LogiCORE IP Floating-Point Operator v6.1

Product Guide

PG060 July 25, 2012

Table of Contents

SECTION I: SUMMARY

Introduction

The Xilinx® Floating-Point Operator core provides designers 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
  - divide
  - square-root
  - comparison
  - reciprocal
  - reciprocal square root
  - absolute value
  - natural logarithm
  - conversion from floating-point to fixed-point
  - conversion from fixed-point to floating-point
  - conversion between floating-point types
- Compliance with *IEEE-754 Standard* [Ref 1] (with only minor documented deviations)
- Parameterized fraction and exponent wordlengths for most operators
- Use of XtremeDSP™ slice for multiply, single and double precision add/subtract operations
- Optimizations for speed and latency
- Fully synchronous design using a single clock

LogiCORE IP Facts Table	
Core Specifics	
Supported Device Family[1]	Zynq™-7000[2], Virtex®-7, Kintex™-7, Artix™-7, Virtex-6, Spartan®-6[3]
Supported User Interfaces	AXI4-Stream
Resources	See Table 2-16 to Table 2-24.
Provided with Core	
Design Files	ISE: Netlist Vivado: Encrypted RTL
Example Design	Not Provided
Test Bench	VHDL
Constraints File	Not Provided
Simulation Model	Verilog VHDL C Model
Supported S/W Driver	N/A
Tested Design Flows[4]	
Design Entry	Vivado™ Design Suite[5] ISE Design Suite: CORE Generator™ tool ISE Design Suite: System Generator for DSP
Simulation	Mentor Graphics ModelSim Cadence Incisive Enterprise Simulator (IES) Synopsys VCS and VCS MX ISim Vivado Simulator
Synthesis	Xilinx Synthesis Technology (XST) Vivado Synthesis
Support	
Provided by Xilinx @ www.xilinx.com/support	

Notes:

1. For a complete listing of supported devices, see the release notes for this core.
2. Supported in ISE Design Suite implementations only.
3. Spartan 6 is not supported on all Floating Point Operators.
4. For the supported versions of the tools, see the Xilinx Design Tools: Release Notes Guide.
5. Supports 7 series devices only.

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. When a user selects an operation that requires only one operand, the B input channel is omitted.

Unsupported Features

See Standards Compliance.

Licensing and Ordering Information

This Xilinx LogiCORE™ IP module is provided at no additional cost with the Xilinx Vivado™ Design Suite and ISE® Design Suite tools 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.

XILINX

Chapter 2

Product Specification

Standards Compliance

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 greater range of fraction and exponent wordlength than defined in the *IEEE-754 Standard*.

Standard formats commonly implemented by programmable processors:

- **Single Format** – uses 32 bits, with a 24-bit fraction and 8-bit exponent.
- **Double Format** – uses 64 bits, with 53-bit fraction and 11-bit exponent.

Less commonly implemented standard formats are:

- **Single Extended** – wordlength extensions of 43 bits and above
- **Double Extended** – wordlength extensions of 79 bits and above

The Xilinx core supports formats with fraction and exponent wordlengths outside of these standard wordlengths.

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 currently supported. 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.

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, divide, square-root and conversion operators. The reciprocal, reciprocal square-root and natural logarithm operators produce results which are accurate to one ULP.

Performance

Latency

The latency of all 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 is tabulated for a range of width and operation types in Tables 2-1 through 2-14. The latency values in these tables represent the fully-pipelined latency of the internal Floating-Point Operator core. They do not include additional latency overhead due to AXI4-Stream interface logic required when using a Blocking flow control scheme.

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 (see Operation Selection for further details).

Table 2-1: **Latency of Floating-Point Multiplication Using Logic Only**

Fraction Width	Maximum Latency (Clock Cycles)
4 to 5	5
6 to 11	6
12 to 23	7

Table 2-1: **Latency of Floating-Point Multiplication Using Logic Only *(Cont'd)***

Fraction Width	Maximum Latency (Clock Cycles)
24 to 47 (inc. single)	8
48 to 64 (inc. double)	9

Table 2-2: **Latency of Floating-Point Multiplication Using DSP48A1**

Fraction Width	Maximum Latency (Clock Cycles)		
	Medium Usage	Full Usage	Max Usage
4 to 17		6	5
18 to 34 (inc. single)	9[1]	11	10
35 to 51		18	17
52 to 64 (inc. double)		27	26

1. Single precision only.

Table 2-3: **Latency of Floating-Point Multiplication Using DSP48E1**

Fraction Width	Maximum Latency (Clock Cycles)		
	Medium Usage	Full Usage	Max Usage
single	8	8	6
double	15	15	16
4 to 17		6	8
18 to 24		8	9
25 to 34		10	11
35 to 41		12	13
42 to 51		15	16
52 to 58		18	19
59 to 64		22	23

Table 2-4: **Latency of Floating-Point Multiplication Using DSP48E1 and Low Latency Optimization**

Fraction Width	Maximum Latency (Clock Cycles)
	Max Usage
double	10

Table 2-5: **Latency of Floating-Point Addition Using Full Usage and DSP48E1**

Width	Maximum Latency (Clock Cycles)
single	11
double	14

Table 2-6: **Latency of Floating-Point Addition Using Logic and Low-Latency Optimization on Virtex-6 and 7 Series FPGAs**

Fraction Width	Maximum Latency (Clock Cycles)
single	8
double	8

Table 2-7: **Latency of Floating-Point Addition Using Logic and Speed Optimization on 7 Series, Virtex-6 and Spartan-6 FPGAs**

Fraction Width	Maximum Latency (Clock Cycles)
4 to 13	8
14	9
15	10
16, 17	11
18 to 61 (single, double)	12
62 to 64	13

Table 2-8: **Latency of Fixed-Point to Floating-Point Conversion**

Operand Width	Maximum Latency (Cycles)
4 to 8	5
9 to 32	6
33 to 64	7

Table 2-9: **Latency of Floating-Point to Fixed-Point Conversion**

Maximum of (A Fraction Width+1) and Result Width	Maximum Latency (Cycles)
5 to 16	5
17 to 64	6
65	7

Table 2-10: **Latency of Floating-Point Reciprocal Using DSP48E1**

Fraction Width	Maximum Latency (Clock Cycles)	
	No Usage	Full Usage
single	36	29
double		35

Table 2-11: **Latency of Floating-Point Reciprocal Using DSP48A1**

Fraction Width	Maximum Latency (Clock Cycles)	
	No Usage	Full Usage
single	36	33
double		43

Table 2-12: **Latency of Floating-Point Reciprocal Square Root Using DSP48E1**

Fraction Width	Maximum Latency (Clock Cycles)	
	No Usage	Full Usage
single	37	32
double		112

Table 2-13: **Latency of Floating-Point Absolute Value**

Fraction Width	Maximum Latency (Clock Cycles)
single, double, custom	0

Table 2-14: **Latency of Floating-Point Natural Logarithm Using DSP48E1**

Fraction Width	Maximum Latency (Clock Cycles)		
	No Usage	Medium Usage	Full Usage
single	23	22	28
double	37	52	67

Resource Utilization

The resource requirements and maximum clock rates achievable on Kintex™-7, Artix-7 and Zynq-7000 FPGAs are summarized as follows for the case of maximum latency and no `aresetn` or `aclken` pins. Unless otherwise stated, Non-Blocking flow control is used for all configurations. For selected use cases, figures are provided for the Blocking and Performance flow control configuration which permits backpressure.

Note: Both LUT and FF resource usage and maximum frequency reduce with latency. Minimizing latency minimizes resources.

The maximum clock frequency results were obtained by double-registering input and output ports to reduce dependence on I/O placement. The inner level of registers used a separate clock signal to measure the path from the input registers to the first output register through the core.

The resource usage results do not include the "characterization" registers and represent the true logic used by the core. LUT counts include SRL16s or SRL32s.

The map options used were: "`map -ol high`."

The par options used were: "`par -ol high`."

Clock frequency does not take clock jitter into account and should be derated by an amount appropriate to the clock source jitter specification.

The maximum achievable clock frequency and the resource counts might also be affected by other tool options, additional logic in the FPGA, using a different version of Xilinx tools, and other factors.

It is possible to improve performance of the Xilinx Floating-Point Operator within a system context by placing the operator within an area group. Placement of both the logic slices and XtremeDSP™ slices can be contained in this way. If multiply-add operations are used, then placing them in the same group can be helpful. Groups can also include any supporting logic to ensure that it is placed close to the operators.

All results were produced using ISE 14.2 software.

Table 2-15: **Speed File Version**

FPGA Family	Speed File Version
Kintex-7	ADVANCED 1.05a 2012-05-29
Artix-7	ADVANCED 1.04b 2012-05-29
Zynq-7000	ADVANCED 1.01 2012-05-29

Custom Format: 17-Bit Fraction and 24-Bit Total Wordlength

The resource requirements and maximum clock rates achievable with 17-bit fraction and 24-bit total wordlength on Kintex-7 are summarized in Table 2-16.

Table 2-16: **Characterization of 17-Bit Fraction and 24-Bit Total Wordlength on Kintex-7 FPGA**

Operation	Resources[1]					Maximum Frequency (MHz)[2][3]
	Embedded		FPGA Logic			Kintex-7
	Type	Number	LUT-FF Pairs	LUTs	FFs	-1 Speed Grade
Multiply	DSP48E1 (max usage)	2	175	94	190	445
	DSP48E1 (full usage)	1	172	109	181	463
	Logic (no usage)		441	348	453	424
Add/Subtract	Logic (no usage)		448	317	482	518
Fixed to float	Int24 input		213	150	191	424
Float to fixed	Int24 result		209	147	235	>550
Float to float	Single to 24-17 format		124	68	139	529
	24-17 to single		79	35	108	>550
Compare	Programmable		81	56	73	>550
Divide	RATE=1		749	491	805	475
	RATE=19		233	184	231	386
Square Root	RATE=1		444	335	513	>550
	RATE=18		166	111	200	541
Absolute Value	Any width	0	0	0	0	>550
Multiply Flow Control: Blocking, Optimize Goal: Performance	DSP48E1 (max usage)	2	288	173	325	434
Add/Subtract Flow Control: Blocking, Optimize Goal: Performance	Logic (no usage)		558	401	613	500

Notes:

1. The device used for these figures is an XC7K70T-1.
2. Area and maximum clock frequencies are provided as a guide and might vary with new releases of the Xilinx implementation tools.
3. Maximum clock frequencies are shown in MHz. Clock frequency does not take jitter into account and should be de-rated by an amount appropriate to the clock source jitter specification.
4. The absolute value operator uses neither logic or registers so will not become the critical path in any realistic circuit.

The resource requirements and maximum clock rates achievable with 17-bit fraction and 24-bit total wordlength on Artix-7 are summarized in Table 2-17.

Table 2-17: **Characterization of 17-Bit Fraction and 24-Bit Total Wordlength on Artix-7 FPGAs**

Operation	Resources[1]					Maximum Frequency (MHz)[2][3]
	Embedded		FPGA Logic			Artix-7
	Type	Number	LUT-FF Pairs	LUTs	FFs	-1 Speed Grade
Multiply	DSP48E1 (max usage)	2	193	80	190	326
	DSP48E1 (full usage)	1	182	102	181	366
	Logic (no usage)		455	338	453	265
Add/Subtract	Logic (no usage)		469	301	480	326
Fixed to float	Int24 input		211	149	191	299
Float to fixed	Int24 result		208	147	233	378
Float to float	Single to 24-17 format		131	70	139	338
	24-17 to single		83	33	108	456
Compare	Programmable		95	48	73	373
Divide	RATE=1		745	472	805	347
	RATE=19		258	177	231	255
Square Root	RATE=1		440	334	513	368
	RATE=18		161	110	200	332
Absolute Value	Any width	0	0	0	0	464
Multiply Flow Control: Blocking, Optimize Goal: Performance	DSP48E1 (max usage)	2	302	157	325	322
Add/Subtract Flow Control: Blocking, Optimize Goal: Performance	Logic (no usage)		538	398	611	329

Notes:

1. The device used for these figures is an XC7A100T-1.
2. Area and maximum clock frequencies are provided as a guide and might vary with new releases of the Xilinx implementation tools.
3. Maximum clock frequencies are shown in MHz. Clock frequency does not take jitter into account and should be de-rated by an amount appropriate to the clock source jitter specification.
4. The absolute value operator uses no logic nor registers so will not become the critical path in any realistic circuit.

The resource requirements and maximum clock rates achievable with 17-bit fraction and 24-bit total wordlength on Zynq-7000 FPGAs are summarized in Table 2-18.

Table 2-18: **Characterization of 17-Bit Fraction and 24-Bit Total Wordlength on Zynq-7000 Devices**

Operation	Resources(1)					Maximum Frequency (MHz)(2)(3)
	Embedded		FPGA Logic			Zynq-7000
	Type	Number	LUT-FF Pairs	LUTs	FFs	-1 Speed Grade
Multiply	DSP48E1 (max usage)	2	189	80	190	413
	DSP48E1 (full usage)	1	191	96	181	463
	Logic (no usage)	0	453	338	453	430
Add/Subtract	Logic (no usage)		463	300	482	464
Fixed to float	Int24 input		213	147	191	447
Float to fixed	Int24 result		214	143	235	464
Float to float	Single to 24-17 format		134	67	139	464
	24-17 to single		84	34	108	464
Compare	Programmable		92	49	73	464
Divide	RATE=1		674	482	805	464
	RATE=19		259	179	231	390
Square Root	RATE=1		462	332	513	464
	RATE=18		165	111	200	464
Absolute Value	Any width	0	0	0	0	464
Multiply Flow Control: Blocking, Optimize Goal: Performance	DSP48E1 (max usage)	2	266	175	325	435
Add/Subtract Flow Control: Blocking, Optimize Goal: Performance	Logic (no usage)		545	401	613	462

Notes:

1. The device used for these figures is an XC7Z045-1.
2. Area and maximum clock frequencies are provided as a guide and might vary with new releases of the Xilinx implementation tools.
3. Maximum clock frequencies are shown in MHz. Clock frequency does not take jitter into account and should be de-rated by an amount appropriate to the clock source jitter specification.
4. The absolute value operator uses no logic nor registers so will not become the critical path in any realistic circuit.

Single-Precision Format

The resource requirements and maximum clock rates achievable with single-precision format on Kintex-7 FPGAs is summarized in Table 2-19.

Table 2-19: **Characterization of Single-Precision Format on Kintex-7 FPGAs**

Operation	Resources[1]					Maximum Frequency (MHz)[2][3]
	Embedded		FPGA Logic			Kintex-7
	Type	Number	LUT-FF Pairs	LUTs	FFs	-1 Speed Grade
Multiply	DSP48E1 (max usage)	3	162	133	206	463
	DSP48E1 (full usage)	2	252	129	261	463
	DSP48E1 (medium usage)	1	390	303	432	463
	Logic	0	709	657	770	471
Add/Subtract	DSP48E1 (speed optimized, full usage)	2	390	243	441	423
	Logic (speed optimized, no usage)	0	629	412	655	486
	Logic (low latency)	0	549	504	726	485
Fixed to float	Int32 input		271	178	289	544
Float to fixed	Int32 result		271	189	297	537
Float to float	Single to double		121	35	142	>550
Compare	Programmable		107	68	89	>550
Divide	RATE=1		1324	847	1442	423
	RATE=26		265	222	294	372
Square Root	RATE=1		709	540	878	430
	RATE=25		209	144	263	411
Reciprocal	DSP48E1 (full usage)	8	283	187	323	529
	Logic (no usage)	0	1371	1264	1285	343
Reciprocal Square Root	DSP48E1 (full usage)	9	500	388	474	533
	Logic (no usage)	0	2148	2044	1978	417
Absolute Value	N/A	0	0	0	0	>550
Natural Logarithm	DSP48E1 (full usage)	13	817	609	932	482
	DSP48E1 (medium usage)	4	1028	871	1161	427
	Logic	0	1611	1437	1678	441
Multiply Flow Control: Blocking, Optimize Goal: Performance	DSP48E1 (max usage)	3	342	218	382	463

Table 2-19: **Characterization of Single-Precision Format on Kintex-7 FPGAs *(Cont'd)***

Operation	Resources[1]					Maximum Frequency (MHz)[2][3]
	Embedded		FPGA Logic			Kintex-7
	Type	Number	LUT-FF Pairs	LUTs	FFs	-1 Speed Grade
Add/Subtract Flow Control: Blocking, Optimize Goal: Performance	DSP48E1 (speed optimized, full usage)	2	515	354	611	423

Notes:

1. The device used for these figures is an XC7K70T-1.
2. Area and maximum clock frequencies are provided as a guide and might vary with new releases of the Xilinx implementation tools.
3. Maximum clock frequencies are shown in MHz. Clock frequency does not take jitter into account and should be de-rated by an amount appropriate to the clock source jitter specification.
4. The absolute value operator uses no logic nor registers so will not become the critical path in any realistic circuit.

The resource requirements and maximum clock rates achievable with single-precision format on Artix-7 FPGAs is summarized in Table 2-20.

Table 2-20: **Characterization of Single-Precision Format on Artix-7 FPGAs**

Operation	Resources[1]					Maximum Frequency (MHz)[2][3]
	Embedded		FPGA Logic			Artix-7
	Type	Number	LUT-FF Pairs	LUTs	FFs	-1 Speed Grade
Multiply	DSP48E1 (max usage)	3	207	113	206	378
	DSP48E1 (full usage)	2	251	137	261	354
	DSP48E1 (medium usage)	1	422	294	430	301
	Logic	0	754	630	770	319
Add/Subtract	DSP48E1 (speed optimized, full usage)	2	419	217	441	282
	Logic (speed optimized, no usage)	0	627	414	651	347
	Logic (low latency)	0	668	508	726	323
Fixed to float	Int32 input		244	198	288	348
Float to fixed	Int32 result		278	181	295	339
Float to float	Single to double		124	30	142	417
Compare	Programmable		117	60	89	371
Divide	RATE=1		1087	847	1444	323
	RATE=26		331	220	294	254
Square Root	RATE=1		676	555	875	338
	RATE=25		200	141	261	331
Reciprocal	DSP48E1 (full usage)	8	289	176	321	366
	Logic (no usage)	0	1378	1269	1285	280
Reciprocal Square Root	DSP48E1 (full usage)	9	498	386	474	334
	Logic (no usage)	0	2170	2039	1978	279
Absolute Value	N/A	0	0	0	0	464
Natural Logarithm	DSP48E1 (full usage)	13	810	601	934	318
	DSP48E1 (medium usage)	4	1111	866	1160	272
	Logic	0	1650	1473	1657	273
Multiply Flow Control: Blocking, Optimize Goal: Performance	DSP48E1 (max usage)	3	349	212	382	366

Table 2-20: **Characterization of Single-Precision Format on Artix-7 FPGAs *(Cont'd)***

Operation	Resources[1]					Maximum Frequency (MHz)[2][3]
	Embedded		FPGA Logic			Artix-7
	Type	Number	LUT-FF Pairs	LUTs	FFs	-1 Speed Grade
Add/Subtract Flow Control: Blocking, Optimize Goal: Performance	DSP48E1 (speed optimized, full usage)	2	542	327	611	299

Notes:

1. The device used for these figures is an XC7A100T-1.
2. Area and maximum clock frequencies are provided as a guide and might vary with new releases of the Xilinx implementation tools.
3. Maximum clock frequencies are shown in MHz. Clock frequency does not take jitter into account and should be de-rated by an amount appropriate to the clock source jitter specification.
4. The absolute value operator uses no logic nor registers so will not become the critical path in any realistic circuit.

The resource requirements and maximum clock rates achievable with single-precision format on Zynq-7000 FPGAs is summarized in Table 2-21.

Table 2-21: **Characterization of Single-Precision Format on Zynq-7000 Devices**

Operation	Resources(1)					Maximum Frequency (MHz)(2)(3)
	Embedded		FPGA Logic			Zynq-7000
	Type	Number	LUT-FF Pairs	LUTs	FFs	-1 Speed Grade
Multiply	DSP48E1 (max usage)	3	198	121	206	463
	DSP48E1 (full usage)	2	253	133	261	463
	DSP48E1 (medium usage)	1	433	292	432	463
	Logic	0	754	636	770	463
Add/Subtract	DSP48E1 (speed optimized, full usage)	2	412	226	441	432
	Logic (speed optimized, no usage)	0	624	413	655	461
	Logic (low latency)	0	689	505	726	464
Fixed to float	Int32 input		258	194	289	464
Float to fixed	Int32 result		281	181	297	464
Float to float	Single to double		131	25	142	464
Compare	Programmable		108	66	89	464
Divide	RATE=1		1123	845	1442	462
	RATE=26		326	230	294	399
Square Root	RATE=1		713	551	878	423
	RATE=25		212	140	263	445
Reciprocal	DSP48E1 (full usage)	8	303	176	323	464
	Logic (no usage)	0	1394	1262	1285	430
Reciprocal Square Root	DSP48E1 (full usage)	9	495	384	474	464
	Logic (no usage)	0	2163	2041	1978	431
Absolute Value	N/A	0	0	0	0	464
Natural Logarithm	DSP48E1 (full usage)	13	841	590	932	462
	DSP48E1 (medium usage)	4	1116	881	1158	416
	Logic	0	1659	1469	1678	375
Multiply Flow Control: Blocking, Optimize Goal: Performance	DSP48E1 (max usage)	3	357	208	382	462

Table 2-21: **Characterization of Single-Precision Format on Zynq-7000 Devices *(Cont'd)***

Operation	Resources[1]					Maximum Frequency (MHz)[2][3]
	Embedded		FPGA Logic			Zynq-7000
	Type	Number	LUT-FF Pairs	LUTs	FFs	-1 Speed Grade
Add/Subtract Flow Control: Blocking, Optimize Goal: Performance	DSP48E1 (speed optimized, full usage)	2	542	330	611	426

Notes:

1. The device used for these figures is an XC7Z045-1.
2. Area and maximum clock frequencies are provided as a guide and might vary with new releases of the Xilinx implementation tools.
3. Maximum clock frequencies are shown in MHz. Clock frequency does not take jitter into account and should be de-rated by an amount appropriate to the clock source jitter specification.
4. The absolute value operator uses no logic nor registers so will not become the critical path in any realistic circuit.

Double-Precision Format

The resource requirements and maximum clock rates achievable with double-precision format on Kintex-7 FPGAs are summarized in Table 2-22.

Table 2-22: **Characterization of Double-Precision Format on Kintex-7 FPGAs**

Operation	Resources[1]						Maximum Frequency (MHz)[2][3]
	Embedded		FPGA Logic			18k Block RAMs	Kintex-7
	Type	Number	LUT-FF Pairs	LUTs	FFs		-1 Speed Grade
Multiply	DSP48E1 (max usage)	11	452	340	620	0	463
	DSP48E1 (full usage)	10	537	336	653	0	454
	DSP48E1 (medium usage)	9	607	390	712	0	404
	Logic	0	2470	2332	2615	0	329
	DSP48E1 (low latency, max usage)	13	518	218	494	0	384
Add/Subtract	DSP48E1 (speed optimized, full usage)	3	1065	745	1157	0	444
	Logic (speed optimized, no usage)	0	1195	803	1244	0	449
	Logic (low latency, no usage)	0	1260	989	1371	0	400
Fixed to float	Int64 input		458	437	618	0	405
Float to fixed	Int64 result		475	394	577	0	435
Float to float	Double to single		213	87	202	0	499
Compare	Programmable		171	141	153	0	482
Divide	RATE=1		3849	3524	6166	0	344
	RATE=55		612	440	550	0	266
Square Root	RATE=1		2239	2067	3423	0	317
	RATE=54		382	322	512	0	314
Reciprocal	DSP48E1 (full usage)	14	522	381	650	0	411
Reciprocal Square Root	DSP48E1 (full usage)	75	2336	1844	2781	1	392
Absolute Value	N/A	0	0	0	0	0	>550
Natural Logarithm	DSP48E1 (full usage)	61	2737	2011	3484	0	417
	DSP48E1 (medium usage)	23	3564	3085	3803	0	332
	Logic	0	5667	5476	5649	0	302
Multiply Flow Control: Blocking, Optimize Goal: Performance	DSP48E1 (max usage)	11	781	484	984	0	409

Table 2-22: **Characterization of Double-Precision Format on Kintex-7 FPGAs *(Cont'd)***

Operation	Resources[1]						Maximum Frequency (MHz)[2][3]
	Embedded		FPGA Logic			18k Block RAMs	Kintex-7
	Type	Number	LUT-FF Pairs	LUTs	FFs		-1 Speed Grade
Add/Subtract Flow Control: Blocking, Optimize Goal: Performance	DSP48E1 (speed optimized, full usage)	3	1335	910	1485	0	450

Notes:

1. The device used for these figures is an XC7K70-1.
2. Area and maximum clock frequencies are provided as a guide and might vary with new releases of the Xilinx implementation tools.
3. Maximum clock frequencies are shown in MHz. Clock frequency does not take jitter into account and should be de-rated by an amount appropriate to the clock source jitter specification.
4. The absolute value operator uses no logic nor registers so will not become the critical path in any realistic circuit.

The resource requirements and maximum clock rates achievable with double-precision format on Artix-7 FPGAs are summarized in Table 2-23.

Table 2-23: **Characterization of Double-Precision Format on Artix-7 FPGAs**

Operation	Resources[1]						Maximum Frequency (MHz)[2][3]
	Embedded		FPGA Logic			18k Block RAMs	Artix-7
	Type	Number	LUT-FF Pairs	LUTs	FFs		-1 Speed Grade
Multiply	DSP48E1 (max usage)	11	502	303	620	0	374
	DSP48E1 (full usage)	10	552	313	653	0	314
	DSP48E1 (medium usage)	9	568	424	693	0	284
	Logic	0	2538	2329	2615	0	191
	DSP48E1 (low latency, max usage)	13	500	234	494	0	287
Add/Subtract	DSP48E1 (speed optimized, full usage)	3	1059	761	1157	0	335
	Logic (speed optimized, no usage)	0	1228	804	1224	0	299
	Logic (low latency, no usage)	0	1235	982	1373	0	255
Fixed to float	Int64 input		594	419	618	0	251
Float to fixed	Int64 result		541	370	577	0	258
Float to float	Double to single		195	105	202	0	322
Compare	Programmable		200	114	153	0	319
Divide	RATE=1		3731	3443	6166	0	220
	RATE=55		623	421	550	0	200
Square Root	RATE=1		2447	2028	3414	0	245
	RATE=54		422	289	511	0	241
Reciprocal	DSP48E1 (full usage)	14	539	368	650	0	284
Reciprocal Square Root	DSP48E1 (full usage)	75	2337	1869	2759	1	241
Absolute Value	N/A	0	0	0	0	0	464
Natural Logarithm	DSP48E1 (full usage)	61	2748	2001	3465	0	264
	DSP48E1 (medium usage)	23	3564	3165	3810	0	203
	Logic	0	5956	5484	5656	0	200
Multiply Flow Control: Blocking, Optimize Goal: Performance	DSP48E1 (max usage)	11	750	512	984	0	331

Table 2-23: **Characterization of Double-Precision Format on Artix-7 FPGAs *(Cont'd)***

Operation	Resources[1]						Maximum Frequency (MHz)[2][3]
	Embedded		FPGA Logic			18k Block RAMs	Artix-7
	Type	Number	LUT-FF Pairs	LUTs	FFs		-1 Speed Grade
Add/Subtract Flow Control: Blocking, Optimize Goal: Performance	DSP48E1 (speed optimized, full usage)	3	1294	953	1485	0	302

Notes:

1. The device used for these figures is an XC7A100T-1.
2. Area and maximum clock frequencies are provided as a guide and might vary with new releases of the Xilinx implementation tools.
3. Maximum clock frequencies are shown in MHz. Clock frequency does not take jitter into account and should be de-rated by an amount appropriate to the clock source jitter specification.
4. The absolute value operator uses no logic nor registers so will not become the critical path in any realistic circuit.

The resource requirements and maximum clock rates achievable with double-precision format on Zynq-7000 FPGAs are summarized in Table 2-24.

Table 2-24: **Characterization of Double-Precision Format on Zynq-7000 Devices**

Operation	Resources(1)						Maximum Frequency (MHz)(2)(3)
	Embedded		FPGA Logic			18k Block RAMs	Zynq-7000
	Type	Number	LUT-FF Pairs	LUTs	FFs		-1 Speed Grade
Multiply	DSP48E1 (max usage)	11	498	308	620	0	463
	DSP48E1 (full usage)	10	570	301	653	0	430
	DSP48E1 (medium usage)	9	623	373	703	0	382
	Logic	0	2573	2320	2615	0	334
	DSP48E1 (low latency, max usage)	13	486	253	494	0	382
Add/Subtract	DSP48E1 (speed optimized, full usage)	3	1114	717	1157	0	435
	Logic (speed optimized, no usage)	0	1184	814	1244	0	376
	Logic (low latency, no usage)	0	1301	942	1371	0	365
Fixed to float	Int64 input		596	431	618	0	398
Float to fixed	Int64 result		540	375	577	0	446
Float to float	Double to single		202	95	202	0	464
Compare	Programmable		197	119	153	0	463
Divide	C_RATE=1		4467	3437	6166	0	360
	C_RATE=55		606	444	550	0	303
Square Root	C_RATE=1		2338	1997	3423	0	335
	C_RATE=54		418	297	512	0	330
Reciprocal	DSP48E1 (full usage)	14	544	378	650	0	438
Reciprocal Square Root	DSP48E1 (full usage)	75	2318	1869	2784	1	402
Absolute Value	N/A	0	0	0	0	0	464
Natural Logarithm	DSP48E1 (full usage)	61	2690	2061	3484	0	376
	DSP48E1 (medium usage)	23	3610	3129	3803	0	335
	Logic	0	5885	5533	5649	0	319
Multiply Flow Control: Blocking, Optimize Goal: Performance	DSP48E1 (max usage)	11	783	479	984	0	402

Table 2-24: **Characterization of Double-Precision Format on Zynq-7000 Devices *(Cont'd)***

Operation	Resources[1]						Maximum Frequency (MHz)[2][3]
	Embedded		FPGA Logic			18k Block RAMs	Zynq-7000
	Type	Number	LUT-FF Pairs	LUTs	FFs		-1 Speed Grade
Add/Subtract Flow Control: Blocking, Optimize Goal: Performance	DSP48E1 (speed optimized, full usage)	3	1341	906	1485	0	360

Notes:

1. The device used for these figures is an XC7Z045-1.
2. Area and maximum clock frequencies are provided as a guide and might vary with new releases of the Xilinx implementation tools.
3. Maximum clock frequencies are shown in MHz. Clock frequency does not take jitter into account and should be de-rated by an amount appropriate to the clock source jitter specification.
4. The absolute value operator uses no logic nor registers so will not become the critical path in any realistic circuit.

Port Descriptions

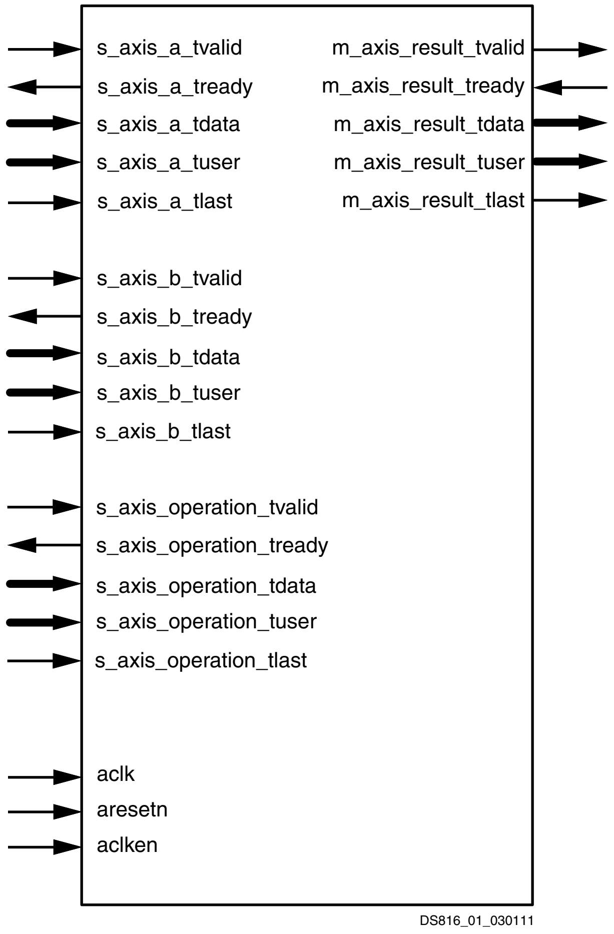

Figure 2-1: **Core Schematic Symbol**

The ports employed by the core are shown in Figure 2-1. They are described in more detail in Table 2-25. All control signals are active-High with the exception of aresetn.

Table 2-25: **Core Signal Pinout**

Name	Direction	Optional	Description
aclk	Input	yes	Rising-edge clock
aclken	Input	yes	Active-High clock enable (optional)
aresetn	Input	yes	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	no	TVALID for channel A
s_axis_a_tready	Output	yes	TREADY for channel A

Table 2-25: **Core Signal Pinout *(Cont'd)***

Name	Direction	Optional	Description
s_axis_a_tdata	Input	no	TDATA for channel A. See TDATA Packing for internal structure
s_axis_a_tuser	Input	yes	TUSER for channel A
s_axis_a_tlast	Input	yes	TLAST for channel A
s_axis_b_tvalid	Input	no	TVALID for channel B
s_axis_b_tready	Output	yes	TREADY for channel B
s_axis_b_tdata	Input	no	TDATA for channel B. See TDATA Packing for internal structure
s_axis_b_tuser	Input	yes	TUSER for channel B
s_axis_b_tlast	Input	yes	TLAST for channel B
s_axis_operation_tvalid	Input	no	TVALID for channel OPERATION
s_axis_operation_tready	Output	yes	TREADY for channel OPERATION
s_axis_operation_tdata	Input	no	TDATA for channel OPERATION. See TDATA Packing for internal structure
s_axis_operation_tuser	Input	yes	TUSER for channel OPERATION
s_axis_operation_tlast	Input	yes	TLAST for channel OPERATION
m_axis_result_tvalid	Output	no	TVALID for channel RESULT
m_axis_result_tready	Input	yes	TREADY for channel RESULT
m_axis_result_tdata	Output	no	TDATA for channel RESULT. See TDATA Subfield for internal structure
m_axis_result_tuser	Output	yes	TUSER for channel RESULT
m_axis_result_tlast	Output	yes	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.

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 that 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.

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 2-26.

Table 2-26: **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

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 2-26, 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 2-27.

Table 2-27: **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 v6.1, but instead are provided in the RESULT channel `m_axis_result_tuser` subfield. For more details, see Output Result Channel.

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. The output is set to a correctly signed ∞.

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 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. Natural 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 2-28.)

Table 2-28: **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 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.

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 $1 \le b_0.b_1b_2...b_{p-1} < 2$ 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.

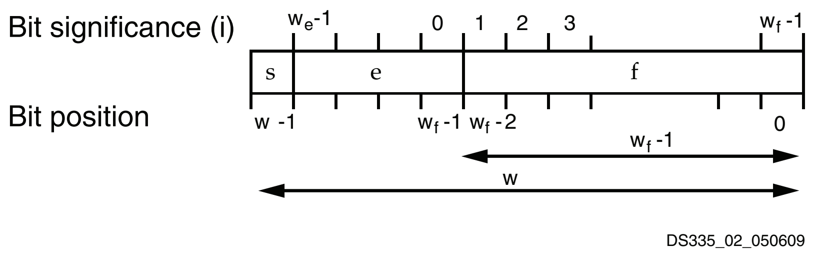

Figure 3-1: **Bit Fields within the Floating-Point Representation**

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, $e = 11\ldots11$)	Any non-zero field. For results that are NaN the most significant bit of fraction is set (that is, $f = 10\ldots00$)
$\pm\infty$	sign of ∞	$2^{w_e - 1}$ -1 (that is, $e = 11\ldots11$)	Zero (that is, $f = 00\ldots00$)

Table 3-1: **Special Values**

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 two's 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 two's complement number).

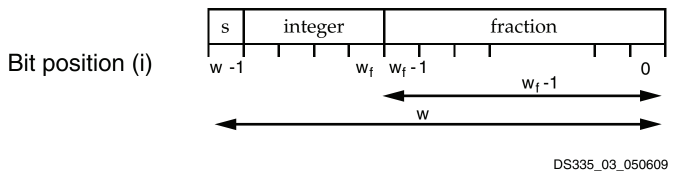

Figure 3-2: **Bit Fields within the Fixed-Point Representation**

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:

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

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

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.

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 6] and [Ref 7].

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.

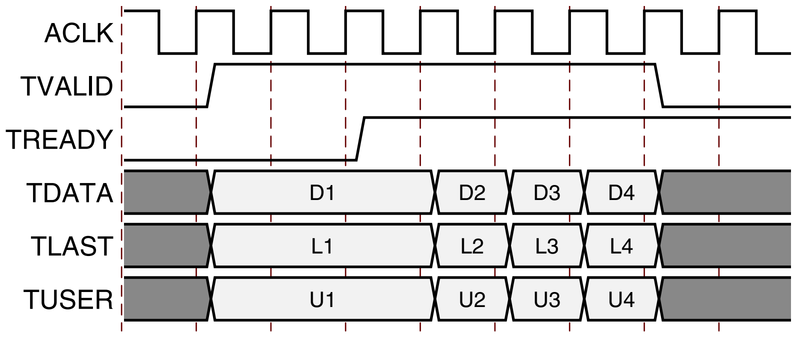

Figure 3-3: **Data Transfer in an AXI4-Stream Channel**

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 GUI 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. This mode is recommended for users wishing to move to this version from a pre-AXI4-Stream core version 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.

Warning: 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.

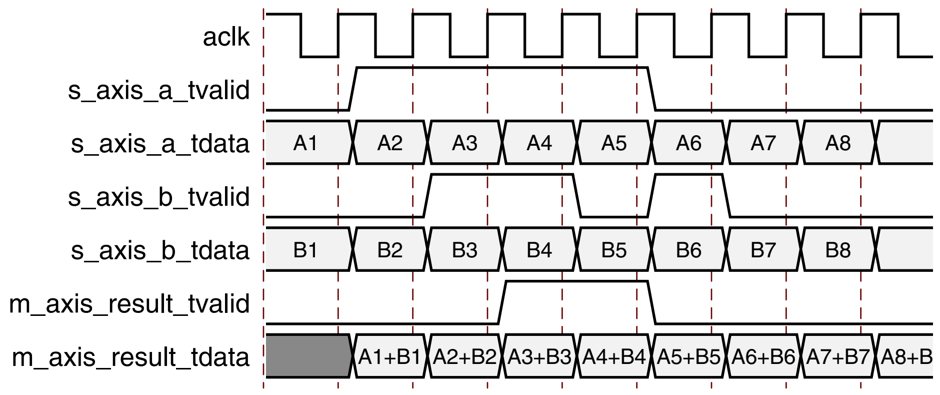

Figure 3-4: **Non-Blocking Mode**

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's 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.

Also note that the core buffers have a greater capacity than implied by the diagram.

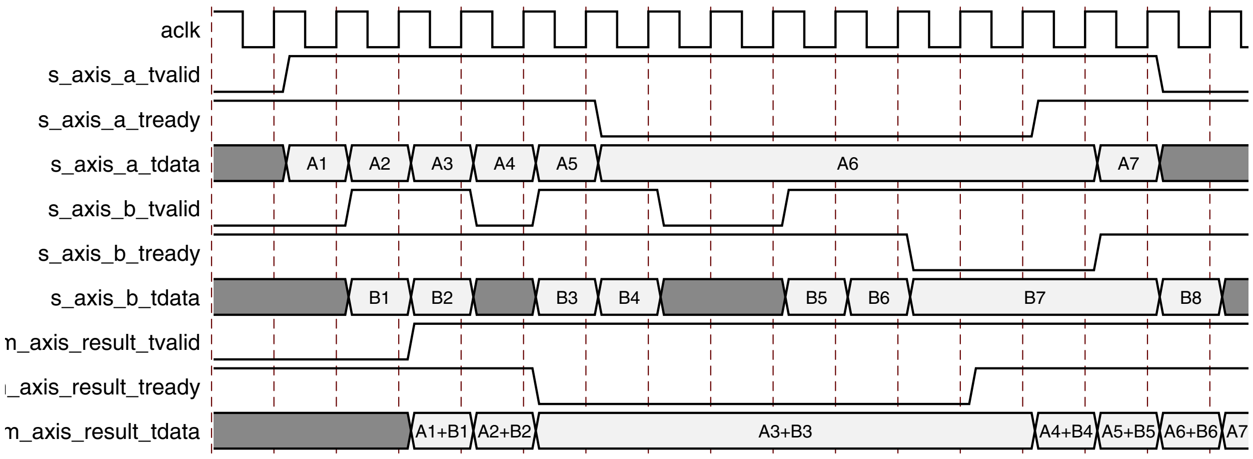

Figure 3-5: **Blocking Mode**

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 and B Input Channels

TDATA Structure for A and B Channels

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

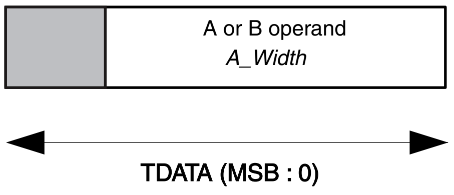

Figure 3-6: **TDATA Structure for A and B Channels**

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

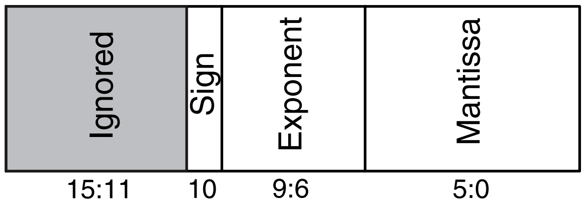

Figure 3-7: **Custom Precision Input (11 bits) Mapped to TDATA Channel**

TDATA Structure for OPERATION Channel

The OPERATION channel exists only when add and subtract operations are selected together, of when a programmable comparator is selected. The binary encoded operation code, as specified in Table 2-26, are 6 bits in length. However, due to the byte-oriented nature of TDATA, this means that TDATA has a width of 8 bits.

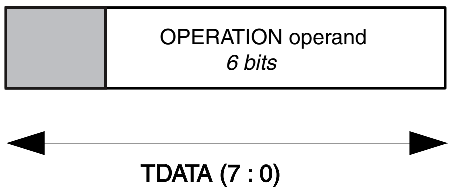

Figure 3-8: **TDATA Structure for OPERATION Channels**

TLAST and TUSER Handling

TLAST in AXI4-Stream is used to denote the last transfer of a block of data. TUSER is for ancillary information which qualifies or augments the primary data in TDATA. The Floating-Point Operator core operates on a per-sample basis where each operation is independent of any before or after. Because of this, there is no need for TLAST on a Floating-Point Operator core, nor is there any need for TUSER. The TLAST and TUSER signals are 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 or ancillary field, but which is not relevant to the core operation. The facility to pass TLAST and/or TUSER removes the burden of matching latency to the TDATA path, which can be variable, through the Floating-Point Operator core.

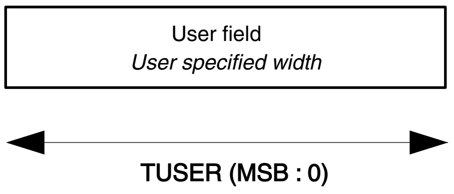

Figure 3-9: **TUSER Structure for A, B and OPERATION Channels**

TLAST Options

TLAST for each input channel is optional. Each, when present, can be passed through the Floating-Point Operator core, or, when more than one channel has TLAST enabled, 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.

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, then 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 (underflow, etc.) 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 down to 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 (add, multiply, etc.) 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) 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.

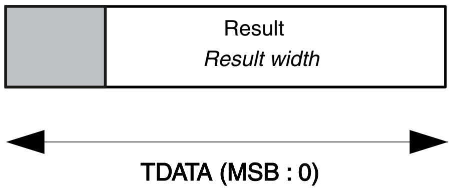

Figure 3-10: **TDATA Structure for Numerical Result Channel**

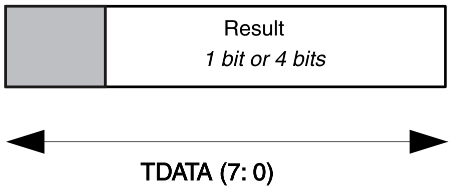

Figure 3-11: **TDATA Structure for Comparator Result Channel**

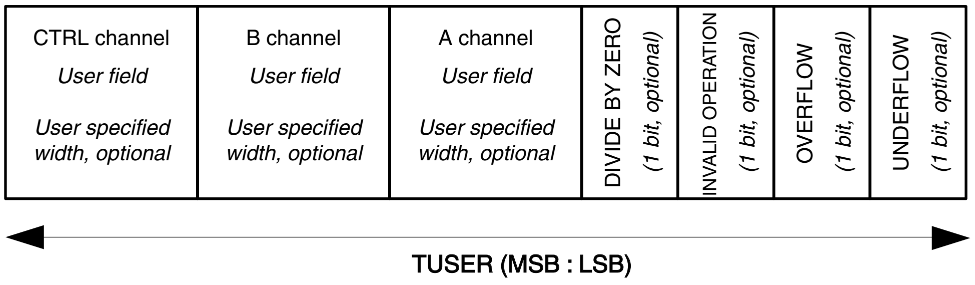

Figure 3-12: **TUSER Structure for Result Channel**

C Model Reference

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 v6.1 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 (see the Vivado documentation to set up the C model output).

The C model is an output of the CORE Generator™ software, listed under Output Product Selection. Ensure that "C Simulation Model" is selected and then generate the core. The C model is generated in `<component_name>/cmodel/` as a zip file for each supported platform.

Features

- Bit accurate with Floating-Point Operator core
- Available for 32-bit and 64-bit Linux platforms
- Available for 32-bit and 64-bit 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 v6.1 bit accurate C model for 32-bit and 64-bit Linux, and 32-bit and 64-bit 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's 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 4] and Multiple Precision Floating-point Reliable (MPFR) [Ref 3] shared libraries, header files and source code
- The C model header file
- The example code showing customers how to call the C model
- Documentation

Note: The C model uses MPIR and MPFR libraries, which are provided in the ZIP files. 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.

Table 4-1: **Example C Model ZIP File Contents - Linux**

File	Description
floating_point_v6_1_bitacc_cmodel.h	Header file which defines the C model API
libIp_floating_point_v6_1_bitacc_cmodel.so	Model shared object library
libgmp.so.7	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
README.txt	Release notes
pg060-floating-point.pdf	This product guide
mpir-2.2.1.tar.bz2	MPIR source code
mpfr-3.0.1.tar.bz2	MPFR source code

Table 4-2: **Example C Model ZIP File Contents - Windows**

File	Description
floating_point_v6_1_bitacc_cmodel.h	Header file which defines the C model API
libIp_floating_point_v6_1_bitacc_cmodel.dll	Model dynamically linked library
libIp_floating_point_v6_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
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
README.txt	Release notes
pg060-floating-point.pdf	This product guide
mpir-2.2.1.tar.bz2	MPIR source code
mpfr-3.0.1.tar.bz2	MPFR source code
mpfr.build.vc9.zip	Microsoft Visual Studio 2008 project files for compiling MPFR on Windows
mpfr.build.vc10.zip	Microsoft Visual Studio 2010 project files for compiling MPFR on Windows
mpfr_nt_stdint.h	Header file to enable some MPFR functions when compiling MPFR on Windows

Installation

Linux

- Unpack the contents of the ZIP file.
- Ensure that the directory where the `libIp_floating_point_v6_1_bitacc_cmodel.so`, `libgmp.so.7` 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_v6_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 4] and MPFR [Ref 3] are free, open source software libraries, distributed under the GNU Lesser General Public License. The source code and 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_v6_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 4-3.

Table 4-3: **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)

Table 4-3: **Floating-Point Operator C Model Data Types *(Cont'd)***

Name	Type	Description
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 v6.1 core (for example, add with different precision operands)

`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 the user 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 4] and MPFR [Ref 3].

`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.

Functions

There are several C model functions accessible to the user.

Information Functions

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

Table 4-4: **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 v6.1 this is "6.1".

Initialization Functions

The Floating-Point Operator C model initialization functions are shown in Table 4-5. Most functions have variants to handle floating-point and fixed-point variables.

Table 4-5: **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*.

Table 4-5: **Floating-Point Operator C Model Initialization Functions *(Cont'd)***

Name	Return	Arguments	Description
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.
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:

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

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 4-6. 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 4-6: **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	

Table 4-6: **Floating-Point Operator C Model Assignment Functions *(Cont'd)***

Name	Return	Arguments	Description
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	
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 significand 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 variable's precision.

Conversion functions

The Floating-Point Operator C model conversion functions are shown in Table 4-7. 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 4-7: **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	

Table 4-7: **Floating-Point Operator C Model Conversion Functions *(Cont'd)***

Name	Return	Arguments	Description
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.
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 lower case letters are used, with a = 10, b = 11, ... z = 35. For *base* in the range 37 to 62, digits, upper case, and lower case 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 lower case letters are used, with a = 10, b = 11, ... z = 35. For base in the range 37 to 62, digits, upper case, and lower case 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 4-8. 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 4-8: **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_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.

Table 4-8: **Floating-Point Operator C Model Operation Functions *(Cont'd)***

Name	Return	Arguments	Description
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.
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.

Table 4-8: **Floating-Point Operator C Model Operation Functions *(Cont'd)***

Name	Return	Arguments	Description
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_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.
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.

Table 4-8: **Floating-Point Operator C Model Operation Functions *(Cont'd)***

Name	Return	Arguments	Description
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.
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 4-9 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 4-9 for the condition code encoding. Single precision version.

Table 4-8: **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 4-9 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_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).

1. Only supported for `xip_fpo_t` operands with 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 v6.1 core, and the returned exceptions are guaranteed to match exactly the signaled exceptions of the Floating-Point Operator v6.1 core, for identical inputs.

When the operand and result variables do not meet constraints of the Floating-Point Operator v6.1 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 4-9. Encodings not shown are reserved and are not returned by the functions.

Table 4-9: **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_v6_1_bitacc_cmodel.h` header file and the header files of the MPIR [Ref 4] and MPFR [Ref 3] 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_v6_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_v6_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_v6_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. Pre-compiled MPIR and MPFR libraries are provided with the C model. 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.7`
- `libmpfr.so.4`
- `libIp_floating_point_v6_1_bitacc_cmodel.so`

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

```
-L. -lgmp -lmpfr -lIp_floating_point_v6_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 run_bitacc_cmodel.c -o run_bitacc_cmodel -I. -L. -lgmp -lmpfr
-lIp_floating_point_v6_1_bitacc_cmodel
```

Windows

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

- `libgmp.dll`
- `libmpfr.dll`
- `libIp_floating_point_v6_1_bitacc_cmodel.dll`

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

- `libgmp.lib`
- `libmpfr.lib`
- `libIp_floating_point_v6_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. Pre-compiled MPIR and MPFR libraries are provided with the C model, using the following versions of the libraries:

- MPIR 2.2.1
- MPFR 3.0.1

As 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 4] and MPFR [Ref 3] web sites. Microsoft Visual Studio project files for compiling MPFR on Windows can be obtained from Brian Gladman's website [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_v6_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;
    }
```

SECTION II: VIVADO DESIGN SUITE

Customizing and Generating the Core

This chapter includes information on using Xilinx tools to customize and generate the core in the Vivado™ Design Suite.

GUI

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

The GUI allows configuration of the following:

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

Main Configuration Screen

The main configuration screen allows the following parameters to be specified:

- Component Name
- Operation Selection

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

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

- Add/Subtract
- Multiply
- Divide
- Square-root
- Compare
- Reciprocal
- Reciprocal square root
- Absolute value
- Natural logarithm
- Fixed-to-float
- Float-to-fixed
- Float-to-float

When *Add/Subtract* 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 2-26).

When *Add/Subtract* or *Multiply* is selected, the level of XtremeDSP 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 2-26). 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.

Second and Third Configuration Screens

Depending on the configuration you select from the first screen, the second and third configuration screens let you specify the precision of the operand and result.

Precision of the Operand and Results

This parameter defines 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 5-1 defines the general limits of the format widths.

Table 5-1: **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 GUI:

- 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 5-2.

Table 5-2: **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

Penultimate Configuration Screen

The penultimate configuration screen lets you specify the following:

- Architecture Optimizations
- Family Optimizations

Architecture Optimizations

For double precision multiplication and addition/subtraction 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.

Family Optimizations

- Multiplier Usage allows the level of XtremeDSP slice multiplier use to be specified.

Multiplier Usage

The level and type of multiplier usage depends upon the operation. Table 5-3 summarizes these options for multiplication.

Table 5-3: **Impact of Multiplier Usage on the Implementation of the Multiplier**

Multiplier Usage	
No usage	Logic
Medium usage	DSP48E1+logic in multiplier body
Full usage	DSP48E1 used in multiplier body
Max usage	DSP48E1 multiplier body and rounder

Table 5-4 summarizes these options for addition/subtraction.

Table 5-4: **Impact of Precision, and Multiplier Usage on the Implementation of the Adder/ Subtractor**

Multiplier Usage (only valid values listed)	Other	Single	Double
No usage	Logic	Logic	Logic
Full usage	Not supported	2 DSP48E1	3 DSP48E1

Final Configuration Screen

The final configuration screen lets you specify:

- Flow Control Options
- Latency and Rate Configuration
- Control Signals
- Optional Output Fields
- AXI4-Stream Channel Options

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. The maximum latency of the Floating-Point Operator core is tabulated for a range of width and operation types in Tables 2-1 through 2-14.

Cycles per Operation

The 'Cycles per Operation' GUI 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 and DIVIDE_BY_ZERO.
- 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.
- 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.

Using the Floating-Point Operator IP Core

The Vivado Customize IP dialog box performs error-checking on all input parameters. Resource estimation and optimum 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 documentation.

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
- Mult
- CMult (Constant Multiplier)
- Divide
- Reciprocal
- SquareRoot
- Reciprocal SquareRoot
- Absolute
- Logarithm
- Relational (provides compare operations)
- Convert (provides fixed to float, float to fixed, float to float)

See the System Generator for DSP User Guide for more information.

Parameter Values in the XCI File

Table 5-5 defines valid entries for the XCI parameters. Parameters are case sensitive. Default values are displayed in bold. Xilinx strongly recommends that XCI parameters not be manually edited in the XCI file; instead, use Vivado software GUI to configure the core and perform range and parameter value checking.

Table 5-5: **XCI Parameters**

XCI Parameter	XCI Values
Component_Name	Name must begin with a letter and be composed of the following characters: a to z, A to Z, 0 to 9 and "_".
Operation_Type	**Add_Subtract**, Multiply, Divide, Square_Root, Compare, Reciprocal Rec_Square_Root Absolute, Logarithm, Fixed_to_float, Float_to_fixed, Float_to_float
Add_Sub_Value	**Both**, Add, Subtract
C_Compare_Operation	**Programmable**, Unordered, Less_Than, Equal, Less_Than_Or_Equal, Greater_Than, Not_Equal, Greater_Than_Or_Equal, Condition_Code
A_Precision_Type	**Single**, Double, Int32, Custom
C_A_Exponent_Width	Integer with range summarized in Table 5-1 and Table 5-2. Required when A_Precision_Type is Custom.
C_A_Fraction_Width	Integer with range summarized in Table 5-1 and Table 5-2. Required when A_Precision_Type is Custom.
Result_Precision_Type	**Single**, Double, Int32, Custom.
C_Result_Exponent_Width	Integer with range summarized in Table 5-1 and Table 5-2. Required when Result_Precision_Type is Custom.
C_Result_Fraction_Width	Integer with range summarized in Table 5-1 and Table 5-2. Required when Result_Precision_Type is Custom.
C_Optimization	**Speed_Optimized**, Low_Latency
C_Mult_Usage	**No_Usage**, Medium_Usage, Full_Usage, Max_Usage
Maximum_Latency	False, **True**
C_Latency	Integer with range 0 to the maximum latency of core as summarized by Tables 2-1 through 2-14 (default is **maximum latency**). Required when Maximum_Latency is False.

Table 5-5: **XCI Parameters *(Cont'd)***

XCI Parameter	XCI Values
C_Rate	Integer with range 1 to maximum rate as described in Cycles per Operation (default is **1**).
Has_ARESETn	**False**, True
Has_ACLKEN	**False**, True
C_Has_UNDERFLOW	**False**, True
C_Has_OVERFLOW	**False**, True
C_Has_INVALID_OP	**False**, True
C_Has_DIVIDE_BY_ZERO	**False**, True
Flow_Control	**Blocking**, NonBlocking
Axi_Optimize_Goal	**Resources**, Performance
Has_RESULT_TREADY	**True**, False
Has_A_TLAST	**False**, True
Has_A_TUSER	**False**, True
A_TUSER_Width	Integer with range 1 to 256. Default is **1**.
Has_B_TLAST	**False**, True
Has_B_TUSER	**False**, True
B_TUSER_Width	Integer with range 1 to 256. Default is **1**.
Has_OPERATION_TLAST	**False**, True
Has_OPERATION_TUSER	**False**, True
OPERATION_TUSER_Width	Integer with range 1 to 256. Default is **1**.
RESULT_TLAST_Behv	**Null**, Pass_A_TLAST, Pass_B_TLAST, Pass_OPERATION_TLAST, OR_All_TLASTs, AND_all_TLASTs

Output Generation

The output of generation consists of some or all of the following:

Table 5-6: **Output Files**

Name	Description
<component_name>.xci	Input file containing the parameters used to customize the core.
<component_name>.veo	Template files containing code that can be used as a model for instantiation of the customized core.
<component_name>.vho <component_name>.vhd	VHDL model of the core.
<component_name>.v	Structural Verilog model of the core.
/doc/pg060-floating-point.pdf /doc/floating_point_v6_1_vinfo.html	Core documents
<component_name>_readme.txt	Readme file for the core.

Detailed Example Design

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. See the Vivado documentation for more information on delivery of the demonstration test bench

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's signals in your simulator's waveform viewer to see the operations of the test bench.

The Demonstration Test Bench in Detail

The demonstration test bench performs the following tasks:

- Instantiates the core
- Generates an input data frame consisting of one or the sum of two complex sinusoids
- Generates a clock signal
- Drives the core's input signals to demonstrate core features
- Checks that the core's output signals 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 implementing and simulating your core, see the Vivado Design Suite documentation.

Chapter 7

Constraining the Core

There are no constraints associated with this core.

SECTION III: ISE DESIGN SUITE

Customizing and Generating the Core

This chapter includes information on using Xilinx tools to customize and generate the core in the ISE® Design Suite.

GUI

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

The GUI allows configuration of the following:

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

Main Configuration Screen

The main configuration screen allows the following parameters to be specified:

- Component Name
- Operation Selection

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

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

- Add/Subtract
- Multiply
- Divide
- Square-root
- Compare
- Reciprocal
- Reciprocal square root
- Absolute value
- Natural logarithm
- Fixed-to-float
- Float-to-fixed
- Float-to-float

When *Add/Subtract* 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 2-26).

When *Add/Subtract* or *Multiply* is selected, the level of XtremeDSP 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 2-26). 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.

The *Natural logarithm* operator is not supported for Spartan-6 devices.

Second and Third Configuration Screens

Depending on the configuration you select from the first screen, the second and third configuration screens let you specify the precision of the operand and result.

Precision of the Operand and Results

This parameter defines 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 8-1 defines the general limits of the format widths.

Table 8-1: **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 GUI:

- 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 $[\log_2(\text{Fraction Width}+3)] + 1$

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 $[\log_2(\text{Total Width}+3)] + 1$

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 8-2.

Table 8-2: **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

Penultimate Configuration Screen

The penultimate configuration screen lets you specify the following:

- Architecture Optimizations
- Family Optimizations

Architecture Optimizations

On Virtex®-6 and 7 series FPGAs, for double precision multiplication and addition/subtraction 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.

Family Optimizations

- Multiplier Usage allows the level of XtremeDSP slice multiplier use to be specified.

Multiplier Usage

The level and type of multiplier usage depend upon the operation and FPGA family. Table 8-3 summarizes these options for multiplication.

Table 8-3: **Impact of Family and Multiplier Usage on the Implementation of the Multiplier**

Multiplier Usage	Spartan-6 FPGA Family	Virtex-6 and 7 Series FPGA Families
No usage	Logic	Logic
Medium usage	DSP48A1+logic(1) in multiplier body	DSP48E1+logic(1) in multiplier body
Full usage	DSP48A1 used in multiplier body	DSP48E1 used in multiplier body
Max usage	DSP48A1 multiplier body and rounder	DSP48E1 multiplier body and rounder

1. Logic-assisted multiplier variant is available only for single and double precision formats in Virtex-6 and 7 Series FPGAs and single precision in Spartan-6 FPGAs.

Table 8-4 summarizes these options for addition/subtraction.

Table 8-4: **Impact of Family, Precision, and Multiplier Usage on the Implementation of the Adder/Subtractor**

Multiplier Usage (only valid values listed)	Spartan-6 FPGA Family	Virtex-6 and 7 Series FPGA Families		
	Any	Other	Single	Double
No usage	Logic	Logic	Logic	Logic
Full usage	Not supported	Not supported	2 DSP48E1	3 DSP48E1

Final Configuration Screen

The final configuration screen lets you specify:

- Flow Control Options
- Latency and Rate Configuration
- Control Signals
- Optional Output Fields
- AXI4-Stream Channel Options

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. The maximum latency of the Floating-Point Operator core is tabulated for a range of width and operation types in Tables 2-1 through 2-14.

Cycles per Operation

The 'Cycles per Operation' GUI 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 and DIVIDE_BY_ZERO.
- 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.
- 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.

Using the Floating-Point Operator IP Core

The CORE Generator™ GUI performs error-checking on all input parameters. Resource estimation and optimum 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 CORE Generator software documentation.

Simulation Models

The core has two options for simulation models:

- VHDL RTL-based simulation model in XilinxCoreLib
- Verilog UNISIM-based structural simulation model

The models required can be selected in the CORE Generator project options.

Xilinx recommends that simulations utilizing UNISIM-based structural models be run using a resolution of 1 ps. Some Xilinx library components require a 1 ps resolution to work properly in either functional or timing simulation. The UNISIM-based structural simulation models can produce incorrect results if simulated with a resolution other than 1 ps. See the "Register Transfer Level (RTL) Simulation Using Xilinx Libraries" section in Chapter 6 of the *Synthesis and Simulation Design Guide* [Ref 8].

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
- Mult
- CMult (Constant Multiplier)
- Divide
- Reciprocal
- SquareRoot
- Reciprocal SquareRoot
- Absolute
- Logarithm
- Relational (provides compare operations)
- Convert (provides fixed to float, float to fixed, float to float)

See the System Generator for DSP User Guide for more information.

Parameter Values in the XCO File

Table 8-5 defines valid entries for the XCO parameters. Parameters are not case sensitive. Default values are displayed in bold. Xilinx strongly recommends that XCO parameters not be manually edited in the XCO file; instead, use CORE Generator software GUI to configure the core and perform range and parameter value checking.

Table 8-5: **XCO Parameters**

XCO Parameter	XCO Values
Component_Name	Name must begin with a letter and be composed of the following characters: a to z, A to Z, 0 to 9 and "_".
Operation_Type	**Add_Subtract**, Multiply, Divide, Square_Root, Compare, Reciprocal Rec_Square_Root Absolute, Logarithm, Fixed_to_float, Float_to_fixed, Float_to_float
Add_Sub_Value	**Both**, Add, Subtract
C_Compare_Operation	**Programmable**, Unordered, Less_Than, Equal, Less_Than_Or_Equal, Greater_Than, Not_Equal, Greater_Than_Or_Equal, Condition_Code
A_Precision_Type	**Single**, Double, Int32, Custom
C_A_Exponent_Width	Integer with range summarized in Table 8-1 and Table 8-2. Required when A_Precision_Type is Custom.
C_A_Fraction_Width	Integer with range summarized in Table 8-1 and Table 8-2. Required when A_Precision_Type is Custom.
Result_Precision_Type	**Single**, Double, Int32, Custom.
C_Result_Exponent_Width	Integer with range summarized in Table 8-1 and Table 8-2. Required when Result_Precision_Type is Custom.
C_Result_Fraction_Width	Integer with range summarized in Table 8-1 and Table 8-2. Required when Result_Precision_Type is Custom.
C_Optimization	**Speed_Optimized**, Low_Latency
C_Mult_Usage	**No_Usage**, Medium_Usage, Full_Usage, Max_Usage
Maximum_Latency	False, **True**
C_Latency	Integer with range 0 to the maximum latency of core as summarized by Tables 2-1 through 2-14 (default is **maximum latency**). Required when Maximum_Latency is False.

Table 8-5: **XCO Parameters *(Cont'd)***

XCO Parameter	XCO Values
C_Rate	Integer with range 1 to maximum rate as described in Cycles per Operation (default is **1**).
Has_ARESETn	**False**, True
Has_ACLKEN	**False**, True
C_Has_UNDERFLOW	**False**, True
C_Has_OVERFLOW	**False**, True
C_Has_INVALID_OP	**False**, True
C_Has_DIVIDE_BY_ZERO	**False**, True
Flow_Control	**Blocking**, NonBlocking
Axi_Optimize_Goal	**Resources**, Performance
Has_RESULT_TREADY	**True**, False
Has_A_TLAST	**False**, True
Has_A_TUSER	**False**, True
A_TUSER_Width	Integer with range 1 to 256. Default is **1**.
Has_B_TLAST	**False**, True
Has_B_TUSER	**False**, True
B_TUSER_Width	Integer with range 1 to 256. Default is **1**.
Has_OPERATION_TLAST	**False**, True
Has_OPERATION_TUSER	**False**, True
OPERATION_TUSER_Width	Integer with range 1 to 256. Default is **1**.
RESULT_TLAST_Behv	**Null**, Pass_A_TLAST, Pass_B_TLAST, Pass_OPERATION_TLAST, OR_All_TLASTs, AND_all_TLASTs

Output Generation

Table 8-6: **Output Files**

Name	Description
<component_name>.xco	CORE Generator input file containing the parameters used to generate a core.
<component_name>.ngc	Binary Xilinx implementation netlist files containing the information required to implement the module in a Xilinx (R) FPGA.
<component_name>.vho <component_name>.veo	Template files containing code that can be used as a model for instantiating.
<component_name>.vhd	VHDL behavioral model

Table 8-6: **Output Files *(Cont'd)***

Name	Description
<component_name>.v	Structural simulation model
/doc/pg060-floating-point.pdf /doc/floating_point_v6_1_vinfo.html	Core documents
<component_name>.asy	Graphical symbol information file. Used by the ISE tools and some third party tools to create a symbol representing the core.
<component_name>_xmdf.tcl	ISE® Project Navigator interface file. ISE uses this file to determine how the files output by CORE Generator for the core can be integrated into your ISE project.
<component_name>.gise <component_name>.xise	ISE Project Navigator support files. These are generated files and should not be edited directly.
<component_name>_readme.txt	Readme file for the IP.
<component_name>_flist.txt	Text file listing all of the output files produced when a customized core was generated in the CORE Generator.

Detailed Example Design

There is no example design for this core.

Demonstration Test Bench

When the core is generated using CORE Generator™ in the ISE® 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 CORE Generator output directory. The source code is comprehensively commented.

Using the Demonstration Test Bench

The demonstration test bench instantiates the generated Floating-Point Operator core. If the CORE Generator project options were set to generate a structural model, a VHDL or Verilog netlist named `<component_name>.vhd` or `<component_name>.v` was generated. If this file is not present, generate it using the netgen program, for example:

```
netgen -sim -ofmt vhdl <component_name>.ngc <component_name>.vhd
```

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's signals in your simulator's waveform viewer to see the operations of the test bench.

The Demonstration Test Bench in Detail

The demonstration test bench performs the following tasks:

- Instantiates the core
- Generates an input data frame consisting of one or the sum of two complex sinusoids
- Generates a clock signal
- Drives the core's input signals to demonstrate core features

- Checks that the core's output signals 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.

Chapter 10

Constraining the Core

There are no constraints associated with this core.

SECTION IV: APPENDICES

Migrating

In the ISE® Design Suite, the CORE Generator™ core upgrade functionality can be used to update an existing XCO file from versions 4.0, 5.0 and 6.0 to Floating-Point Operator, v6.1, but it should be noted that for v4.0 and v5.0 the upgrade mechanism alone does not create a core compatible with v6.1. See Instructions for Minimum Change Migration. Floating-Point Operator v6.1 has parameters additional to v4.0 and v5.0 for AXI4-Stream support. Floating Point Operator v6.1 is backwards compatible with v6.0 both in terms of parameters and ports. Figure A-1 shows the changes to XCO parameters from versions 4.0 and 5.0 to version 6.1. For clarity, XCO parameters with no changes are not shown.

See also UG911, *Vivado Design Suite Migration Methodology Guide* for information on migrating to the Vivado™ Design Suite.

Parameter Changes in the XCO File

Table A-1: **XCO Parameter Changes from v4.0 and v5.0 to v6.1**

Version 4.0 and 5.0	Version 6.1	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

Table A-1: **XCO Parameter Changes from v4.0 and v5.0 to v6.1 *(Cont'd)***

Version 4.0 and 5.0	Version 6.1	Notes
	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

For more information on this upgrade feature, see the CORE Generator software documentation.

Port Changes

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

Table A-2: **Port Changes from v4.0 and v5.0 to v6.1**

Versions 4.0 and 5.0	Version 6.1	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	

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

Versions 4.0 and 5.0	Version 6.1	Notes
	s_axis_a_tvalid	TVALID (AXI4-Stream channel handshake signal) for each channel
	s_axis_b_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_operation_tready	
	m_axis_result_tready	
	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_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_operation_tuser(G-1:0)	
	m_axis_result_tuser(H-1:0)	

Functionality Changes

Latency Changes

There is no change in latency from Floating Point Operator v6.0 to v6.1. The latency of Floating-Point Operator v6.1 is different compared to v4.0 and v5.0 in general. The update process cannot account for this and guarantee equivalent performance.

Importantly, when in Blocking Mode, the latency of the core is variable, so only the minimum possible latency can be determined.

When in Non-Blocking Mode, the latency of the core for equivalent performance is the same as that for the equivalent configuration of v4.0 and v5.0.

Special Considerations when Migrating to AXI

Instructions for Minimum Change Migration

To configure the Floating-Point Operator v6.1 to most closely mimic the behavior of previous versions the translation is as follows:

Parameters

Set Flow Control to NonBlocking and uncheck all AXI4-Stream channel options (TUSER and TLAST).

Ports

Rename and map signals as detailed in Port Changes. Tie all TVALID signals on input channels (A, B, OPERATION) to 1.

Remember to account for `aresetn` being active-Low, and the requirement to assert `aresetn` for at least two clock cycles to reset the core.

Performance

The fully-pipelined latency of the v6.1 core with a Non-Blocking interface configuration is the same as the v4.0 and v5.0 cores.

Debugging

See Solution Centers in Appendix C for information helpful to the debugging progress.

Additional Resources

Xilinx Resources

For support resources such as Answers, Documentation, Downloads, and Forums, see the Xilinx Support website at:

www.xilinx.com/support.

For a glossary of technical terms used in Xilinx documentation, see:

www.xilinx.com/company/terms.htm.

Solution Centers

See the Xilinx Solution Centers for support on devices, software tools, and intellectual property at all stages of the design cycle. Topics include design assistance, advisories, and troubleshooting tips.

References

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 Floating-Point Reliable (MPFR) Library www.mpfr.org
4. The GNU Multiple Precision Integers and Rationals (MPIR) library www.mpir.org
5. Multiple Precision Arithmetic on Windows, Brian Gladman: http://gladman.plushost.co.uk/oldsite/computing/gmp4win.php
6. *Xilinx AXI Reference Guide* (UG761)
7. AMBA 4 AXI4-Stream Protocol Version 1.0 Specification
8. Synthesis and Simulation Design Guide (UG626)
9. Vivado documentation website

Technical Support

Xilinx provides technical support at www.xilinx.com/support for this LogiCORE™ IP product when used as described in the product documentation. Xilinx cannot guarantee timing, functionality, or support of product if implemented in devices that are not defined in the documentation, if customized beyond that allowed in the product documentation, or if changes are made to any section of the design labeled DO NOT MODIFY.

See the IP Release Notes Guide (XTP025) for more information on this core. For each core, there is a master Answer Record that contains the Release Notes and Known Issues list for the core being used. The following information is listed for each version of the core:

- New Features
- Resolved Issues
- Known Issues

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
07/25/12	1.0	Initial Xilinx release. This Product Guide is derived from DS816 and UG812.

Notice of Disclaimer