_v7_1_bitacc_cmodel.h	Header file which defines the C model API
libIp_floating_point_v7_1_bitacc_cmodel.so	Model shared object library
libgmp.so.11	MPIR library, used by the C model
libmpfr.so.4	MPFR library, used by the C model
gmp.h	MPIR header file, used by the C model
mpfr.h	MPFR header file, used by the C model
run_bitacc_cmodel.c	Example program for calling the C model
allfns.c	Detailed example C code showing how to call every C model function

Table 5-2: **Example C Model ZIP File Contents – Windows**

File	Description
floating_point_v7_1_bitacc_cmodel.h	Header file which defines the C model API
libIp_floating_point_v7_1_bitacc_cmodel.dll	Model dynamically linked library
libIp_floating_point_v7_1_bitacc_cmodel.lib	Model .lib file for compiling
libgmp.dll	MPIR library, used by the C model
libgmp.lib	MPIR .lib file for compiling
libmpfr.dll	MPFR library, used by the C model
libmpfr.lib	MPFR .lib file for compiling
gmp.h	MPIR header file, used by the C model

Table 5-2: **Example C Model ZIP File Contents – Windows *(Cont'd)***

File	Description
mpfr.h	MPFR header file, used by the C model
run_bitacc_cmodel.c	Example program for calling the C model
allfns.c	Detailed example C code showing how to call every C model function

Installation

Linux

- Unpack the contents of the ZIP file.
- Ensure that the directory where the `libIp_floating_point_v7_1_bitacc_cmodel.so`, `libgmp.so.11` and `libmpfr.so.4` files reside is included in the path of the environment variable LD_LIBRARY_PATH.

Windows

- Unpack the contents of the ZIP file.
- Ensure that the directory where the `libIp_floating_point_v7_1_bitacc_cmodel.dll`, `libgmp.dll` and `libmpfr.dll` files reside is

  a. included in the path of the environment variable PATH or

  b. the directory in which the executable that calls the C model is run.

C Model Interface

The Floating-Point Operator C model has a C function based Application Programming Interface (API), which is very similar to the APIs of other floating-point arithmetic libraries MPIR (Multiple Precision Integers and Rationals) and MPFR (GNU Multiple Precision Floating-point Reliable library). The C model uses these libraries internally and provides functions to convert between their data types.

Note: MPIR [Ref 3] and MPFR [Ref 4] are free, open source software libraries, distributed under the GNU Lesser General Public License. A compiled version of each library is provided with the C model. MPIR is a compatible alternative to GMP (GNU Multiple Precision Arithmetic) [Ref 2] that provides greater support for Windows platforms. MPIR and GMP can be used interchangeably.

Two example C files, `run_bitacc_cmodel.c` and `allfns.c`, are included, that demonstrate how to call the C model. See these files for examples of using the interface described in the following sections.

The Application Programming Interface (API) of the C model is defined in the header file `floating_point_v7_1_bitacc_cmodel.h`. The interface consists of data structures and functions as described in the following sections.

Data Types

The C types defined for the Floating-Point Operator C model are shown in Table 5-3.

Table 5-3: **Floating-Point Operator C Model Data Types**

Name	Type	Description
xip_fpo_prec_t	long	Precision of mantissa or exponent (bits)
xip_fpo_sign_t	int	Sign bit of a floating-point number
xip_fpo_exp_t	long	Exponent of a floating-point number
xip_fpo_t	struct[1]	Custom precision floating-point number (internally defined as a one-element array of a structure)
xip_fpo_fix_t	struct[1]	Custom precision fixed-point number (internally defined as a one-element array of a structure)
xip_fpo_ptr	struct *	Pointer to underlying custom precision floating-point struct. Equivalent to `xip_fpo_t` but easier to use in certain situations (for example, terminator in `xip_fpo_inits2` function).
xip_fpo_fix_ptr	struct *	Pointer to underlying custom precision fixed-point struct. Equivalent to `xip_fpo_fix_t` but easier to use in certain situations (for example, terminator in `xip_fpo_fix_inits2` function).
xip_fpo_exc_t	int	Bitwise flags which when set indicate exceptions that occurred during an operation: bit 0: underflow bit 1: overflow bit 2: invalid operation bit 3: divide by zero bit 4: operation not supported by Floating-Point Operator v7.1 core (for example, add with different precision operands) bit 5: accumulator input overflow bit 6: accumulator overflow

`xip_fpo_prec_t` is used for initializing variables of type `xip_fpo_t` and `xip_fpo_fix_t`.

`xip_fpo_prec_t` and `xip_fpo_exp_t` are of type `long` for compatibility with MPFR, not because they need a greater numerical range than provided by `int`.

The Floating-Point Operator C model functions use `xip_fpo_t` and `xip_fpo_fix_t` for input and output variables. Users should use these types for all custom precision floating-point and fixed-point variables. Defining this type as a one-element array of the underlying struct means that when a user declares a variable of this type, the memory for the struct members is automatically allocated, and you can pass the variable as-is to functions with no need to add a * to pass a pointer, and it is automatically passed by reference. This is the same method as used by MPIR [Ref 3] and MPFR [Ref 4].

`xip_fpo_t` is an IEEE-754 compatible floating-point type, except that signaling NaNs and denormalized numbers are not supported. If a signaling NaN is stored in an `xip_fpo_t` variable, the value becomes a quiet NaN. Similarly, denormalized numbers are converted to zero (with an underflow exception, if appropriate).

`xip_fpo_exc_t` is the return value type of most functions.

The C model API also provides versions of its operation functions for single and double precision, using standard C data types `float` and `double` respectively. This provides an easy use model for applications that do not require custom precision.

Because there is no standard C data type to model half precision (binary16) data, the custom precision variant of each C model function should be used with appropriately defined `xip_fpo_t` variables to describe a half precision datapath.

Functions

There are several C model functions accessible to you.

Information Functions

The Floating-Point Operator C model information functions are shown in Table 5-4.

Table 5-4: **Floating-Point Operator C Model Information Functions**

Name	Return	Arguments	Description
xip_fpo_get_version	const char *	void	Return the Floating-Point Operator C model version, as a null-terminated string. For v7.1 this is "7.1".

Initialization Functions

The Floating-Point Operator C model initialization functions are shown in Table 5-5. Most functions have variants to handle floating-point and fixed-point variables.

Table 5-5: **Floating-Point Operator C Model Initialization Functions**

Name	Return	Arguments	Description
xip_fpo_init2	void	xip_fpo_t x, xip_fpo_prec_t exp, xip_fpo_prec_t mant	Initialize floating-point variable *x*, set its exponent precision to *exp*, its mantissa precision to *mant*, and its value to NaN.
xip_fpo_fix_init2	void	xip_fpo_fix_t x, xip_fpo_prec_t i, xip_fpo_prec_t frac	Initialize fixed-point variable *x*, set its integer precision to *i*, its fraction precision to *frac*, and its value to zero.
xip_fpo_inits2	void	xip_fpo_prec_t exp, xip_fpo_prec_t mant, xip_fpo_t x, ...	Initialize all `xip_fpo_t` variables pointed to by the argument list, set their exponent precision to *exp*, their mantissa precision to *mant*, and their value to NaN. The last item in the list must be a null pointer of type `xip_fpo_t` (or equivalently `xip_fpo_ptr`).
xip_fpo_fix_inits2	void	xip_fpo_prec_t i, xip_fpo_prec_t frac, xip_fpo_fix_t x, ...	Initialize all `xip_fpo_fix_t` variables pointed to by the argument list, set their integer precision to *i*, their fraction precision to *frac*, and their value to zero. The last item in the list must be a null pointer of type `xip_fpo_fix_t` (or equivalently `xip_fpo_fix_ptr`).
xip_fpo_clear	void	xip_fpo_t x	Free the memory used by *x*.
xip_fpo_fix_clear	void	xip_fpo_fix_t x	Free the memory used by *x*.
xip_fpo_clears	void	xip_fpo_t x,...	Free the memory used by all `xip_fpo_t` variables pointed to by the argument list. The last item in the list must be a null pointer of type `xip_fpo_t` (or equivalently `xip_fpo_ptr`).
xip_fpo_fix_clears	void	xip_fpo_fix_t x,...	Free the memory used by all `xip_fpo_fix_t` variables pointed to by the argument list. The last item in the list must be a null pointer of type `xip_fpo_fix_t` (or equivalently `xip_fpo_fix_ptr`).
xip_fpo_set_prec	void	xip_fpo_t x, xip_fpo_prec_t exp, xip_fpo_prec_t mant	Reset *x* to an exponent precision of *exp*, a mantissa precision of *mant*, and set its value to NaN. The previous value of *x* is lost.
xip_fpo_fix_set_prec	void	xip_fpo_fix_t x, xip_fpo_prec_t i, xip_fpo_prec_t frac	Reset *x* to an integer precision of *i*, a fraction precision of *frac*, and set its value to zero. The previous value of *x* is lost.

Table 5-5: **Floating-Point Operator C Model Initialization Functions *(Cont'd)***

Name	Return	Arguments	Description
xip_fpo_get_prec_mant	xip_fpo_prec_t	xip_fpo_t x	Return the mantissa precision (in bits) of *x*.
xip_fpo_get_prec_exp	xip_fpo_prec_t	xip_fpo_t x	Return the exponent precision (in bits) of *x*.
xip_fpo_fix_get_prec_frac	xip_fpo_prec_t	xip_fpo_fix_t x	Return the fraction precision (in bits) of *x*.
xip_fpo_fix_get_prec_int	xip_fpo_prec_t	xip_fpo_fix_t x	Return the integer precision (in bits) of *x*.

A floating-point number has a minimum exponent required to support normalization:

$$\text{minimum exponent width} = \mathit{ceil}(\log_2(\text{fraction width} + 3)) + 1$$

If the exponent width specified for `xip_fpo_init2` or `xip_fpo_set_prec` for initializing or resetting a floating-point variable is too small, it is internally increased to the minimum permitted width.

A variable should be initialized only once, or be cleared using `xip_fpo_clear` between initializations. To change the precision of a variable that has already been initialized, use `xip_fpo_set_prec`.

An example of initializing and clearing floating-point variables is shown:

```
xip_fpo_t x, y, z;
xip_fpo_init2 (x, 11, 53);     // double precision
xip_fpo_inits2 (7, 17, y, z, (xip_fpo_ptr) 0);  // custom precision
// perform operations
xip_fpo_set_prec (8, 24, y);   // change to single precision
// more operations
xip_fpo_clears (x, y, z, (xip_fpo_ptr) 0);
```

Assignment Functions

The Floating-Point Operator C model assignment functions are shown in Table 5-6. Most functions have variants to handle both floating-point and fixed-point variables. Functions are provided for assigning Floating-Point Operator C model variables from MPIR and MPFR variables for ease of use alongside these existing libraries.

Table 5-6: **Floating-Point Operator C Model Assignment Functions**

Name	Return	Arguments	Description
xip_fpo_set	xip_fpo_exc_t	xip_fpo_t rop, xip_fpo_t op	Set the value of *rop* to *op*.[1]
xip_fpo_fix_set	xip_fpo_exc_t	xip_fpo_fix_t rop, xip_fpo_fix_t op	
xip_fpo_set_ui	xip_fpo_exc_t	xip_fpo_t rop, unsigned long op	
xip_fpo_fix_set_ui	xip_fpo_exc_t	xip_fpo_fix_t rop, unsigned long op	
xip_fpo_set_si	xip_fpo_exc_t	xip_fpo_t rop, signed long op	
xip_fpo_fix_set_si	xip_fpo_exc_t	xip_fpo_fix_t rop, signed long op	
xip_fpo_set_uj	xip_fpo_exc_t	xip_fpo_t rop, uintmax_t op	
xip_fpo_fix_set_uj	xip_fpo_exc_t	xip_fpo_fix_t rop, uintmax_t op	
xip_fpo_set_sj	xip_fpo_exc_t	xip_fpo_t rop, intmax_t op	
xip_fpo_fix_set_sj	xip_fpo_exc_t	xip_fpo_fix_t rop, intmax_t op	
xip_fpo_set_flt	xip_fpo_exc_t	xip_fpo_t rop, float op	
xip_fpo_fix_set_flt	xip_fpo_exc_t	xip_fpo_fix_t rop, float op	
xip_fpo_set_d	xip_fpo_exc_t	xip_fpo_t rop, double op	
xip_fpo_fix_set_d	xip_fpo_exc_t	xip_fpo_fix_t rop, double op	
xip_fpo_set_z	xip_fpo_exc_t	xip_fpo_t rop, mpz_t op	Set the value of *rop* to the value of GMP/MPIR integer *op*.[1]
xip_fpo_fix_set_z	xip_fpo_exc_t	xip_fpo_fix_t rop, mpz_t op	
xip_fpo_set_q	xip_fpo_exc_t	xip_fpo_t rop, mpq_t op	Set the value of *rop* to the value of GMP/MPIR rational number *op*.[1]
xip_fpo_fix_set_q	xip_fpo_exc_t	xip_fpo_fix_t rop, mpq_t op	
xip_fpo_set_f	xip_fpo_exc_t	xip_fpo_t rop, mpf_t op	Set the value of *rop* to the value of GMP/MPIR floating-point number *op*.[1]
xip_fpo_fix_set_f	xip_fpo_exc_t	xip_fpo_fix_t rop, mpf_t op	

Table 5-6: **Floating-Point Operator C Model Assignment Functions *(Cont'd)***

Name	Return	Arguments	Description
xip_fpo_set_fr	xip_fpo_exc_t	xip_fpo_t rop, mpfr_t op	Set the value of *rop* to the value of MPFR floating-point number *op.*[1]
xip_fpo_fix_set_fr	xip_fpo_exc_t	xip_fpo_fix_t rop, mpfr_t op	
xip_fpo_set_ui_2exp	xip_fpo_exc_t	xip_fpo_t rop, unsigned long op, xip_fpo_exp_t e	Set the value of *rop* to *op* multiplied by two to the power of *e.*[1]
xip_fpo_set_si_2exp	xip_fpo_exc_t	xip_fpo_t rop, signed long op, xip_fpo_exp_t e	
xip_fpo_set_uj_2exp	xip_fpo_exc_t	xip_fpo_t rop, uintmax_t op, intmax_t e	
xip_fpo_set_sj_2exp	xip_fpo_exc_t	xip_fpo_t rop, intmax_t op, intmax_te	
xip_fpo_set_str	xip_fpo_exc_t	xip_fpo_t rop, const char *s, int base	Set the value of *rop* to the string in *s* which is in the base *base*. See xip_fpo_set_str and xip_fpo_fix_set_str for details.[1]
xip_fpo_fix_set_str	xip_fpo_exc_t	xip_fpo_fix_t rop, const char *s, int base	
xip_fpo_set_nan	void	xip_fpo_t x	Set the value of *x* to NaN.
xip_fpo_set_inf	void	xip_fpo_t x, int sign	Set the value of *x* to plus infinity if *sign* is non-negative, minus infinity otherwise.
xip_fpo_set_zero	void	xip_fpo_t x, int sign	Set the value of *x* to plus zero if *sign* is non-negative, minus zero otherwise.

Notes:

1. Any exceptions that occur are signaled in the return value.
   When assigning to a fixed-point variable, if overflow occurs, the result is saturated and the return value is the largest representable fixed-point number of the correct sign. Converting a NaN returns the most negative representable fixed-point number and the invalid operation exception is signaled in the return value.

xip_fpo_set_str and xip_fpo_fix_set_str

The functions `xip_fpo_set_str` and `xip_fpo_fix_set_str` take a string argument (actually const char *) and an integer base. They have the same usage as the MPFR function `mpfr_set_str`.

The base is a value between 2 and 62 or zero. The string is a representation of numeric data to be read and stored in the floating-point variable. The whole string must represent a valid floating-point number.

The form of numeric data is a non-empty sequence of significand digits with an optional decimal point, and an optional exponent consisting of an exponent prefix followed by an optional sign and a non-empty sequence of decimal digits. A significand digit is either a decimal digit or a Latin letter (62 possible characters), with A = 10, B = 11, ..., Z = 35; case is ignored in bases less or equal to 36, in bases larger than 36, a = 36, b = 37, ..., z = 61. The value of a significant digit must be strictly less than the base. The decimal point can be either the one defined by the current locale or the period (the first one is accepted for consistency with the C standard and the practice, the second one is accepted to allow the programmer to provide numbers from strings in a way that does not depend on the current locale). The exponent prefix can be e or E for bases up to 10, or @ in any base; it indicates a multiplication by a power of the base. In bases 2 and 16, the exponent prefix can also be p or P, in which case the exponent, called binary exponent, indicates a multiplication by a power of 2 instead of the base (there is a difference only for base 16); in base 16 for example 1p2 represents 4 whereas 1@2 represents 256.

If the argument *base* is 0, then the base is automatically detected as follows. If the significand starts with 0b or 0B, base 2 is assumed. If the significand starts with 0x or 0X, base 16 is assumed. Otherwise base 10 is assumed.

Note: The exponent (if present) must contain at least a digit. Otherwise, the possible exponent prefix and sign are not part of the number (which ends with the significand). Similarly, if 0b, 0B, 0x or 0X is not followed by a binary/hexadecimal digit, then the subject sequence stops at the character 0, thus 0 is read.

Special data (for infinities and NaN) can be `@inf@` or `@nan@(n-char-sequence-opt)`, and if base <= 16, it can also be `infinity`, `inf`, `nan` or `nan(n-char-sequence-opt)`, all case insensitive. A `n-char-sequence-opt` is a possibly empty string containing only digits, Latin letters and the underscore (0, 1, 2, ..., 9, a, b, ..., z, A, B, ..., Z, _).

Note: There is an optional sign for all data, even NaN. For example, `-@nAn@(This_Is_Not_17)` is a valid representation for NaN in base 17.

If the whole string cannot be parsed into a floating-point or fixed-point number, then an invalid operation exception is signaled. In this case, *rop* might have changed. Overflow or underflow can occur if the string is parsed to a floating-point or fixed-point number that is too large or too small to represent in the floating-point or fixed-point precision of the variable.

Conversion Functions

The Floating-Point Operator C model conversion functions are shown in Table 5-7. Most functions have variants to handle both floating-point and fixed-point variables.

Functions that convert to a standard C data type return the converted result as that data type. Any exceptions that occur are ignored. Functions that convert to GMP or MPFR data types place the result in the first argument and return exception flags, as with most Floating-Point Operator C model functions.

Table 5-7: **Floating-Point Operator C Model Conversion Functions**

Name	Return	Arguments	Description
xip_fpo_get_ui	unsigned long	xip_fpo_t op	Convert *op* to an unsigned long int after rounding.
xip_fpo_fix_get_ui	unsigned long	xip_fpo_fix_t op	
xip_fpo_get_si	signed long	xip_fpo_t op	Convert *op* to a signed long int after rounding.
xip_fpo_fix_get_si	signed long	xip_fpo_fix_t op	
xip_fpo_get_uj	uintmax_t	xip_fpo_t op	Convert *op* to an unsigned maximum size integer after rounding.
xip_fpo_fix_get_uj	uintmax_t	xip_fpo_fix_t op	
xip_fpo_get_sj	intmax_t	xip_fpo_t op	Convert *op* to a signed maximum size integer after rounding.
xip_fpo_fix_get_sj	intmax_t	xip_fpo_fix_t op	
xip_fpo_get_flt	float	xip_fpo_t op	Convert *op* to a float.
xip_fpo_fix_get_flt	float	xip_fpo_fix_t op	
xip_fpo_get_d	double	xip_fpo_t op	Convert *op* to a double.
xip_fpo_fix_get_d	double	xip_fpo_fix_t op	
xip_fpo_get_d_2exp	double	long *exp, xip_fpo_t op	Convert the mantissa of *op* to a double such that 0.5<=abs(mantissa)<1, and set the value pointed to by *exp* to the exponent of *op*. If *op* is zero, zero is returned and *exp* is zero. If *op* is NaN or infinity, NaN or infinity respectively is returned and *exp* is undefined.
xip_fpo_get_z	xip_fpo_exc_t	mpz_t rop, xip_fpo_t op	Convert *op* to a GMP/MPIR integer after rounding and store in *rop*. If *op* is NaN or infinity, *rop* is set to 0 and an invalid operation exception is returned.
xip_fpo_fix_get_z	xip_fpo_exc_t	mpz_t rop, xip_fpo_fix_t op	
xip_fpo_get_f	xip_fpo_exc_t	mpf_t rop, xip_fpo_t op	Convert *op* to a GMP/MPIR floating-point number and store it in *rop*. If *op* is NaN or infinity, *rop* is set to 0 and an invalid operation exception is returned.
xip_fpo_fix_get_f	xip_fpo_exc_t	mpf_t rop, xip_fpo_fix_t op	
xip_fpo_get_fr	xip_fpo_exc_t	mpfr_t rop, xip_fpo_t op	Convert *op* to an MPFR floating-point number and store it in *rop*.
xip_fpo_fix_get_fr	xip_fpo_exc_t	mpfr_t rop, xip_fpo_fix_t op	
xip_fpo_get_str	char *	char * str, xip_fpo_exp_t * exp, int base, int n_digits, xip_fpo_t op	Convert *op* to a string of digits in base *base*, returning the exponent separately in the variable pointed to by *exp*. See xip_fpo_get_str for details.
xip_fpo_fix_get_str	char *	char * str, int base, xip_fpo_fix_t op	Convert *op* to a string of digits in base *base*. See for details.

Table 5-7: **Floating-Point Operator C Model Conversion Functions *(Cont'd)***

Name	Return	Arguments	Description
xip_fpo_free_str	void	char * str	Free a string allocated by `xip_fpo_get_str` or `xip_fpo_fix_get_str`.
xip_fpo_fix_free_str	void	char * str	A synonym for `xip_fpo_free_str`.
xip_fpo_sizeinbase	int	xip_fpo_t op, int base	Return the size of *op* measured in number of digits in the given *base*. *base* can vary from 2 to 62. The sign of *op* is ignored. Returns –1 if an error occurs. Use to determine the space required when converting *op* to a string using `xip_fpo_get_str` or `xip_fpo_fix_get_str`.
xip_fpo_fix_sizeinbase	int	xip_fpo_fix_t op, int base	

xip_fpo_get_str

The function `xip_fpo_get_str` has the same usage as the MPFR function `mpfr_get_str`. *n_digits* is either zero or the number of significant digits output in the string; in the latter case, *n_digits* must be greater or equal to 2. The base can vary from 2 to 62. If the input number is an ordinary number, the exponent is written through the pointer *exp* (for input 0, the exponent is set to 0).

The generated string is in the base specified by *base*. Each string character is either a decimal digit or a Latin letter (62 possible characters). For *base* in the range 2 to 36, decimal digits and lowercase letters are used, with a = 10, b = 11, ... z = 35. For *base* in the range 37 to 62, digits, uppercase, and lowercase letters are used, with A = 10, B = 11, ..., Z = 35, a = 36, b = 37, ..., z = 61.

The generated string is a fraction, with an implicit radix point immediately to the left of the first digit. For example, the number –3.1416 would be returned as "–31416" in the string and 1 written at *exp*. The value is rounded to provide *n_digits* of output, using round to nearest even: if *op* is exactly in the middle of two consecutive possible outputs, the one with an even significand is chosen, where both significands are considered with the exponent of *op*. For an odd base, this might not correspond to an even last digit: for example with 2 digits in base 7, (14) and a half is rounded to (15) which is 12 in decimal, (16) and a half is rounded to (20) which is 14 in decimal, and (26) and a half is rounded to (26) which is 20 in decimal.

If *n_digits* is zero, the number of digits of the significand is chosen large enough so that re-reading the printed value with the same precision recovers the original value of *op*. More precisely, in most cases, the chosen precision of *str* is the minimal precision m depending only on p = PREC*(op)* and b that satisfies the above property, that is,
m = 1 + ceil(p × log(2)/log(b)), with p replaced by p – 1 if b is a power of 2.

If *str* is a null pointer, space for the significand is allocated using the GMP/MPIR current allocation function which is `malloc()` by default, and a pointer to the string is returned. To free the memory used by the returned string, you must use `xip_fpo_free_str`.

If *str* is not a null pointer, it should point to a block of storage large enough for the significand, that is, at least max(*n_digits* + 2, 7) if *n_digits* > 0, or `xip_fpo_sizeinbase`(*op*, *base*) + 2 otherwise. The extra two bytes are for a possible minus sign, and for the terminating null character, and the value 7 accounts for `-@Inf@` plus the terminating null character.

A pointer to the string is returned, unless there is an error, in which case a null pointer is returned.

xip_fpo_fix_get_str

The function `xip_fpo_fix_get_str` has the same usage as the GMP/MPIR function `mpz_get_str`. The base can vary from 2 to 62.

The generated string is in the base specified by *base*. Each string character is either a decimal digit or a Latin letter (62 possible characters). For base in the range 2 to 36, decimal digits and lowercase letters are used, with a = 10, b = 11, ... z = 35. For base in the range 37 to 62, digits, uppercase, and lowercase letters are used, with A = 10, B = 11, ..., Z = 35, a = 36, b = 37, ..., z = 61.

The generated string is either an integer value with no radix point, or a fraction with an explicit radix point. All significant digits are returned, but no leading or trailing zeros are returned. No rounding is carried out.

If *str* is a null pointer, space for the significand is allocated using the current allocation function, and a pointer to the string is returned. To free the memory used by the returned string, you must use `xip_fpo_fix_free_str`.

If *str* is not a null pointer, it should point to a block of storage large enough for the result, that being `xip_fpo_fix_sizeinbase`(*op*, *base*) + 2. The extra two bytes are for a possible minus sign, and the terminating null character.

Operation Functions

The Floating-Point Operator C model functions that model operations of the core are shown in Table 5-8. In addition to functions using `xip_fpo_t` and `xip_fpo_fix_t` type arguments to provide custom precision, alternative versions of functions using standard C data types `float` and `double` are also provided to make it easy for customers who do not need custom precision. For fixed to float and float to fixed functions, `float` and `double` to and from `int` are provided. For float to float functions, all combinations of `float` and `double` are provided: where these data types are the same, the function provides a means to condition numbers (convert signaling NaNs to quiet NaNs, convert denormalized numbers to zero).

Table 5-8: **Floating-Point Operator C Model Operation Functions**

Name	Return	Arguments	Description
xip_fpo_add	xip_fpo_exc_t	xip_fpo_t rop, xip_fpo_t op1, xip_fpo_t op2	Set *rop* = *op1* + *op2*. *rop*, *op1* and *op2* must have identical precisions, otherwise an operation not supported exception is returned.
xip_fpo_add_flt	xip_fpo_exc_t	float * rop, float op1, float op2	Set *rop* = *op1* + *op2*. Single precision version.
xip_fpo_add_d	xip_fpo_exc_t	double * rop, double op1, double op2	Set *rop* = *op1* + *op2*. Double precision version.
xip_fpo_sub	xip_fpo_exc_t	xip_fpo_t rop, xip_fpo_t op1, xip_fpo_t op2	Set *rop* = *op1* – *op2*. *rop*, *op1* and *op2* must have identical precisions, otherwise an operation not supported exception is returned.
xip_fpo_sub_flt	xip_fpo_exc_t	float * rop, float op1, float op2	Set *rop* = *op1* – *op2*. Single precision version.
xip_fpo_sub_d	xip_fpo_exc_t	double * rop, double op1, double op2	Set *rop* = *op1* – *op2*. Double precision version.
xip_fpo_accum_create_state	xil_fpo_accum_state *	int exponent_width, int fractional_width, int accumulator_msb, int input_msb, int accumulator_lsb	Returns a pointer to a xil_fpo_accum_state object. This is needed by all the other accumulator functions.
xip_fpo_accum_reset_state	void	xil_fpo_accum_state *state	Resets the xil_fpo_accum_state object. Call this to start a new accumulation. Has the effect of clearing the accumulator and resetting the exception flags.
xip_fpo_accum_destroy_state	void	xil_fpo_accum_state *state	Destroys the xil_fpo_accum_state object and frees any memory allocated. The xil_fpo_accum_state object pointer can no longer be used.

Table 5-8: **Floating-Point Operator C Model Operation Functions *(Cont'd)***

Name	Return	Arguments	Description
xip_fpo_accum_sample	xip_fpo_exc_t	xip_fpo_t rop, xip_fpo_t op, bool subtract, xil_fpo_accum_state *state	Add/Subtract *op* to/from the accumulator and return the result. Adds when *subtract* = *FALSE*; Subtracts when *subtract* = *TRUE*. *rop* and *op* must have identical precisions, otherwise an operation not supported exception is returned.
xip_fpo_accum_sample_flt	xip_fpo_exc_t	float * rop, float op, bool subtract, xil_fpo_accum_state *state	Add/Subtract *op* to/from the accumulator and return the result. Single precision version. Adds when *subtract* = *FALSE*; Subtracts when *subtract* = *TRUE*.
xip_fpo_accum_sample_d	xip_fpo_exc_t	double * rop, double op, bool subtract, xil_fpo_accum_state *state	Add/Subtract op to/from the accumulator and return the result. Double precision version. Adds when *subtract* = *FALSE*; Subtracts when *subtract* = *TRUE*.
xip_fpo_mul	xip_fpo_exc_t	xip_fpo_t rop, xip_fpo_t op1, xip_fpo_t op2	Set *rop* = *op1* × *op2*. *rop*, *op1* and *op2* must have identical precisions, otherwise an operation not supported exception is returned.
xip_fpo_mul_flt	xip_fpo_exc_t	float * rop, float op1, float op2	Set *rop* = *op1* × *op2*. Single precision version.
xip_fpo_mul_d	xip_fpo_exc_t	double * rop, double op1, double op2	Set *rop* = *op1* × *op2*. Double precision version.
xip_fpo_fma	xip_fpo_exc_t	xip_fpo_t rop, xip_fpo_t op1, xip_fpo_t op2, xip_fpo_t op3	Set *rop* = *(op1 x op2)* + *op3*. *rop*, *op1*, *op2*, and *op3* must have identical precisions, otherwise an operation not supported exception is returned.
xip_fpo_fma_flt	xip_fpo_exc_t	float * rop, float op1, float op2, float op3	Set *rop* = *(op1 x op2)* + *op3*. Single precision version.

Table 5-8: **Floating-Point Operator C Model Operation Functions *(Cont'd)***

Name	Return	Arguments	Description
xip_fpo_fma_d	xip_fpo_exc_t	double * rop, double op1, double op2, double op3	Set *rop = (op1 x op2) + op3*. Double precision version.
xip_fpo_fms	xip_fpo_exc_t	xip_fpo_t rop, xip_fpo_t op1, xip_fpo_t op2, xip_fpo_t op3	Set *rop = (op1 x op2) – op3*. *rop*, *op1*, *op2*, and *op3* must have identical precisions, otherwise an operation not supported exception is returned.
xip_fpo_fms_flt	xip_fpo_exc_t	float * rop, float op1, float op2, float op3	Set *rop = (op1 x op2) – op3*. Single precision version.
xip_fpo_fms_d	xip_fpo_exc_t	double * rop, double op1, double op2, double op3	Set *rop = (op1 x op2) – op3*. Double precision version.
xip_fpo_div	xip_fpo_exc_t	xip_fpo_t rop, xip_fpo_t op1, xip_fpo_t op2	Set *rop = op1/op2*. *rop*, *op1* and *op2* must have identical precisions, otherwise an operation not supported exception is returned.
xip_fpo_div_flt	xip_fpo_exc_t	float * rop, float op1, float op2	Set *rop = op1 / op2*. Single precision version.
xip_fpo_div_d	xip_fpo_exc_t	double * rop, double op1, double op2	Set *rop = op1/op2*. Double precision version.
xip_fpo_rec[1]	xip_fpo_exc_t	xip_fpo_t rop, xip_fpo_t op	Set *rop* = 1/*op*. *rop* and *op* must have identical precisions, otherwise an operation not supported exception is returned.
xip_fpo_rec_flt	xip_fpo_exc_t	float * rop, float op	Set *rop* = 1/*op*. Single precision version.
xip_fpo_rec_d	xip_fpo_exc_t	double * rop, double op	Set *rop* = 1/*op*. Double precision version.
xip_fpo_sqrt	xip_fpo_exc_t	xip_fpo_t rop, xip_fpo_t op	Set *rop* = square root of *op*. *rop* and *op* must have identical precisions, otherwise an operation not supported exception is returned.
xip_fpo_sqrt_flt	xip_fpo_exc_t	float * rop, float op	Set *rop* = square root of *op*. Single precision version.

Table 5-8: **Floating-Point Operator C Model Operation Functions *(Cont'd)***

Name	Return	Arguments	Description
xip_fpo_sqrt_d	xip_fpo_exc_t	double * rop, double op	Set *rop* = square root of *op*. Double precision version.
xip_fpo_recsqrt[1]	xip_fpo_exc_t	xip_fpo_t rop, xip_fpo_t op	Set *rop* = 1/(square root of *op*). *rop* and *op* must have identical precisions, otherwise an operation not supported exception is returned.
xip_fpo_recsqrt_flt	xip_fpo_exc_t	float * rop, float op	Set *rop* = 1/(square root of *op*). Single precision version.
xip_fpo_recsqrt_d	xip_fpo_exc_t	double * rop, double op	Set *rop* = 1/(square root of *op*). Double precision version.
xip_fpo_abs	xip_fpo_exc_t	xip_fpo_t rop, xip_fpo_t op	Set *rop* = |*op*|. *rop* and *op* must have identical precisions, otherwise an operation not supported exception is returned.
xip_fpo_abs_flt	xip_fpo_exc_t	float * rop, float op	Set *rop* = |*op*|. Single precision version.
xip_fpo_abs_d	xip_fpo_exc_t	double * rop, double op	Set *rop* = |*op*|. Double precision version.
xip_fpo_log[1]	xip_fpo_exc_t	xip_fpo_t rop, xip_fpo_t op	Set *rop* = natural logarithm of *op*. *rop* and *op* must have identical precisions, otherwise an operation not supported exception is returned.
xip_fpo_log_flt	xip_fpo_exc_t	float * rop, float op	Set *rop* = natural logarithm of *op*. Single precision version.
xip_fpo_log_d	xip_fpo_exc_t	double * rop, double op	Set *rop* = natural logarithm of *op*. Double precision version.
xip_fpo_exp[1]	xip_fpo_exc_t	xip_fpo_t rop, xip_fpo_t op	Set *rop* = exponential of *op*. *rop* and *op* must have identical precisions, otherwise an operation not supported exception is returned.
xip_fpo_exp_flt	xip_fpo_exc_t	float * rop, float op	Set *rop* = exponential of *op*. Single precision version.
xip_fpo_exp_d	xip_fpo_exc_t	double * rop, double op	Set *rop* = exponential of *op*. Double precision version.

Table 5-8: **Floating-Point Operator C Model Operation Functions *(Cont'd)***

Name	Return	Arguments	Description
xip_fpo_exp_array[1]	int	xip_fpo_t * rop, xip_fpo_t * op, xip_fpo_exc_t *exceptions, unsigned long long num_values	Process an array of values This provides a significant performance improvement over calling xip_fpo_exp for individual values. Set *rop[i]* = exponential of *op[i]*. All *rop[]* and *op[]* entries must have identical precisions, otherwise an operation not supported exception is returned. The return value indicates the last array element processed (either successfully or with an operation not supported exception)
xip_fpo_exp_flt_array	void	float * rop, float * op, xip_fpo_exc_t exceptions*, unsigned long long num_values	Process an array of values. This provides a significant performance improvement over calling xip_fpo_exp_flt for individual values. Set *rop[i]* = exponential of *op[i]*. Single precision version.
xip_fpo_exp_d_array	void	double * rop, double * op, xip_fpo_exc_t * exceptions, unsigned long long num_values	Process an array of values. This provides a significant performance improvement over calling xip_fpo_exp_d for individual values. Set *rop[i]* = exponential of *op[i]*. Double precision version.
xip_fpo_unordered	xip_fpo_exc_t	int * res, xip_fpo_t op1, xip_fpo_t op2	Set *res* = 1 if *op1* or *op2* is a NaN, 0 otherwise. *op1* and *op2* must have identical precisions, otherwise an operation not supported exception is returned.
xip_fpo_unordered_flt	xip_fpo_exc_t	int * res, float op1, float op2	Set *res* = 1 if *op1* or *op2* is a NaN, 0 otherwise. Single precision version.
xip_fpo_unordered_d	xip_fpo_exc_t	int * res, double op1, double op2	Set *res* = 1 if *op1* or *op2* is a NaN, 0 otherwise. Double precision version.

Table 5-8: **Floating-Point Operator C Model Operation Functions *(Cont'd)***

Name	Return	Arguments	Description
xip_fpo_equal	xip_fpo_exc_t	int * res, xip_fpo_t op1, xip_fpo_t op2	Set *res* = 1 if *op1* = *op2*, 0 otherwise. *op1* and *op2* must have identical precisions, otherwise an operation not supported exception is returned.
xip_fpo_equal_flt	xip_fpo_exc_t	int * res, float op1, float op2	Set *res* = 1 if *op1* = *op2*, 0 otherwise. Single precision version.
xip_fpo_equal_d	xip_fpo_exc_t	int * res, double op1, double op2	Set *res* = 1 if *op1* = *op2*, 0 otherwise. Double precision version.
xip_fpo_less	xip_fpo_exc_t	int * res, xip_fpo_t op1, xip_fpo_t op2	Set *res* = 1 if *op1* < *op2*, 0 otherwise. *op1* and *op2* must have identical precisions, otherwise an operation not supported exception is returned.
xip_fpo_less_flt	xip_fpo_exc_t	int * res, float op1, float op2	Set *res* = 1 if *op1* < *op2*, 0 otherwise. Single precision version.
xip_fpo_less_d	xip_fpo_exc_t	int * res, double op1, double op2	Set *res* = 1 if *op1* < *op2*, 0 otherwise. Double precision version.
xip_fpo_lessequal	xip_fpo_exc_t	int * res, xip_fpo_t op1, xip_fpo_t op2	Set *res* = 1 if *op1* <= *op2*, 0 otherwise. *op1* and *op2* must have identical precisions, otherwise an operation not supported exception is returned.
xip_fpo_lessequal_flt	xip_fpo_exc_t	int * res, float op1, float op2	Set *res* = 1 if *op1* <= *op2*, 0 otherwise. Single precision version.
xip_fpo_lessequal_d	xip_fpo_exc_t	int * res, double op1, double op2	Set *res* = 1 if *op1* <= *op2*, 0 otherwise. Double precision version.
xip_fpo_greater	xip_fpo_exc_t	int * res, xip_fpo_t op1, xip_fpo_t op2	Set *res* = 1 if *op1* > *op2*, 0 otherwise. *op1* and *op2* must have identical precisions, otherwise an operation not supported exception is returned.
xip_fpo_greater_flt	xip_fpo_exc_t	int * res, float op1, float op2	Set *res* = 1 if *op1* > *op2*, 0 otherwise. Single precision version.

Table 5-8: **Floating-Point Operator C Model Operation Functions *(Cont'd)***

Name	Return	Arguments	Description
xip_fpo_greater_d	xip_fpo_exc_t	int * res, double op1, double op2	Set *res* = 1 if *op1* > *op2*, 0 otherwise. Double precision version.
xip_fpo_greaterequal	xip_fpo_exc_t	int * res, xip_fpo_t op1, xip_fpo_t op2	Set *res* = 1 if *op1* >= *op2*, 0 otherwise. *op1* and *op2* must have identical precisions, otherwise an operation not supported exception is returned.
xip_fpo_greaterequal_flt	xip_fpo_exc_t	int * res, float op1, float op2	Set *res* = 1 if *op1* >= *op2*, 0 otherwise. Single precision version.
xip_fpo_greaterequal_d	xip_fpo_exc_t	int * res, double op1, double op2	Set *res* = 1 if *op1* >= *op2*, 0 otherwise. Double precision version.
xip_fpo_notequal	xip_fpo_exc_t	int * res, xip_fpo_t op1, xip_fpo_t op2	Set *res* = 1 if *op1* <> *op2* or either *op1* or *op2* are NaN, 0 otherwise. *op1* and *op2* must have identical precisions, otherwise an operation not supported exception is returned.
xip_fpo_notequal_flt	xip_fpo_exc_t	int * res, float op1, float op2	Set *res* = 1 if *op1* <> *op2* or either *op1* or *op2* are NaN, 0 otherwise. Single precision version.
xip_fpo_notequal_d	xip_fpo_exc_t	int * res, double op1, double op2	Set *res* = 1 if *op1* <> *op2* or either *op1* or *op2* are NaN, 0 otherwise. Double precision version.
xip_fpo_condcode	xip_fpo_exc_t	int * res, xip_fpo_t op1, xip_fpo_t op2	Compare *op1* and *op2*, and set the least significant 4 bits of *res* to the resulting condition code. See Table 5-9 for the condition code encoding. *op1* and *op2* must have identical precisions, otherwise an operation not supported exception is returned.
xip_fpo_condcode_flt	xip_fpo_exc_t	int * res, float op1, float op2	Compare *op1* and *op2*, and set the least significant 4 bits of res to the resulting condition code. See Table 5-9 for the condition code encoding. Single precision version.

Table 5-8: **Floating-Point Operator C Model Operation Functions *(Cont'd)***

Name	Return	Arguments	Description
xip_fpo_condcode_d	xip_fpo_exc_t	int * res, double op1, double op2	Compare *op1* and *op2*, and set the least significant 4 bits of res to the resulting condition code. See Table 5-9 for the condition code encoding. Double precision version.
xip_fpo_flttofix	xip_fpo_exc_t	xip_fpo_fix_t rop, xip_fpo_t op	Set *rop* = *op*, rounding as required. *rop* and *op* must have compatible precisions (see xip_fpo_flttofix and xip_fpo_fixtoflt), otherwise an operation not supported exception is returned.
xip_fpo_flttofix_int_flt	xip_fpo_exc_t	int * rop, float op	Set *rop* = *op*, rounding as required. Single precision to integer version.
xip_fpo_flttofix_int_d	xip_fpo_exc_t	int * rop, double op	Set *rop* = *op*, rounding as required. Double precision to integer version.
xip_fpo_fixtoflt	xip_fpo_exc_t	xip_fpo_t rop, xip_fpo_fix_t op	Set *rop* = *op*, rounding as required. *rop* and *op* must have compatible precisions (see xip_fpo_flttofix and xip_fpo_fixtoflt), otherwise an operation not supported exception is returned.
xip_fpo_fixtoflt_flt_int	xip_fpo_exc_t	float * rop, int op	Set *rop* = *op*, rounding as required. Integer to single precision version.
xip_fpo_fixtoflt_d_int	xip_fpo_exc_t	double * rop, int op	Set *rop* = *op*, rounding as required. Integer to double precision version.
xip_fpo_fixtoflt_flt_int64	xip_fpo_exc_t	float * rop long long op	Set *rop* = *op*, rounding as required. Long long integer to single precision version.
xip_fpo_fixtoflt_d_int64	xip_fpo_exc_t	double * rop long long op	Set *rop* = *op*, rounding as required. Long long integer to double precision version.
xip_fpo_fixtoflt_flt_uint	xip_fpo_exc_t	float * rop unsigned int op	Set *rop* = *op*, rounding as required. Unsigned integer to single precision version.
xip_fpo_fixtoflt_d_uint	xip_fpo_exc_t	double * rop unsigned int op	Set *rop* = *op*, rounding as required. Unsigned integer to double precision version.

Table 5-8: **Floating-Point Operator C Model Operation Functions *(Cont'd)***

Name	Return	Arguments	Description
xip_fpo_fixtoflt_flt_uint64	xip_fpo_exc_t	float * rop unsigned long long op	Set *rop* = *op*, rounding as required. Unsigned long long integer to single precision version.
xip_fpo_fixtoflt_d_uint64	xip_fpo_exc_t	double * rop unsigned long long op	Set *rop* = *op*, rounding as required. Unsigned long long integer to double precision version.
xip_fpo_flttoflt	xip_fpo_exc_t	xip_fpo_t rop, xip_fpo_t op	Set *rop* = *op*, rounding as required. *rop* and *op* can have different precisions.
xip_fpo_flttoflt_flt_flt	xip_fpo_exc_t	float * rop, float op	Set *rop* = *op*, rounding as required. Single to single precision version (for conditioning numbers).
xip_fpo_flttoflt_flt_d	xip_fpo_exc_t	float * rop, double op	Set *rop* = *op*, rounding as required. Double to single precision version.
xip_fpo_flttoflt_d_flt	xip_fpo_exc_t	double * rop, float op	Set *rop* = *op*, rounding as required. Single to double precision version.
xip_fpo_flttoflt_d_d	xip_fpo_exc_t	double * rop, double op	Set *rop* = *op*, rounding as required. Double to double precision version (for conditioning numbers).

Notes:

1. Only supported for `xip_fpo_t` operands with IEEE-754 half precision (exponent=5, mantissa=11) or IEEE-754 single precision (exponent=8, mantissa=24) or double precision (exponent=11, mantissa=53).

For all functions, the result is guaranteed to match exactly the numerical output of the Floating-Point Operator v7.1 core, and the returned exceptions are guaranteed to match exactly the signaled exceptions of the Floating-Point Operator v7.1 core, for identical inputs.

No direct C function is provided for Unfused Multiply Add, Unfused Multiply accumulator and Accumulator Primitive. This is because these operations are simply combinations of existing multiply and addition operations.

When the operand and result variables do not meet constraints of the Floating-Point Operator v7.1 core, an operation not supported exception is returned. In this case, no other exception bits are set in the return value, and the result variable is not modified.

`xip_fpo_condcode` functions set the 4 least significant bits of their integer result to a condition code, which has the encoding shown in Table 5-9. Encodings not shown are reserved and are not returned by the functions.

Table 5-9: **Condition Code Encoding**

Integer Result	Condition Code Bit				Meaning
	3	2	1	0	
	Unordered	Greater Than	Less Than	Equal	
1	0	0	0	1	op1 = op2
2	0	0	1	0	op1 < op2
4	0	1	0	0	*op1 > op2*
8	1	0	0	0	*op1*, *op2* or both are NaN

For all comparison functions, the sign of zero is ignored, such that –0 = +0.

xip_fpo_flttofix and xip_fpo_fixtoflt

`xip_fpo_flttofix` and `xip_fpo_fixtoflt` functions have restrictions on the precisions of the fixed-point and floating-point operand and result. The exponent width of the floating-point variable must be at least:

minimum floating-point exponent width = ceil($\log_2$(fixed-point total width + 3)) + 1

If the operand and result variable do not meet this condition, an operation not supported exception is returned and the result variable is not modified.

Compiling

Compilation of user code requires access to the `floating_point_v7_1_bitacc_cmodel.h` header file and the header files of the MPIR and MPFR dependent libraries, `gmp.h` and `mpfr.h`. The header files should be copied to a location where they are available to the compiler. Depending on the location chosen, the include search path of the compiler might need to be modified.

The `floating_point_v7_1_bitacc_cmodel.h` header file must be included first, because it defines some symbols that are used in the MPIR and MPFR header files. The `floating_point_v7_1_bitacc_cmodel.h` header file includes the MPIR and MPFR header files, so these do not need to be explicitly included in source code that uses the C model. When compiling on Windows, the symbol `NT` must be defined, either by a compiler option, or in user source code before the `floating_point_v7_1_bitacc_cmodel.h` header file is included.

Linking

To use the C model, the user executable must be linked against the correct libraries for the target platform.

Note: The C model uses MPIR and MPFR libraries. It is also possible to use GMP or MPIR, and MPFR libraries from other sources, for example, compiled from source code. For details, see Dependent Libraries.

Linux

The executable must be linked against the following shared object libraries:

- `libgmp.so.11`
- `libmpfr.so.4`
- `libIp_floating_point_v7_1_bitacc_cmodel.so`

Using GCC, linking is typically achieved by adding the following command line options:

```
-L. -Wl,-rpath,. -lIp_floating_point_v7_1_bitacc_cmodel
```

This assumes the shared object libraries are in the current directory. If this is not the case, the `-L.` option should be changed to specify the library search path to use.

Using GCC, the provided example program `run_bitacc_cmodel.c` can be compiled and linked using the following command:

```
gcc -x c++ -I. -L. -lIp_floating_point_v7_1_bitacc_cmodel -Wl,-rpath,. -o
run_bitacc_cmodel run_bitacc_cmodel.c
```

Windows

The executable must be linked against the following dynamic link libraries:

- `libgmp.dll`
- `libmpfr.dll`
- `libIp_floating_point_v7_1_bitacc_cmodel.dll`

Depending on the compiler, the import libraries might also be required:

- `libgmp.lib`
- `libmpfr.lib`
- `libIp_floating_point_v7_1_bitacc_cmodel.lib`

Using Microsoft Visual Studio, linking is typically achieved by adding the import libraries to the Additional Dependencies entry under the Linker section of Project Properties.

Dependent Libraries

The C model uses MPIR and MPFR libraries. They are governed by the GNU Lesser General Public License. You can obtain source code for the MPIR and MPFR libraries from https://www.xilinx.com/products/design-tools/guest-resources.html. Pre-compiled MPIR and MPFR libraries are provided with the C model, using the following versions of the libraries:

- MPIR 2.6.0
- MPFR 3.1.2

Because MPIR is a compatible alternative to GMP, the GMP library can be used in place of MPIR. It is possible to use GMP or MPIR and MPFR libraries from other sources, for example, compiled from source code.

GMP and MPIR in particular, and MPFR to a lesser extent, contain many low level optimizations for specific processors. The libraries provided are compiled for a generic processor on each platform, using no optimized processor-specific code. These libraries work on any processor, but run more slowly than libraries compiled to use optimized processor-specific code. For the fastest performance, compile libraries from source on the machine on which you run the executables.

Source code and compilation scripts are provided for the versions of MPIR and MPFR that were used to compile the provided libraries. Source code and compilation scripts for any version of the libraries can be obtained from the GMP [Ref 2], MPIR [Ref 3] and MPFR [Ref 4] web sites. Microsoft Visual Studio project files for compiling MPFR on Windows can be obtained from the website of Brian Gladman [Ref 5].

Note: If compiling MPIR using its `configure` script (for example, on Linux platforms), use the `--enable-gmpcompat` option when running the `configure` script. This generates a `libgmp.so` library and a `gmp.h` header file that provide full compatibility with the GMP library. This compatibility is required by the MPFR compilation scripts.

Note: Some Windows compilers, for example Microsoft Visual Studio versions prior to 2010, do not have full support for the C99 standard of the C programming language. The MPFR library contains functions that use the C99 types `intmax_t` and `uintmax_t` (for example, functions with `_sj` and `_uj` suffixes). When MPFR is compiled, it checks if these types are present, and excludes these functions if not. The C model requires these functions in MPFR. Therefore, when compiling MPFR using a Windows compiler without C99 support, include the provided `mpfr_nt_stdint.h` header file, which defines the types `intmax_t` and `uintmax_t`. Using Microsoft Visual Studio, this header file can be included without modifying source code by adding it to the Force Includes entry under the Advanced sub-section of the C/C++ section of Project Properties.

Example

The `run_bitacc_cmodel.c` file contains example code to show basic operation of the C model. Part of this example code is shown here. The comments assist in understanding the code.

This code calculates *e*, the base of natural logarithms, in the given precision. The Taylor Series expansion for the exponential function e^x is:

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots + \frac{x^n}{n!} + \ldots$$

To calculate *e*, set $x = 1$:

$$e^x = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \ldots + \frac{1}{n!} + \ldots$$

This code calculates terms iteratively until the accuracy of *e* no longer improves.

```
#include <stdio.h>
#include "floating_point_v7_1_bitacc_cmodel.h"
int main()
{
xip_fpo_exp_t exp_prec, mant_prec;
  // The algorithm will work for any legal combination
  // of values for exp_prec and mant_prec
  exp_prec  = 16;
  mant_prec = 64;
  printf("Using Taylor Series expansion to calculate e, the base of natural
logarithms, in %d-bit mantissa precision\n", mant_prec);

int i, done;
  xip_fpo_t n, fact, one, term, e, e_old;
  xip_fpo_exc_t ex;
  xip_fpo_exp_t exp;
  char * result = 0;
  double e_d;

xip_fpo_inits2 (exp_prec, mant_prec, n, fact, one, term, e,
                  e_old, (xip_fpo_ptr) 0);
  xip_fpo_set_ui (one, 1);

// 0th term
  i = 0;
  xip_fpo_set_ui (fact, 1);
  xip_fpo_set_ui (e, 1);

// Main iteration loop
  do {

// Set up this iteration
    i++;
    xip_fpo_set_ui (n, i);
    xip_fpo_set (e_old, e);

// Calculate the next term: 1/n!
```

```
    ex  = xip_fpo_mul (fact, fact, n);  // n!
    ex |= xip_fpo_div (term, one, fact);  // 1/n!
    // Note: an alternative to the preceding line is:
    //   ex |= xip_fpo_rec (term, fact);
    // but this is only possible if using single or double
    // (exp_prec, mant_prec = 8, 24 or 11, 53 respectively)
    // because xip_fpo_rec only supports single and double

// Calculate the estimate of e
    ex |= xip_fpo_add (e, e, term);

// Are we done?
    ex |= xip_fpo_equal (&done, e, e_old);

// Check for exceptions (none should occur)
    if (ex) {
      printf ("Iteration %d: exception occurred: %d\n", i, ex);
      return 1;
    }

// Print result so far
    result = xip_fpo_get_str (result, &exp, 10, 0, e);
    printf ("After %2d iteration(s), e is 0.%s * 10 ^ %d\n",
            i, result, exp);

} while (!done);

// Convert result to C's double precision type
  e_d = xip_fpo_get_d (e);
  printf ("As a C double, e is %.20f\n", e_d);

// Free up memory
  xip_fpo_clears (n, fact, one, term, e, e_old, xip_fpo_ptr) 0);
  xip_fpo_free_str (result);
return 0;
}
```

Chapter 6

Test Bench

This chapter contains information about the test bench provided in the Vivado® Design Suite.

Demonstration Test Bench

When the core is generated using the Vivado Design Suite, a demonstration test bench is created. This is a simple VHDL test bench that exercises the core.

The demonstration test bench source code is one VHDL file: `demo_tb/tb_<component_name>.vhd` in the Vivado output directory. The source code is comprehensively commented.

Using the Demonstration Test Bench

The demonstration test bench instantiates the generated Floating-Point Operator core.

Compile the netlist and the demonstration test bench into the work library (see your simulator documentation for more information on how to do this). Then simulate the demonstration test bench. View the test bench signals in your simulator's waveform viewer to see the operations of the test bench.

Demonstration Test Bench in Detail

The demonstration test bench performs the following tasks:

- Instantiates the core
- Generates input data
- Generates a clock signal
- Drives the input signals of the core to demonstrate core features
- Checks that the output signals of he core obey AXI4-Stream protocol rules (data values are not checked to keep the test bench simple)
- Provides signals showing the separate fields of AXI4-Stream TDATA and TUSER signals

The demonstration test bench drives the core input signals to demonstrate the features and modes of operation of the core. The operations performed by the demonstration test bench are appropriate for the configuration of the generated core, and are a subset of the following operations:

1. An initial phase where the core is initialized and no operations are performed.
2. Perform a single operation, and wait for the result.
3. Perform 100 consecutive operations with incrementing data.
4. Perform operations while demonstrating the AXI4-Stream control signals' use and effects.
5. If `aclken` is present: Demonstrate the effect of toggling `aclken`.
6. If `aresetn` is present: Demonstrate the effect of asserting `aresetn`.
7. Demonstrate the handling of special floating-point values (NaN, zero, infinity).

Customizing the Demonstration Test Bench

The clock frequency of the core can be modified by changing the `clock_period` constant.

For instructions on simulating your core, see the *Vivado Design Suite User Guide: Logic Simulation (UG900)* [Ref 10].

For instruction on implementing your core, see *Vivado Design Suite User Guide: Implementation (UG904)* [Ref 11].

Appendix A

Upgrading

This appendix contains information about migrating a design from ISE® to the Vivado® Design Suite, and for upgrading to a more recent version of the IP core. For customers upgrading in the Vivado Design Suite, important details (where applicable) about any port changes and other impact to user logic are included.

Migrating to the Vivado Design Suite

For information on migrating to the Vivado Design Suite, see *ISE to Vivado Design Suite Migration Guide* (UG911) [Ref 13].

In the Vivado Design Suite, you can update an existing XCO or XCI file from versions 4.0, 5.0, 6.0, 6.1, 6.2 and 7.0 to Floating-Point Operator, v7.1.

IMPORTANT: *For v4.0 and v5.0 the upgrade mechanism alone does not create a core compatible with v7.1. Floating-Point Operator v7.1 has parameters additional to v4.0 and v5.0 for AXI4-Stream support.*

Floating-Point Operator v7.1 is backwards compatible with v6.1, v6.2, and v7.0 both in terms of parameters and ports. Figure A-1 shows the changes to user parameters from versions 4.0, 5.0, 6.0, 6.1, 6.2, and 7.0 to version 7.1. For clarity, user parameters with no changes are not shown.

Upgrading in the Vivado Design Suite

This section provides information about any changes to the user logic or port designations that take place when you upgrade to a more current version of this IP core in the Vivado Design Suite.

Parameter Changes

Table A-1: **XCO/XCI Parameter Changes from v4.0, v5.0, v6.0, v6.1, v6.2, and v7.0 to v7.1**[(1)]

Version 4.0 and 5.0	Version 7.1	Notes
C_Has_CE	Has_ACLKEN	Renamed only
C_Has_SCLR	Has_ARESETn	Renamed only. While the sense of the `aresetn` signal has changed (now active-Low), this XCO parameter determined whether or not the pin exists and has not changed.
C_Latency	C_Latency	Depending on the AXI4-Stream Flow Control options selected (Blocking/ NonBlocking), a minimum latency greater than previous core versions might be imposed.
	Flow_Control	New as of version 6.0
	Axi_Optimize_Goal	New as of version 6.0
	Has_RESULT_TREADY	New as of version 6.0
	Has_A_TLAST	New as of version 6.0
	Has_A_TUSER	New as of version 6.0
	A_TUSER_Width	New as of version 6.0
	Has_B_TLAST	New as of version 6.0
	Has_B_TUSER	New as of version 6.0
	B_TUSER_Width	New as of version 6.0
	Has_OPERATION_TLAST	New as of version 6.0
	Has_OPERATION_TUSER	New as of version 6.0
	OPERATION_TUSER_Width	New as of version 6.0
	RESULT_TLAST_Behv	New as of version 6.0
	Has_C_TLAST	New as of version 6.2
	Has_C_TUSER	New as of version 6.2
	C_TUSER_Width	New as of version 6.2
	C_Has_ACCUM_INPUT_OVERFLOW	New as of version 6.2
	C_Has_ACCUM_OVERFLOW	New as of version 6.2
	C_Accum_Msb	New as of version 6.2
	C_Accum_Lsb	New as of version 6.2
	C_Accum_Input_Msb	New as of version 6.2

Notes:

1. Parameters in v6.0, v6.1, v6.2, v7.0, and v7.1 are unchanged except for the additions required for new operators.

Port Changes

Table A-2 details the changes to port naming, additional or deprecated ports and polarity changes from v4.0, v5.0, v6.0, v6.1, v6.2, and v7.0 to v7.1.

Table A-2: **Port Changes from v4.0, v5.0, v6.0, v6.1, v6.2, and v7.0 to v7.1**[(1)]

Versions 4.0 and 5.0	Version 7.1	Notes
CLK	aclk	Rename only
CE	aclken	Rename only
SCLR	aresetn	Rename and change of sense (now active-Low). Must now be asserted for at least two clock cycles to effect a reset.
A(N-1:0)	s_axis_a_tdata (byte (N)-1:0)	byte (N) is to round N up to the next multiple of 8
B(N-1:0)	s_axis_b_tdata (byte (N)-1:0)	byte (N) is to round N up to the next multiple of 8
OPERATION(5:0)	s_axis_operation_tdata (7:0)	
RESULT(R-1:0)	m_axis_result_tdata (byte (R)-1:0)	byte (R) is to round R up to the next multiple of 8.
OPERATION_ND	Deprecated	Nearest equivalents are s_axis_<operand>_tvalid
OPERATION_RFD	Deprecated	Nearest equivalents are s_axis_<operand>_tready
RDY	Deprecated	Nearest equivalent is m_axis_result_tvalid
UNDERFLOW	Deprecated	Exception signals are now subfields of m_axis_result_tuser. See Figure 3-12 for data structure.
OVERFLOW	Deprecated	
INVALID_OP	Deprecated	
DIVIDE_BY_ZERO	Deprecated	
	s_axis_a_tvalid	TVALID (AXI4-Stream channel handshake signal) for each channel
	s_axis_b_tvalid	
	s_axis_c_tvalid	
	s_axis_operation_tvalid	
	m_axis_result_tvalid	
	s_axis_a_tready	TREADY (AXI4-Stream channel handshake signal) for each channel.
	s_axis_b_tready	
	s_axis_c_tready	
	s_axis_operation_tready	
	m_axis_result_tready	

Table A-2: **Port Changes from v4.0, v5.0, v6.0, v6.1, v6.2, and v7.0 to v7.1[1] *(Cont'd)***

Versions 4.0 and 5.0	Version 7.1	Notes
	s_axis_a_tlast	TLAST (AXI4-Stream packet signal indicating the last transfer of a data structure) for each channel. The Floating-Point Operator does not use TLAST, but provides the facility to pass TLAST with the same latency as TDATA.
	s_axis_b_tlast	
	s_axis_c_tlast	
	s_axis_operation_tlast	
	m_axis_result_tlast	
	s_axis_a_tuser(E-1:0)	TUSER (AXI4-Stream ancillary field for application-specific information) for each channel. The Floating-Point Operator does not use TUSER, but provides the facility to pass TUSER with the same latency as TDATA.
	s_axis_b_tuser(F-1:0)	
	s_axis_c_tuser(G-1:0)	
	s_axis_operation_tuser(H-1:0)	
	m_axis_result_tuser(I-1:0)	

Notes:

1. Ports in v6.0, v6.1, v6.2, v7.0, and v7.1 are unchanged except for the additions required for new operators and new precision support for existing operators.

Other Changes

No change.

Debugging

This appendix includes details about resources available on the Xilinx® Support website and debugging tools.

Finding Help on Xilinx.com

To help in the design and debug process when using the Fast Fourier Transform, the Xilinx Support web page contains key resources such as product documentation, release notes, answer records, information about known issues, and links for obtaining further product support.

Documentation

This product guide is the main document associated with the Floating-Point Operator. This guide, along with documentation related to all products that aid in the design process, can be found on the Xilinx Support web page or by using the Xilinx Documentation Navigator.

Download the Xilinx Documentation Navigator from the Downloads page. For more information about this tool and the features available, open the online help after installation.

Answer Records

Answer Records include information about commonly encountered problems, helpful information on how to resolve these problems, and any known issues with a Xilinx product. Answer Records are created and maintained daily ensuring that users have access to the most accurate information available.

Answer Records for this core can be located by using the Search Support box on the main Xilinx support web page. To maximize your search results, use proper keywords such as:

- Product name
- Tool message(s)
- Summary of the issue encountered

A filter search is available after results are returned to further target the results.

Master Answer Record for the Floating-Point Operator Core

AR: 54504

Technical Support

Xilinx provides technical support in the Xilinx Support web page for this LogiCORE™ IP product when used as described in the product documentation. Xilinx cannot guarantee timing, functionality, or support if you do any of the following:

- Implement the solution in devices that are not defined in the documentation.
- Customize the solution beyond that allowed in the product documentation.
- Change any section of the design labeled DO NOT MODIFY.

To contact Xilinx Technical Support, navigate to the Xilinx Support web page.

Debug Tools

There are many tools available to address Floating-Point Operator core design issues. It is important to know which tools are useful for debugging various situations.

Vivado Design Suite Debug Feature

The Vivado® Design Suite debug feature inserts logic analyzer and virtual I/O cores directly into your design. The debug feature also allows you to set trigger conditions to capture application and integrated block port signals in hardware. Captured signals can then be analyzed. This feature in the Vivado IDE is used for logic debugging and validation of a design running in Xilinx devices.

The Vivado logic analyzer is used with the logic debug LogiCORE IP cores, including:

- ILA 2.0 (and later versions)
- VIO 2.0 (and later versions)

See *Vivado Design Suite User Guide: Programming and Debugging* (UG908) [Ref 14].

C Model Reference

See *Chapter 5, C Model* in this guide for tips and instructions for using the provided C-Model files to debug your design.

Simulation Debug

When the Floating-Point Operator core is configured to use Non-Blocking mode, any transaction input in the cycle following the deassertion of `aresetn` is lost. For more information, see Non-Blocking Mode in the AXI4-Stream Considerations section.

AXI4-Stream Interface Debug

If data is not being transmitted or received, check the following conditions:

- If transmit <*interface_name*>_`tready` is stuck Low following the <*interface_name*>_`tvalid` input being asserted, the core cannot send data.
- If the receive <*interface_name*>_`tvalid` is stuck Low, the core is not receiving data.
- Check that the `aclk` inputs are connected and toggling.
- Check that the AXI4-Stream waveforms are being followed. See AXI4-Stream Considerations.

Appendix C

Additional Resources and Legal Notices

Xilinx Resources

For support resources such as Answers, Documentation, Downloads, and Forums, see Xilinx Support.

Documentation Navigator and Design Hubs

Xilinx® Documentation Navigator provides access to Xilinx documents, videos, and support resources, which you can filter and search to find information. To open the Xilinx Documentation Navigator (DocNav):

- From the Vivado® IDE, select **Help > Documentation and Tutorials**.
- On Windows, select **Start > All Programs > Xilinx Design Tools > DocNav**.
- At the Linux command prompt, enter docnav.

Xilinx Design Hubs provide links to documentation organized by design tasks and other topics, which you can use to learn key concepts and address frequently asked questions. To access the Design Hubs:

- In the Xilinx Documentation Navigator, click the **Design Hubs View** tab.
- On the Xilinx website, see the Design Hubs page.

Note: For more information on Documentation Navigator, see the Documentation Navigator page on the Xilinx website.

References

These documents provide supplemental material useful with this product guide:

1. *ANSI/IEEE, IEEE Standard for Binary Floating-Point Arithmetic,* ANSI/IEEE Standard 754-2008. IEEE-754.

2. The GNU Multiple Precision Arithmetic (GMP) Library gmplib.org
3. The GNU Multiple Precision Integers and Rationals (MPIR) library www.mpir.org
4. The GNU Multiple Precision Floating-Point Reliable (MPFR) Library www.mpfr.org
5. Multiple Precision Arithmetic on Windows, Brian Gladman: http://mpir.org/index.html
6. *Vivado Design Suite User Guide: Designing with IP* (UG896)
7. *Vivado Design Suite User Guide: Getting Started* (UG910)
8. *Vivado Design Suite User Guide: Designing IP Subsystems Using IP Integrator* (UG994)
9. *System Generator for DSP User Guide* (UG640)
10. *Vivado Design Suite User Guide: Logic Simulation* (UG900)
11. *Vivado Design Suite User Guide: Implementation* (UG904)
12. *Vivado Design Suite User Guide: Using the Vivado IDE* (UG893)
13. *ISE to Vivado Design Suite Migration Methodology Guide* (UG911)
14. *Vivado Design Suite User Guide: Programming and Debugging* (UG908)
15. *Xilinx Vivado AXI Reference Guide* (UG1037)
16. *AMBA® AXI4-Stream Protocol Specification* (Arm IHI 0051A)
17. Florent de Dinechin, Bogdan Pasca, Octavian Cret, Radu Tudoran. *An FPGA-specific Approach to Floating-Point Accumulation and Sum-of-Products*. Field-Programmable Technology, Dec 2008, Taipei, Taiwan. ffensl-00268348v3 https://hal-ens-lyon.archives-ouvertes.fr/ensl-00268348v3/document
18. *Versal ACAP DSP Engine Architecture Manual* (AM004)

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
12/16/2020	7.1	• Added Versal ACAP product support. • Added Accumulator Primitive section.
10/30/2019	7.1	• Internal Precision Tab section updated. • Latency and Rate Configuration section updated.
10/04/2017	7.1	Fixed Point Number Representation section updated.
11/18/2015	7.1	UltraScale+ device support added.
09/30/2015	7.1	• Addition of Half Precision support for Reciprocal, Reciprocal Square Root, Exponential and Logarithm operators.

Date	Version	Revision
04/02/2014	7.0	• Added link to resource utilization figures. • Updated template.
12/18/2013	7.0	• Added UltraScale and IP Integrator support. • Added Simulation, Synthesis, Example Design, and Test Bench chapters. • Updated Migrating chapter and Debugging Appendix.
10/02/2013	7.0	Minor updates to IP Facts table and Migrating appendix. Document version number advanced to match the core version number.
03/20/2013	3.0	• Updated for core v7.0. • Added support for Zynq-7000 devices.
12/18/2012	2.0	• Updated to core v6.2 and Vivado Design Suite 2012.4. • Removed support for ISE Design Suite, and Virtex-6, Spartan-6 and Zynq-7000 devices. • Added support for accumulator, fused multiply-add and exponential operators. • Updated Resource Utilization numbers. • Added C Input Channel. • Added Appendix B, Debugging.
07/25/2012	1.0	• Initial Xilinx release. This Product Guide is derived from DS816 and UG812.

Please Read: Important Legal Notices

The information disclosed to you hereunder (the "Materials") is provided solely for the selection and use of Xilinx products. To the maximum extent permitted by applicable law: (1) Materials are made available "AS IS" and with all faults, Xilinx hereby DISCLAIMS ALL WARRANTIES AND CONDITIONS, EXPRESS, IMPLIED, OR STATUTORY, INCLUDING BUT NOT LIMITED TO WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, OR FITNESS FOR ANY PARTICULAR PURPOSE; and (2) Xilinx shall not be liable (whether in contract or tort, including negligence, or under any other theory of liability) for any loss or damage of any kind or nature related to, arising under, or in connection with, the Materials (including your use of the Materials), including for any direct, indirect, special, incidental, or consequential loss or damage (including loss of data, profits, goodwill, or any type of loss or damage suffered as a result of any action brought by a third party) even if such damage or loss was reasonably foreseeable or Xilinx had been advised of the possibility of the same. Xilinx assumes no obligation to correct any errors contained in the Materials or to notify you of updates to the Materials or to product specifications. You may not reproduce, modify, distribute, or publicly display the Materials without prior written consent. Certain products are subject to the terms and conditions of Xilinx's limited warranty, please refer to Xilinx's Terms of Sale which can be viewed at https://www.xilinx.com/legal.htm#tos; IP cores may be subject to warranty and support terms contained in a license issued to you by Xilinx. Xilinx products are not designed or intended to be fail-safe or for use in any application requiring fail-safe performance; you assume sole risk and liability for use of Xilinx products in such critical applications, please refer to Xilinx's Terms of Sale which can be viewed at https://www.xilinx.com/legal.htm#tos.

AUTOMOTIVE APPLICATIONS DISCLAIMER

AUTOMOTIVE PRODUCTS (IDENTIFIED AS “XA” IN THE PART NUMBER) ARE NOT WARRANTED FOR USE IN THE DEPLOYMENT OF AIRBAGS OR FOR USE IN APPLICATIONS THAT AFFECT CONTROL OF A VEHICLE (“SAFETY APPLICATION”) UNLESS THERE IS A SAFETY CONCEPT OR REDUNDANCY FEATURE CONSISTENT WITH THE ISO 26262 AUTOMOTIVE SAFETY STANDARD (“SAFETY DESIGN”). CUSTOMER SHALL, PRIOR TO USING OR DISTRIBUTING ANY SYSTEMS THAT INCORPORATE PRODUCTS, THOROUGHLY TEST SUCH SYSTEMS FOR SAFETY PURPOSES. USE OF PRODUCTS IN A SAFETY APPLICATION WITHOUT A SAFETY DESIGN IS FULLY AT THE RISK OF CUSTOMER, SUBJECT ONLY TO APPLICABLE LAWS AND REGULATIONS GOVERNING LIMITATIONS ON PRODUCT LIABILITY.