LogiCORE IP Complex Multiplier v6.0

Product Guide for Vivado Design Suite

PG104 December 18, 2013

XILINX®

XILINX

Table of Contents

Introduction

The Xilinx LogiCORE™ IP Complex Multiplier implements AXI4-Stream compliant, high-performance, optimized complex multipliers based on user-specified options.

The two multiplicand inputs and optional rounding bit are input on independent AXI4-Stream channels as slave interfaces and the resulting product output on an AXI4-Stream master interface.

Within each channel, operands and the results are represented in signed two's complement format. The operand widths and the result width are parameterizable.

Features

- Drop-in module for 7 series FPGAs
- AXI4-Stream-compliant interfaces
- 8-bit to 63-bit input precision and up to 127-bit output precision
- Supports truncation or unbiased rounding
- Configurable minimum latency
- Three or four real multiplier implementation options
- Option to use LUTs or DSP Slices

LogiCORE IP Facts Table	
Core Specifics	
Supported Device Family(1)	UltraScale™ Architecture, Zynq®-7000, 7 Series
Supported User Interfaces	AXI4-Stream
Resources	Not Available
Provided with Core	
Design Files	Encrypted RTL
Example Design	Not Provided
Test Bench	VHDL
Constraints File	Not Provided
Simulation Model	Encrypted VHDL
Supported S/W Driver	N/A
Tested Design Flows(2)	
Design Entry	Vivado® Design Suite IP Integrator System Generator for DSP
Simulation	For supported simulators, see the Xilinx Design Tools: Release Notes Guide.
Synthesis	Vivado Synthesis
Support	
Provided by Xilinx @ www.xilinx.com/support	

Notes:

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

Overview

The Complex Multiplier IP core performs complex multiplication of two operands in Cartesian form. The result is also in Cartesian form.

Feature Summary

The Complex Multiplier IP core provides a complex multiplication solution for two complex operands where each operand can be from 8 to 63 bits wide. The real and imaginary components of each operand must be the same width as each other, but the widths of the two operands are individually configured. Options are provided to bias the implementation to the needs of the application. For instance, latency is configurable, implementation can use DSP Slices or LUTs and the algorithm can use a 3 or 4 multiplier solution which trades latency for resource. AXI4-Stream interfaces are provided, but if the full traffic management capabilities of AXI4-Stream interfaces are not required, the interfaces can be configured for no additional resource.

Applications

Complex Multipliers are common in many DSP applications, including signal mixing and in Fast Fourier Transforms.

Licensing and Ordering Information

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

For more information, visit the Complex Multiplier product web page.

Information about other Xilinx LogiCORE IP modules is available at the Xilinx Intellectual Property page. For information on pricing and availability of other Xilinx LogiCORE IP modules and tools, contact your local Xilinx sales representative.

Product Specification

There are two basic architectures to implement complex multiplication, given two operands: $a = a_r + ja_i$ and $b = b_r + jb_i$, yielding an output $p = ab = p_r + jp_i$.

Direct implementation requires four real multiplications:

$$p_r = a_r b_r - a_i b_i \quad \text{Equation 2-2}$$

$$p_i = a_r b_i + a_i b_r \quad \text{Equation 2-3}$$

By exploiting that

$$p_r = a_r b_r - a_i b_i = a_r(b_r + b_i) - (a_r + a_i)b_i \quad \text{Equation 2-4}$$

$$p_i = a_r b_i + a_i b_r = a_r(b_r + b_i) + (a_i - a_r)b_r \quad \text{Equation 2-5}$$

a three real multiplier solution can be devised, which trades off one multiplier for three pre-combining adders and increased multiplier word length.

Performance

Latency

Latency is configurable. For the performance tables, latency was set to automatic, which results in a fully pipelined circuit.

Throughput

The complex multiplier supports full throughput in all configurations, that is, one output per cycle.

Resource Utilization

There are no resource figures for this version of the core.

Port Descriptions

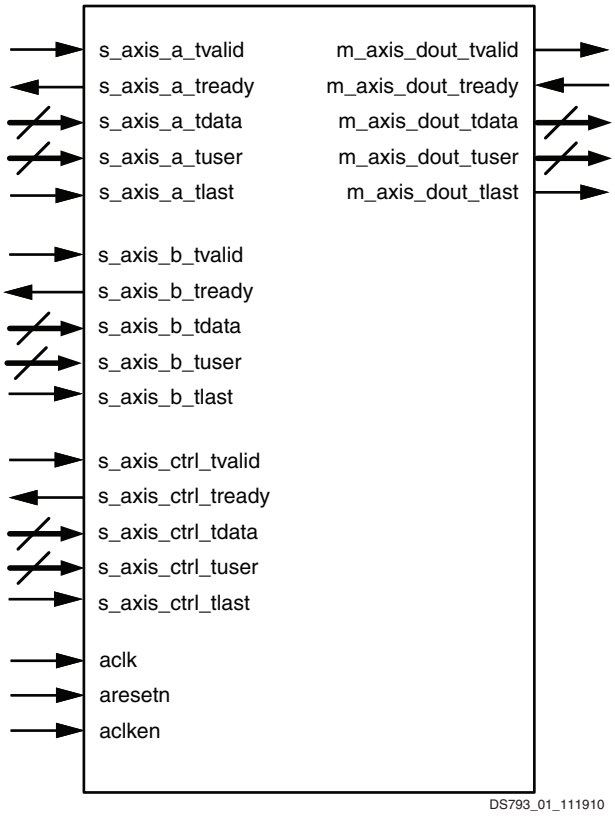

Figure 2-1: **Core Schematic Symbol**

Table 2-1 describes the Complex Multiplier core ports as shown in Figure 2-1.

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

Name	Direction	Optional	Description
aclk	Input	yes	Rising-edge clock. The aclk signal is optional. It is not present when FlowControl is NonBlocking and MinimumLatency = 0.
aclken	Input	yes	Active-High clock enable (optional)

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

Name	Direction	Optional	Description
aresetn	Input	yes	Active-Low synchronous clear (optional, always take priority over aclken) **Note**: aresetn should be asserted or deasserted for not less than two aclk cycles.
s_axis_a_tvalid	Input	no	TVALID for channel A
s_axis_a_tready	Output	yes	TREADY for channel A
s_axis_a_tuser[A-1:0]	Input	yes	TUSER for channel A. Width selectable from 1 to 256 bits
s_axis_a_tdata[B-1:0]	Input	no	TDATA for channel A. See TDATA Packing for internal structure and width.
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_tuser[C-1:0]	Input	yes	TUSER for channel B. Width selectable from 1 to 256 bits
s_axis_b_tdata[D-1:0]	Input	no	TDATA for channel B. See TDATA Packing for internal structure and width.
s_axis_b_tlast	Input	yes	TLAST for channel B.
s_axis_ctrl_tvalid	Input	yes	TVALID for channel CTRL
s_axis_ctrl_tready	Output	yes	TREADY for channel CTRL
s_axis_ctrl_tuser[E-1:0]	Input	yes	TUSER for channel CTRL. Width selectable from 1 to 256 bits
s_axis_ctrl_tdata[7:0]	Input	yes	TDATA for channel CTRL. See TDATA Packing for internal structure and width.
s_axis_ctrl_tlast	Input	yes	TLAST for channel CTRL.
m_axis_dout_tvalid	Output	no	TVALID for channel DOUT
m_axis_dout_tready	Input	yes	TREADY for channel DOUT
m_axis_dout_tuser[G-1:0]	Output	yes	TUSER for channel DOUT. Width is the sum of the enabled TUSER fields on input channels.
m_axis_dout_tdata[H-1:0]	Output	no	TDATA for channel DOUT. See TDATA Packing internal structure.
m_axis_dout_tlast	Output	yes	TLAST for channel DOUT.

Notes:

1. 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.
2. Width constants A to H are arbitrary variables, determined by GUI or configuration parameters.

Designing with the Core

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

Hardware Implementation

Three Real Multiplier Solution

The three real multiplier implementation takes advantage of the pre-adder in the DSP Slice, saving general fabric resources. In general, the three multiplier solution uses more slice resources (LUTs/flipflops) and have a lower maximum achievable clock frequency than the four multiplier solution.

Four Real Multiplier Solution

The four real multiplier solution makes maximum use of DSP Slice resources, and has higher clock frequency performance than the three real multiplier solution, in many cases reaching the maximum clock frequency of the FPGA.

It still consumes slice resources for pipeline balancing, but this slice cost is always less than that required by the equivalent three real multiplier solution.

LUT-based Solution

The core offers the option to build the complex multiplier using LUTs only. While this option uses a significant number of slices, achieves a lower maximum clock frequency and uses more power than DSP Slice implementations, it might be suitable for applications where DSP Slices are in limited supply, or where lower clock rates are in use.

The three real multiplier configuration is used exclusively when LUT implementation is selected.

Rounding

In a DSP system, especially if the system contains feedback, the word length growth through the multiplier should be offset by quantizing the results. Quantization, or reduction in word length, results in error, introduces quantization noise, and can introduce bias. For best results, it is favorable to select a quantization method that introduces zero mean noise and minimizes noise variance. Figure 3-1 illustrates the quantization method used for truncation.

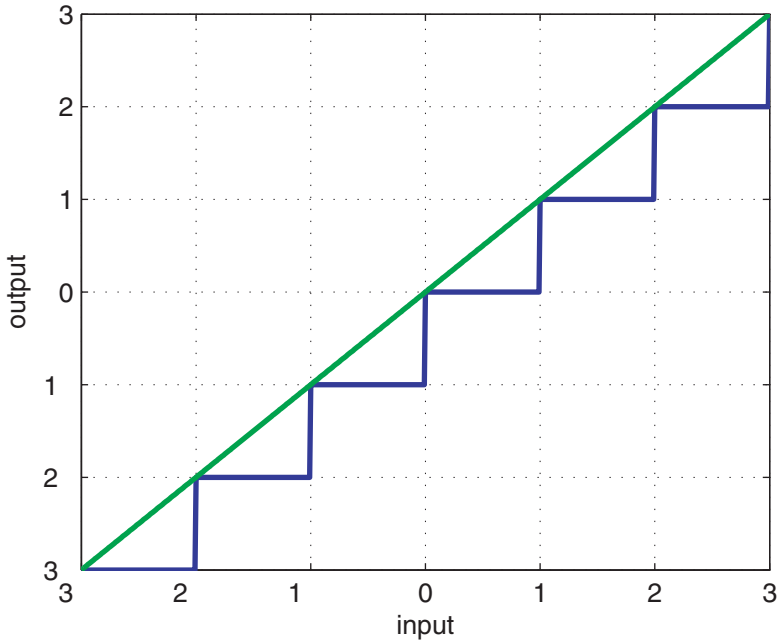

Figure 3-1: **Truncation**

For truncation the probability density function (PDF) of the noise is:

$$p(e) = \begin{cases} \frac{1}{\Delta} & -\Delta < e < 0 \\ 0 & otherwise \end{cases} \quad \text{Equation 3-1}$$

therefore the mean and the variance of the error introduced are:

$$m_e = \int_{-\Delta}^{0} ep(e)de = \frac{1}{\Delta}\int_{-\Delta}^{0} ede = -\frac{\Delta}{2} \quad \text{Equation 3-2}$$

$$\sigma_e^2 = \int_{0}^{\Delta} e^2 p(e)de = \frac{1}{\Delta}\int_{0}^{\Delta} e^2 de = \frac{\Delta^2}{3} \quad \text{Equation 3-3}$$

Implementing truncation has no cost in hardware; the fractional bits are simply trimmed.

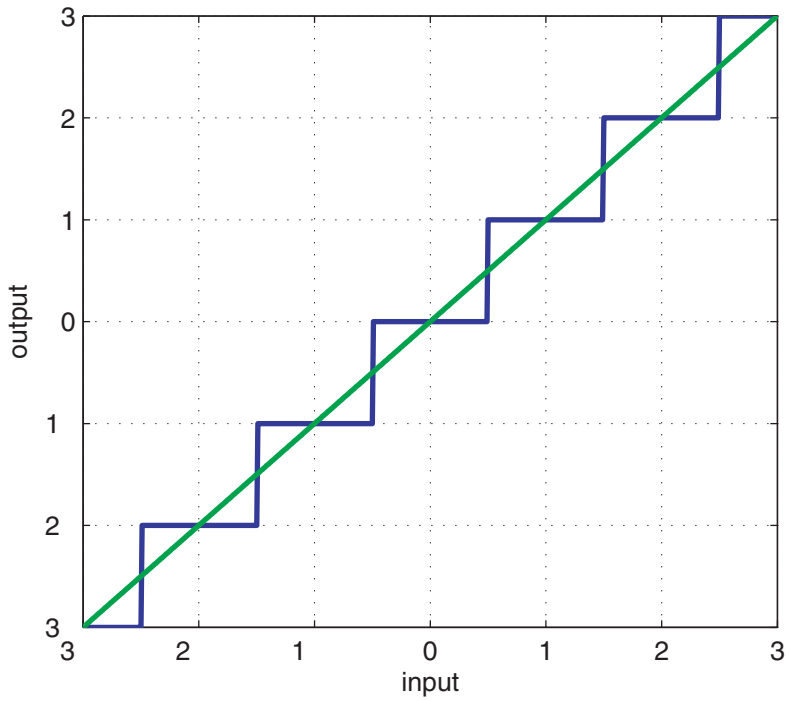

Figure 3-2: **Rounding**

For rounding the PDF of the noise is:

$$p(e) = \begin{cases} \frac{1}{\Delta} & -\Delta/2 < e < \Delta/2 \\ 0 & otherwise \end{cases}$$ *Equation 3-4*

the mean and the variance of the error introduced are:

$$m_e = \int_{-\Delta/2}^{\Delta/2} ep(e)de = \frac{1}{\Delta}\int_{-\Delta/2}^{\Delta/2} ede = 0$$ *Equation 3-5*

$$\sigma_e^2 = \int_{-\Delta/2}^{\Delta/2} e^2 p(e)de = \frac{1}{\Delta}\int_{-\Delta/2}^{\Delta/2} e^2 de = \frac{\Delta^2}{12}$$ *Equation 3-6*

Therefore, the ideal rounder introduces no DC bias to the signal flow. If the full product word (for example, $a_r b_r$ - $a_i b_i$) is represented with B_P bits, and the actual result of the core (for example, p_r) is represented with B_R bits, then bits B_P-1...B_P-B_R are the integer part, and B_P-B_R-1..0 are the fractional part of the result.

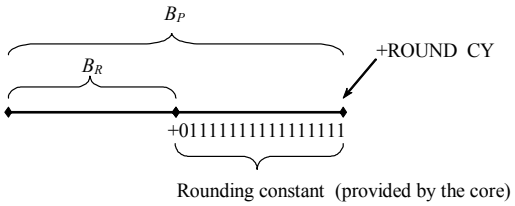

Figure 3-3: **Rounding to B_r Bits from B_p Bits**

To implement the rounding function shown in Figure 3-2, 0.5 (represented in $B_P B_P$-B_R format) has to be added to the full product word, then the lower B_P-B_R bits need to be truncated. However, if the fractional part is exactly 0.5, this method always rounds up, which introduces positive bias to the computation. Also, if the rounding constant is -1 (Figure 3-3), 0.5 would be always rounded down, introducing negative bias.

If 0.5 is rounded using a static rule, the resulting quantization always introduces bias. To avoid bias, rounding must be randomized. Therefore, the core adds a rounding constant, and an extra 1 should be added with ½ probability, thus dithering the exact rounding threshold. Typical round carry sources being used extensively as control signals are listed in Table 3-1.

Table 3-1: **Unbiased Rounding Sources**

0.5 Rounding Rule	Round Carry Source
Round towards 0	-MSB(P)
Round towards +/- infinity	MSB(P)
Round towards nearest even	LSB(P)

Rounding of the results is not trivial when multiple, cascaded DSP Slices are involved in the process, such as evaluation of Equation 3-5 or Equation 3-6. The sign of the output (MSB_o) cannot be predicted from the operands before the actual multiplications and additions take place, and would incur additional latency or resource to implement outside the DSP Slices. Therefore an external signal should be used to feed the round carry input, through the *ROUND_CY* (Bit 0 of `s_axis_ctrl_tdata`) pin.

A good candidate for a source can be a clock-dividing flip-flop, or any 50% duty cycle random signal, which is not correlated with the fractional part of the results. For predictable behavior (as for bit-true modeling) the *ROUND_CY* signal might need to be connected to a CLK independent source in your design, such as an LSB of one of the complex multiplier inputs.

Nevertheless, even when a static rule is used (such as tying *ROUND_CY* = 0), bias and quantization error are reduced compared to using truncation.

In many cases, for DSP Slice implementation, the addition of the rounding constant is 'free', as the C port and carry-in input can be utilized. In devices without DSP Slices, the addition of rounding typically requires an extra slice-based adder and an additional cycle of latency.

Clocking

The core uses a single clock, `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 the Complex Multiplier are conveyed using AXI4-Stream channels. A channel consists of TVALID and TDATA always, plus several optional ports and fields. In the Complex Multiplier, 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 Complex Multiplier operates on the operands contained in the TDATA fields and outputs the result in the TDATA field of the output channel. The Complex Multiplier does not use TUSER and TLAST as such, but the core provides the facility to convey these fields with the same latency as for TDATA. This facility is intended to ease use of the Complex Multiplier in a system. For example, the complex multiplier can be used as a mixer or phase shift operating on streaming packetized data. In this example, the core could be configured pass the TLAST of the packetized data channel saving the system designer the effort of constructing a bypass path for this information.

For further details on AXI4-Stream interfaces see the *Xilinx AXI Reference Guide* (UG761) [Ref 1] and the *AMBA 4 AXI4-Stream Protocol Version: 1.0 Specification* [Ref 2].

Basic Handshake

Figure 3-4 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.

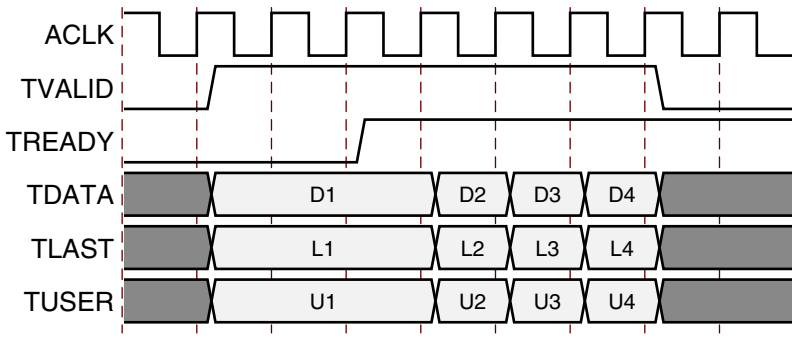

Figure 3-4: **Data Transfer in an AXI-Stream Channel**

NonBlocking Mode

The term NonBlocking 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 NonBlocking behavior is selected using the FlowControl parameter or GUI field. The Complex Multiplier core supports a NonBlocking mode in which the AXI4-Stream channels do not have TREADY, that is, they do not support back pressure. The choice of Blocking or NonBlocking 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. 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-AXI 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 v3.1. Figure 3-5 shows the NonBlocking behavior for a case with latency of one cycle.

IMPORTANT: For performance, ARESETn is registered internally, which delays its action by a clock cycle. The effect of this is that any transaction input in the cycle following the deassertion of ARESETn is reset by the action of ARESETn, resulting in an output data value of zero. TVALID is also inactive on the output channel for this cycle.

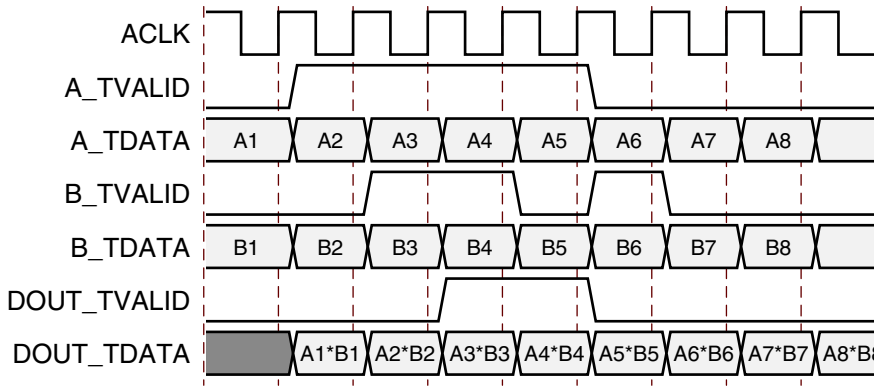

Figure 3-5: **NonBlocking Mode**

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. Blocking or NonBlocking behavior is selected using the FlowControl parameter or GUI field. 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 Complex Multiplier has 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 fills 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 backpressure.

The three 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 3-6 shows both Blocking behavior and back-pressure. 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 DOUT_TREADY. This is back-pressure' Sustained back-pressure on the output along with data availability on the inputs eventually lead to a saturation of the core buffers, leading the core to signal that it can no longer accept further input by deasserting the input channel TREADY signals. The minimum latency in this example is two 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.

IMPORTANT: The core buffers have a greater capacity than implied by Figure 3-6.

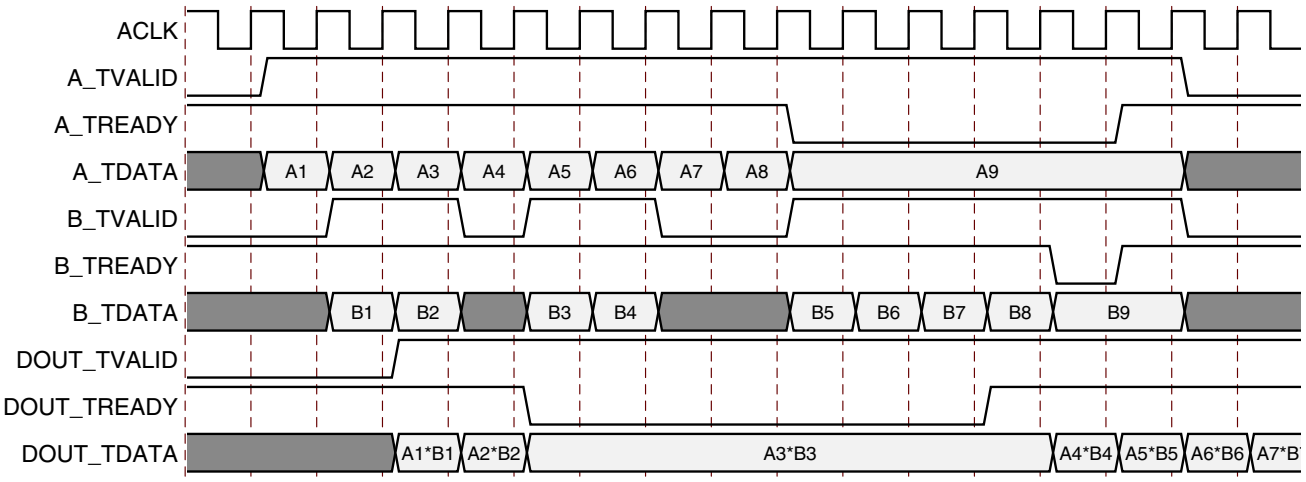

Figure 3-6: **Blocking Mode**

Note: Figure 3-6 illustrates Blocking behavior and handshake protocol. Latencies implied by the diagram might not be accurate.

TDATA Packing

Fields within an AXI4-Stream interface follow a specific naming nomenclature. See Figure 3-7. Normally, information pertinent to the application, complex multiplication in this case, is carried in the TDATA field. In this core the complex operand components, real and imaginary, are both passed to or from the core through the channel TDATA port, with the real component in the least significant position. 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 complex multiplier is configured to have an A operand width of 11 bits. Each of the real and imaginary components of A are 11 bits wide. The real component would occupy bits 10 down to 0. Bits 15 down to 11 would be ignored. Bits 26 down to 16 would hold the imaginary component and bits 31 down to 27 would likewise be ignored. For the output DOUT channel, result fields are sign extended to the byte boundary. The bits added by byte orientation are ignored by the core and do not result in additional resource use.

TDATA Structure for A, B and DOUT Channels

Input ports A, B and output port DOUT carry complex data in their TDATA field. For each, the real component occupies the least significant bits. The imaginary component occupies a bit field which starts on the next byte-boundary above the real component as shown in the previous section. See Figure 3-7.

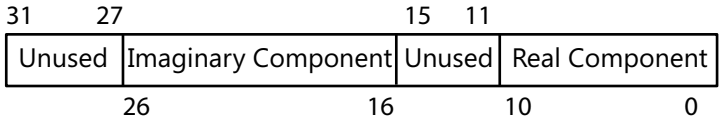

Figure 3-7: **TDATA Structure for A, B and DOUT Channels**

TDATA Structure for CTRL Channel

The CTRL channel exists only when Rounding has been selected and exists to convey only the rounding bit (referred to in this document as ROUND_CY). This bit occupies bit 0 of TDATA for this channel. However, due to the byte-oriented nature of TDATA, this means that TDATA has a width of 8 bits. ROUND_CY is added to the rounding constant of 0.01111..., making the rounding constant either 0.01111.. or 0.100... Therefore, setting this bit to 0 causes the rounding to be rounded to negative infinity; setting it to 1 causes it to be rounded to positive infinity and setting a new random value for each sample gives unbiased random rounding. See Rounding for more information.

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 complex multiplier 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 complex multiplier, 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 complex multiplier does indeed have some packetization or ancillary field, but which is not relevant to the complex multiplier. The facility to pass TLAST and/or TUSER removes the burden of matching latency to the TDATA path, which can be variable, through the complex multiplier.

TLAST Options

TLAST for each input channel is optional. Each, when present, can be passed through the complex multiplier, 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 CTRL.

Examples:

If channels A and CTRL both have TUSER with widths of 5 and 8 bits respectively, the output TUSER is a suitably delayed concatenation of A and CTRL TUSER fields, 13 bits wide, with A in the least significant 5 bit positions (4 down to 0).

If B and CTRL have TUSER widths of 4 and 10 respectively, but A has no TUSER, DOUT TUSER (`m_axis_dout_tuser`) has the bits of B_TUSER (`s_axis_b_tuser`) suitably delayed in positions 3 down to 0 with CTRL_TUSER (`s_axis_ctrl_tuser`) bits, suitably delayed, in positions 13 down to 4.

Customizing and Generating the Core

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

If you are customizing and generating the core in the Vivado IP Integrator, see the *Vivado Design Suite User Guide: Designing IP Subsystems using IP Integrator* (UG994) [Ref 3] for detailed information. IP Integrator might auto-compute certain configuration values when validating or generating the design. To check whether the values do change, see the description of the parameter in this chapter. To view the parameter value you can run the `validate_bd_design` command in the tcl console.

Vivado Integrated Design Environment

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

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

For details, see the *Vivado Design Suite User Guide: Designing with IP* (UG896) [Ref 5] and the "Working with the Vivado IDE" section in the *Vivado Design Suite User Guide: Getting Started* (UG910) [Ref 6].

The Complex Multiplier core in the Vivado Design Environment (IDE) has several fields to set parameter values for the particular instantiation required. The following provides a description of each field.

Component Name: The name of the core component to be instantiated. The name must begin with a letter and be composed of the following characters: a to z, A to Z, 0 to 9 and '_'.

Input and Implementation Tab

Channel A Options:

- **AR/AI Operand Width**: Select the first operand width. The width applies to both the real and imaginary components of the complex operand. See TDATA Structure for A, B and DOUT Channels to see how the operand components map into TDATA for this channel. This parameter is automatically updated in IP Integrator.
- **Has TLAST:** Select whether the channel has TLAST. To ease system design, the core passes any TLAST and TUSER to the output with latency equal to the TDATA field. See TLAST and TUSER Handling
- **Has TUSER:** Select whether the channel has TUSER. To ease system design, the core passes any TLAST and TUSER to the output with latency equal to the TDATA field. See TLAST and TUSER Handling.
- **TUSER Width**: Select the width, in bits, of the TUSER field for this channel. This parameter is automatically updated in IP Integrator.

Channel B Options:

- **BR/BI Operand Width**: Select the second operand width. The width applies to both the real and imaginary components of the complex operand. See TDATA Structure for A, B and DOUT Channels to see how the operand components map into TDATA for this channel. This parameter is automatically updated in IP Integrator.
- **Has TLAST:** Select whether the channel has TLAST. To ease system design, the core passes any TLAST and TUSER to the output with latency equal to the TDATA field. See TLAST and TUSER Handling
- **Has TUSER:** Select whether the channel has TUSER. To ease system design, the core passes any TLAST and TUSER to the output with latency equal to the TDATA field. See TLAST and TUSER Handling
- **TUSER Width**: Select the width, in bits, of the TUSER field for this channel. This parameter is automatically updated in IP Integrator.

Multiplier Construction Options: Allows the choice of using LUTs (slice logic) to construct the complex multiplier, or using DSP Slices.

Optimization Goal: Selects between Resource and Performance optimization.

- This selection affects both the internal architectural decisions and the performance/resource trade-offs in the AXI4-Stream interfaces.
- For Multiplier-based implementations, Resource optimization generally uses the three real multiplier structure. The core uses the four real multiplier structure when the three real multiplier structure uses more multiplier resources. Performance

optimization always uses the four real multiplier structure to allow the best clock frequency performance to be achieved.

Flow Control Options: Selects between Blocking and NonBlocking behavior for the AXI4-Stream interfaces. See NonBlocking Mode and Blocking Mode for greater detail.

Configuration and Output Tab

Output Product Range

- **Output Width**: Selects the width of the output product real and imaginary components. The values are automatically initialized to provide the full-precision product when the A and B operand widths are set. The natural width of a complex multiplication is the sum of the input widths plus one. If Output Width is set to be less than this natural width, the least significant bits are truncated or rounded, as selected by the next GUI field.

Output Rounding

If the full-precision product (output width equals natural width) is selected, no rounding options are available. Otherwise either Truncation or Random Rounding can be selected. When Random Rounding is selected, the CTRL channel is enabled. Bit 0 of the TDATA field of this channel determines the particular type of rounding for the operation in question. See Rounding for further details.

Channel CTRL Options

The control channel exists to supply the bit which determines the rounding type. However, it also provides an opportunity to pass TUSER or TLAST information which has no association with either of the input operands through the core.

- **Has TLAST:** Select whether the channel has TLAST. To ease system design, the core passes any TLAST and TUSER to the output with latency equal to the TDATA field. See TLAST and TUSER Handling.
- **Has TUSER:** Select whether the channel has TUSER. To ease system design, the core passes any TLAST and TUSER to the output with latency equal to the TDATA field. See TLAST and TUSER Handling.
- **TUSER Width**: Select the width, in bits, of the TUSER field for this channel. This parameter is automatically updated in IP Integrator.

Output TLAST Behavior

- **TLAST Behavior:** Determines which of the input channels' TLAST or which combination of input channel TLASTs is conveyed to the output channel TLAST. Available options are to pass any one of the input channels' TLAST or to pass the

logical OR of all available input TLASTs or to pass the logical AND of all available input TLASTs. See TLAST and TUSER Handling.

Core Latency

Select the desired latency for the core.

- **Latency Configuration:** Selects between **Automatic** and **Manual**. When **Automatic**, latency is set such that the core is fully pipelined for maximum performance. **Manual** allows user-selectable minimum latency. Performance drops when the value set is less than the fully pipelined latency. When the value set is larger than fully pipelined, the core delays the output using an SRL. With **Blocking Flow Control** selected, the core latency is not fixed, so only minimum latency can be specified.
- **Minimum Latency:** The value for Manual Latency Configuration.

Control Signals

Selects which control signals should be present on the core. These options are disabled when the core has a minimum latency of zero.

  - **ACLKEN:** Enables the clock enable (`aclken`) pin on the core. All registers in the core are enabled by this signal.
  - **ARESETn:** Enables the active-Low synchronous clear (`aresetn`) pin on the core. All registers in the core are reset by this signal. This can increase resource use and degrade performance, as the number of SRL-based shift registers that can be used is reduced. `aresetn` always take priority over `aclken`.

Implementation Details Tab: Click the Implementation Details tab to see an estimate of the DSP Slice resources used for a particular complex multiplier configuration. This value updates instantaneously with changes in the GUI, allowing trade-offs in implementation to be evaluated immediately.

System Generator for DSP Graphical User Interface

This section describes each tab of the System Generator for DSP GUI and details the parameters that differ from the Vivado IDE. The Complex Multiplier core can be found in the Xilinx Blockset in the Math section. The block is called 'Complex Multiplier 6.0.' See the System Generator for DSP Help page for the 'Complex Multiplier 6.0' block for more information on parameters not mentioned here. For more information about System Generator see the *System Generator for DSP User Guide* (UG640) [Ref 10].

Page 1

Page 1 is used to specify the complex multiplier construction, optimization options and output width in a similar way to the Vivado IDE.

Page 2

Page 2 is used to specify latency, output product rounding options and block control signals. As with page 1, all controls have the same effect as controls in the Vivado IDE.

Implementation

This page is used only for System Generator for DSP FPGA area estimation and has no equivalent parameters on the GUI.

Output Generation

For details, see "Generating IP Output Products" in the *Vivado Design Suite User Guide: Designing with IP* (UG896) [Ref 5].

Constraining the Core

There are no constraints for this core.

Simulation

For comprehensive information about Vivado simulation components, as well as information about using supported third party tools, see the *Vivado Design Suite User Guide: Logic Simulation* (UG900) [Ref 7].

Synthesis and Implementation

For details about synthesis and implementation, see the *Vivado Design Suite User Guide: Designing with IP* (UG896) [Ref 5].

C Model Reference

The Complex Multiplier 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 Complex Multiplier core, and it is suitable for inclusion in a larger framework for system-level simulation or core-specific verification.

Features

- Bit accurate with Complex Multiplier core
- Available for 32-bit and 64-bit Linux platforms
- Available for 32-bit and 64-bit Windows platforms
- Supports all features of the Complex Multiplier core with the exception of those affecting timing or AXI4-Stream configuration
- Designed for integration into a larger system model
- Example C code showing how to use the C model functions

Overview

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

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

Unpacking and Model Contents

There are separate ZIP files containing all the files necessary for use. Each ZIP file contains:

- C model shared library
- C model header file
- Example code showing how to call the C model

Table 8-1 and Table 8-2 list the contents of each ZIP file.

Table 8-1: **C Model ZIP File Contents: Linux**

File	Description
cmpy_v6_0_bitacc_cmodel.h	Header file which defined the C model API
libIp_cmpy_v6_0_bitacc_cmodel.so	Model shared object library
run_bitacc_cmodel.c	Example program for calling the C model.
gmp.h	MPIR header file, used by the C model
libgmp.so.11	MPIR library, used by the C model

Table 8-2: **C Model ZIP File Contents: Windows**

File	Description
cmpy_v6_0_bitacc_cmodel.h	Header file which defined the C model API
libIp_cmpy_v6_0_bitacc_cmodel.dll	Model dynamically linked library
libIp_cmpy_v6_0_bitacc_cmodel.lib	Model LIB file for compiling
run_bitacc_cmodel.c	Example program for calling the C model
gmp.h	MPIR header file, used by the C model
libgmp.dll	MPIR library, used by the C model
libgmp.lib	MPIR .lib file for compiling

Installation

Linux

- Unpack the contents of the ZIP file.
- Ensure that the directory where the `libIP_cmpy_v6_0_bitacc_cmodel.so` resides 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_cmpy_v6_0_bitacc_cmodel.dll` resides is:
  - Included in the path of the environment variable PATH, or
  - In the directory in which the executable that calls the C model is run.

C Model Interface

An example file, `run_bitacc_cmodel.c`, is included. This demonstrates how to call the C model. See this file for examples of using the interface described in this section.

The Application Programming Interface (API) of the C model is defined in the header file `cmpy_v6_0_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 Complex Multiplier C model are listed in Table 8-3.

Table 8-3: **C Model Data Types**

Name	Type	Description
xip_real	Double	Base data type for small width configurations(1)
xip_complex	Struct {re,im} xip_real	Base element of operands for small width configurations(1)
xip_uint	Unsigned int	Used for configuration parameter of integer or Boolean type. For Boolean: 0=false 1=true
xip_mpz	Mpz_t(2)	Base data type for larger width configurations(1)
xip_mpz_complex	Struct {re, im} xip_mpz	Base element of operand for larger width configurations(1)
xip_array_complex	Struct	Structure to hold data for input to the complex multiplier (A or B operands) or output for small configurations(1)
xip_array_mpz_complex	Struct	Structure to hold data for input to the complex multiplier (A or B operands) or output for larger configurations.

Table 8-3: **C Model Data Types *(Cont'd)***

Name	Type	Description
xip_array_uint	Struct	Structure to hold data for input to the complex multiplier (ROUND operand).
xip_cmpy_v6_0_status	Int	Error code return from many C model functions. 0 indicates success. Any other value indicates failure.
xip_status	Int	Same as xip_cmpy_v6_0_status but used for functions which are not core-specific.
xip_real	Double	
xip_cmpy_v6_0_config	Struct	The configuration of the core itself. The members of this structure are listed in the cmpy_v6_0_bitacc_cmodel file. The names closely match the same names in XCI files. The cmpy_v6_0_bitacc_cmodel file also contains #defined values for all.
xip_cmpy_v6_0	Struct	Type defined which C (not C++) can use as a handle (pointer) to a C++ object – the C model itself.

Notes:

1. Small configurations are where the natural width (APortWidth+BPortWidth+1) does not exceed 53 bits.
2. From gmp.h library

The xip_array_complex, xip_array_mpz_complex and xip_array_uint data types are structures with the following members:

- **data**: A pointer to the array of data values.
- **data_size**: Of type size_t, which describes the total size of the data array.
- **data_capacity**: Also of type size_t, which describes how much of the array is currently
- populated.
- **dim**: A pointer to a size_t array of values which indicate the size of each dimension.
- **dim_size** (size_t): Indicates the number of dimensions of the data array.
- **dim_capacity**: Indicates how much of the dimension array is currently populated.
- **owner**: This unsigned int member is provided as a handle for when the data structure is intended to be passed from one core to another, but is not used by any of the Complex Multiplier C model functions.

Functions

There are several accessible C model functions.

Information Functions

Table 8-4 lists the information functions. The prototypes for these functions can be found in the C model header file.

Table 8-4: **Information Functions**

Name	Return	Arguments	Description
xip_cmpy_get_version	Const char*	Void	Return the Complex Multiplier C model version as a null terminated string. For v6.0, this is '6.0'.
xip_cmpy_v6_0_get_default_config	xip_cmpy_v6_0_status	xip_cmpy_v6_0_config*	Populates the contents of structure pointed to by the input argument with the values of a default configuration.

Initialization Functions

The functions to create, configure and destroy the C model and associated data structures are listed in Table 8-5.

Table 8-5: **Initialization Functions**

Name	Return	Arguments	Description
xip_cmpy_v6_0_create	Pointer to structure holding configuration of C model object	Pointer to structure holding configuration	Creates new C model object and returns pointer to config structure (which is pointer to C model itself).
xip_cmpy_v6_0_destroy	xip_cmpy_v6_0_status	Pointer to xip_cmpy_v6_0 (C model itself)	Deallocates memory owned by C model and destroys C model itself.
xip_cmpy_v6_0_get_config	xip_cmpy_v6_0_status	Pointer to C model, pointer to configuration structure	Copies the contents of the configuration of the C model indicated to the designated configuration structure.
xip_array_#TYPE#_create(1)	Pointer to created data structure	None	Allocates memory for the structure itself, not the array members within it.
xip_array_#TYPE#_reserve_data(1)	xip_status	Pointer to data structure, maximum number of elements in data array.	(Re)allocates enough memory for the maximum size. Error is returned if the data_capacity of the structure is greater than space allocated.

Table 8-5: **Initialization Functions *(Cont'd)***

Name	Return	Arguments	Description
xip_array_#TYPE#_reserve_dim(1)	xip_status	Pointer to data structure, maximum number of dimensions.	Allocates a small array which is to contain the size of each dimension of the data array. For example, 100 samples x 4 channels x 3 fields.
xip_array_#TYPE#_destroy(1)	xip_status	Pointer to data structure.	Frees up the memory allocated for the data array, the dimension array, and the data structure itself.

1. #TYPE# can be uint, complex or mpz_complex.

Execution Functions

The run time functions of the C model are described in Table 8-6.

Table 8-6: **Execution Functions**

Name	Return	Arguments	Description
xip_cmpy_v6_0_data_do	xip_cmpy_v6_0_status	Pointer to C model, Pointer to input data structure, Pointer to output data structure, Number of samples, number of channels, number of fields input and number of fields output	The function which prompts execution of the C model. The number of samples, channels and fields must match the size of the array passed or an error is returned.
xip_array_#TYPE#_set_data(1)	xip_status	Pointer to array structure, the value to be written, the sample to be written to	Used to populate the input data structure.
xip_array_#TYPE#_get_data(1)	xip_status	Pointer to the array structure, pointer of #TYPE# type (returned value), sample to be read	Used to read the output (or input) data structure.

1. #TYPE# can be uint, complex or mpz_complex.

Compiling

Compilation of user code requires access to the `cmpy_v6_0_bitacc_cmodel.h` header file and the header file of the MPIR dependent library, `gmp.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 `cmpy_v6_0_bitacc_cmodel.h` header file must be included first, because it defines some symbols that are used in the MPIR header file. The `cmpy_v6_0_bitacc_cmodel.h` header file includes the MPIR header file, so it does 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 `cmpy_v6_0_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 an MPIR library. It is also possible to use GMP or MPIR libraries from other sources, for example, compiled from source code. For details, see Dependent Libraries.

Linux

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

- `libgmp.so.11`
- `libIp_cmpy_v6_0_bitacc_cmodel.so`

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

```
-L. -Wl, -rpath,. -lIp_cmpy_v6_0_bitacc_cmodel
```

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

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

```
gcc -x c++ -I. -L. -lIp_cmpy_v6_0_bitacc_cmodel -Wl, -rpath,. -o run_bitacc_cmodel
run_bitacc_cmodel.c
```

Windows

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

- `libgmp.dll`
- `libIp_cmpy_v6_0_bitacc_cmodel.dll`

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

- `libgmp.lib`
- `libIp_cmpy_v6_0_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 the MPIR library, which is governed by the GNU Lesser General Public License. You can obtain source code for the MPIR library from www.xilinx.com/guest_resources/gnu/. The following pre-compiled version of the MPIR library is provided with the C model:

- MPIR 2.6.0

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

GMP and MPIR 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 version of MPIR used to compile the provided libraries. Source code and compilation scripts for any version of the libraries can be obtained from the GMP [Ref 1] amd MPIR [Ref 12] web sites.

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.

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.

```
#include <iostream>
#include <complex>

#define _USE_MATH_DEFINES

#include <math.h>
#include <fstream> // for debug only
#include "cmpy_v6_0_bitacc_cmodel.h"
#include "gmp.h"

using namespace std;

#define DATA_SIZE 10
```

```
int main()
{
  size_t ii; //loop variable for data samples
  xip_uint roundbit;
  xip_complex value;

  // Create a configuration structure
  xip_cmpy_v6_0_config config, config_ret;
  xip_cmpy_v6_0_status status = xip_cmpy_v6_0_default_config(&config);
  if (status != XIP_CMPY_V6_0_STATUS_OK) {
    cerr << "ERROR: Could not get C model default configuration" << endl;
    return XIP_STATUS_ERROR;
  }

  //Configure this instance.
  config.APortWidth  = 16;
  config.BPortWidth  = 16;
  config.OutputWidth = 33;
  config.RoundMode   = XIP_CMPY_V6_0_TRUNCATE; //Note that the check later in this
file assumes full width

  // Create model object
  xip_cmpy_v6_0* cmpy_std;
  cmpy_std = xip_cmpy_v6_0_create(&config, &msg_print, 0);
  if (status != XIP_CMPY_V6_0_STATUS_OK) {
    cerr << "ERROR: Could not create C model state object" << endl;
    return XIP_STATUS_ERROR;
  }

  // Can we read back the updated configuration correctly?
  if (xip_cmpy_v6_0_get_config(cmpy_std, &config_ret) != XIP_CMPY_V6_0_STATUS_OK) {
    cerr << "ERROR: Could not retrieve C model configuration" << endl;
  }

  int number_of_samples = DATA_SIZE;
  // Create input data structure for operand A samples
  xip_array_complex* reqa = xip_array_complex_create();
  xip_array_complex_reserve_dim(reqa,1); //dimensions are (Number of samples)
  reqa->dim_size = 1;
  reqa->dim[0] = number_of_samples;
  reqa->data_size = reqa->dim[0];
  if (xip_array_complex_reserve_data(reqa,reqa->data_size) == XIP_STATUS_OK) {
    cout << "INFO: Reserved memory for request as [" << number_of_samples << "] array
" << endl;
  } else {
    cout << "ERROR: Unable to reserve memory for input data packet!" << endl;
    exit(2);
  }

  // Create input data structure for operand B samples
  xip_array_complex* reqb = xip_array_complex_create();
  xip_array_complex_reserve_dim(reqb,1); //dimensions are (Number of samples)
  reqb->dim_size = 1;
  reqb->dim[0] = number_of_samples;
  reqb->data_size = reqb->dim[0];
  if (xip_array_complex_reserve_data(reqb,reqb->data_size) == XIP_STATUS_OK) {
    cout << "INFO: Reserved memory for request as [" << number_of_samples << "] array
" << endl;
  } else {
```

```
    cout << "ERROR: Unable to reserve memory for input data packet!" << endl;
    exit(2);
  }

  // Create input data structure for ctrl input (Round bit)
  xip_array_uint* reqctrl = xip_array_uint_create();
  xip_array_uint_reserve_dim(reqctrl,1); //dimensions are (Number of samples)
  reqctrl->dim_size = 1;
  reqctrl->dim[0] = number_of_samples;
  reqctrl->data_size = reqctrl->dim[0];
  if (xip_array_uint_reserve_data(reqctrl,reqctrl->data_size) == XIP_STATUS_OK) {
    cout << "INFO: Reserved memory for request as [" << number_of_samples << "] array
" << endl;
  } else {
    cout << "ERROR: Unable to reserve memory for input data packet!" << endl;
    exit(2);
  }

  //create example input data
  xip_complex a,b;
  for (ii = 0; ii < DATA_SIZE; ii++)
  {
    roundbit = ii % 2;
    a.re = (xip_real)ii;
    a.im = (xip_real)ii;
    b.re = (xip_real)(16-ii);
    b.im = (xip_real)ii;
    if (xip_array_complex_set_data(reqa, a, ii) != XIP_STATUS_OK)
      cerr << "Error in xip_array_complex_set_data" << endl;
    if (xip_array_complex_set_data(reqb, b, ii) != XIP_STATUS_OK)
      cerr << "Error in xip_array_complex_set_data" << endl;
    if (xip_array_uint_set_data(reqctrl, roundbit, ii) != XIP_STATUS_OK)
      cerr << "Error in xip_array_uint_set_data" << endl;

  // Request memory for output data
  xip_array_complex* response = xip_array_complex_create();
  xip_array_complex_reserve_dim(response,1); //dimensions are (Number of samples)
  response->dim_size = 1;
  response->dim[0] = number_of_samples;
  response->data_size = response->dim[0];
  if (xip_array_complex_reserve_data(response,response->data_size) ==
XIP_STATUS_OK) {
    cout << "INFO: Reserved memory for response as [" << number_of_samples << "]
array " << endl;
  } else {
    cout << "ERROR: Unable to reserve memory for output data packet!" << endl;
    exit(3);
  }

  // Run the model
  cout << "Running the C model..." << endl;

  if (xip_cmpy_v6_0_data_do(cmpy_std, reqa, reqb, reqctrl, response) !=
XIP_CMPY_V6_0_STATUS_OK) {
    cerr << "ERROR: C model did not complete successfully" << endl;
    xip_array_complex_destroy(reqa);
    xip_array_complex_destroy(reqb);
    xip_array_uint_destroy(reqctrl);
    xip_array_complex_destroy(response);
```

```
    xip_cmpy_v6_0_destroy(cmpy_std);
    return XIP_STATUS_ERROR;
  } else {
    cout << "C model completed successfully" << endl;
  }

  // Check response is correct
  for (ii = 0; ii < DATA_SIZE; ii++)
  {
    //This example has natural width, so simple calculation
    xip_complex expected, got, x, y;
    xip_array_complex_get_data(reqa, &x, ii);
    xip_array_complex_get_data(reqb, &y, ii);
    xip_array_complex_get_data(response, &got, ii);

    //Note that the following equations assume that the output width is the full
natural
    //width of the calculation, i.e. neither truncation nor rounding occurs
    expected.re = x.re*y.re - x.im*y.im;
    expected.im = x.re*y.im + x.im*y.re;
    if (expected.re != got.re || expected.im != got.im) {
      cerr << "ERROR: C model data output is incorrect for sample" << ii << "Expected
real = " << expected.re << " imag = " << expected.im << " Got real = " << got.re <<
" imag = " << got.im << endl;

      xip_array_complex_destroy(reqa);
      xip_array_complex_destroy(reqb);
      xip_array_uint_destroy(reqctrl);
      xip_array_complex_destroy(response);
      xip_cmpy_v6_0_destroy(cmpy_std);
      return XIP_STATUS_ERROR;
    } else {
      cout << "Sample " << ii << " was as expected" << endl;
    }
  }
  cout << "C model data output is correct" << endl;

  // Clean up
  xip_array_complex_destroy(reqa);
  xip_array_complex_destroy(reqb);
  xip_array_uint_destroy(reqctrl);
  xip_array_complex_destroy(response);
  cout << "C model input and output data freed" << endl;

  xip_cmpy_v6_0_destroy(cmpy_std);
  cout << "C model destroyed" << endl;
```

Test Bench

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

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

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

Using the Demonstration Test Bench

The demonstration test bench instantiates the generated Complex Multiplier core.

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

The Demonstration Test Bench in Detail

The demonstration test bench performs the following tasks:

- Instantiate the core
- Generate two input data tables containing complex sinusoids of different frequencies
- Generate a clock signal
- Drive the core clock enable and reset input signals (if present)
- Drive the core input signals to demonstrate core feature
- Checks that the core output signals obey AXI protocol rules (data values are not checked in order to keep the test bench simple)
- Provide signals showing the separate fields of AXI 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 Complex Multiplier is treated as a mixer that combines two complex sinusoids with different but similar frequencies, but opposite sign and

different amplitude. The output of the core is therefore a complex sinusoid with a frequency equal to the difference in frequencies of the inputs, that is, a much slower frequency. The input data is pre-generated and stored in data tables, and the test bench drives the core data inputs with the sinusoid data throughout the operation of the test bench.

The demonstration test bench drives the AXI handshaking signals in different ways, split into three phases. The operations depend on whether Blocking Mode or NonBlocking Mode is selected:

- Blocking Mode:
  - Phase 1: full throughput, all TVALID and TREADY signals are tied High
  - Phase 2: apply increasing amounts of backpressure by deasserting the TREADY signal of the master channel
  - Phase 3: deprive slave channel A of valid transactions at an increasing rate by deasserting its TVALID signal
- NonBlocking Mode:
  - Phase 1: full throughput, all TVALID and TREADY signals are tied High
  - Phase 2: deprive slave channel A of valid transactions at an increasing rate by deasserting its TVALID signal
  - Phase 3: deprive all slave channels of valid transactions at different rates by deasserting each of their TVALID signals

Customizing the Demonstration Test Bench

It is possible to modify the demonstration test bench to drive the core inputs with different data or to perform different operations.

Input data is pre-generated in the `create_ip_a_table` and `create_ip_b_table` functions and stored in the `IP_A_DATA` and `IP_B_DATA` constants. New input data frames can be added by defining new functions and constants. Make sure that each input data frame is of an appropriate type, similar to the `T_IP_A_TABLE` and `T_IP_B_TABLE` array types.

All operations performed by the demonstration test bench to drive the inputs of the core are done in the `stimuli` process. This process is comprehensively commented, to explain clearly what is being done. New input data or different ways of driving AXI handshaking signals can be added by modifying sections of this process.

The total run time of the test can be modified by changing the `TEST_CYCLES` constant: this controls the number of clock cycles before the simulation is stopped.

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

XILINX

Appendix A

Migrating and Upgrading

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

Migrating to the Vivado Design Suite

For information about migrating to the Vivado Design Suite, *see the ISE to Vivado Design Suite Migration Guide* (UG911) [Ref 8].

Parameter Changes

The upgrade functionality can be used to update an existing XCO or XCI file from v3.1, v4.0 or v5.0 to Complex Multiplier v6.0, but is should be noted that the upgrade mechanism alone does not create a core compatible with v3.1. See Instructions for Minimum Change Migration (v3.1 to v6.0).

There are no parameter, port or functionality changes between v5.0 and v6.0. While there are no parameter changes from v4.0 to v6.0 there are two changes of note. The first is that the behavior of the NonBlocking mode has changed. The second is that the minimum latency for Blocking modes has been reduced by 6 cycles. This means that for a given performance, less latency is required. Retaining the same value of manual latency as v4.0 might result in a resource increase.

Table A-1 shows the changes to XCO parameters from version 3.1 to version 6.0.

Table A-1: **Parameter Changes from v3.1 to v6.0**

Version 3.1	Version 6.0	Notes
APortWidth	APortWidth	Unchanged, but the parameter no longer directly indicates the width of the port carrying the A operand
BPortWidth	BPortWidth	Unchanged, but the parameter no longer directly indicates the width of the port carrying the B operand
MultType	MultType	Unchanged
OptimizeGoal	OptimizeGoal	Unchanged

Table A-1: **Parameter Changes from v3.1 to v6.0 *(Cont'd)***

Version 3.1	Version 6.0	Notes
OutputWidthHigh	OutputWidth[(1)]	
OutputWidthLow		
RoundMode	RoundMode	Unchanged
Latency	LatencyConfig	Latency = -1 is now LatencyConfig = Automatic. All other values map to LatencyConfig = Manual
	MinimumLatency	This field allows the input of Latency when LatencyConfig=Manual, or reflects the actual value of MinimumLatency when LatencyConfig=Automatic
ClockEnable	ACLKEN	Renamed only
SyncClear	ARESETn	Renamed only. Note that while the sense of the aresetn signal has changed, this XCO determined whether or not the signal exists and has not changed. Note also that a minimum length of 2 cycles is recommended when aresetn is asserted.
SclrCEPriority		Deprecated. aresetn always overrides aclken in accordance with AXI4-Stream protocol.
	HasATLAST	Introduced in version 4.0
	HasATUSER	Introduced in version 4.0
	ATUSERWidth	Introduced in version 4.0
	HasBTLAST	Introduced in version 4.0
	HasBTUSER	Introduced in version 4.0
	BTUSERWidth	Introduced in version 4.0
	HasCTRLTLAST	Introduced in version 4.0
	HasCTRLTUSER	Introduced in version 4.0
	CTRLTUSERWidth	Introduced in version 4.0
	FlowControl	Introduced in version 4.0
	OutTLASTBehv	Introduced in version 4.0

Notes:

1. The natural output width of a complex multiplication is APortWidth+BPortWidth+1. When OutputWidth is set to be less than this, the most significant bits of the result are those output. The remaining bits are either truncated or rounded according to RoundMode. In other words OutputWidthHigh has been deprecated and is now fixed at (APortWidth +BPortWidth).

Port Changes

Table A-2 details the changes to port naming, additional or deprecated ports and polarity changes from v3.1 to v6.0 There are no changes from v4.0 to 6.0 nor from v5.0 to v6.0.

Table A-2: **Port Changes from Version 3.1 to Version 6.0**

Version 3.1	Version 6.0	Notes
CLK	aclk	Rename only
CE	aclken	Rename only
SCLR	aresetn	Rename and change of sense (now active-Low). Note recommendation that aresetn should be asserted for a minimum of 2 cycles.
AR(N-1:0)(1)	s_axis_a_tdata(N-1:0)	Both AR and AI map to s_axis_a_tdata
AI(N-1:0)	s_axis_a_tdata(N-1+byte(N):byte(N))	byte(N) is to round N up to the next multiple of 8.
BR(M-1:0)(2)	s_axis_b_tdata(M-1:0)	Both BR and BI map to s_axis_b_tdata
BI(M-1:0)	s_axis_b_tdata(M-1+byte(M):byte(M))	byte(M) is to round M up to the next multiple of 8.
ROUND_CY	s_axis_ctrl_tdata(0)	
PR(S-1:0)(3)	m_axis_dout_tdata(S-1:0)	Both PR and PI map to m_axis_dout_tdata
PI(S-1:0)	m_axis_dout_tdata(S-1+byte(S):byte(S))	byte(S) is to round S up to the next multiple of 8.
	s_axis_a_tvalid	TVALID (AXI4-Stream channel handshake signal) for each channel
	s_axis_b_tvalid	
	s_axis_ctrl_tvalid	
	m_axis_dout_tvalid	
	s_axis_a_tready	TREADY (AXI4-Stream channel handshake signal) for each channel.
	s_axis_b_tready	
	s_axis_ctrl_tready	
	m_axis_dout_tready	
	s_axis_a_tlast	TLAST (AXI4-Stream packet signal indicating the last transfer of a data structure) for each channel. The complex multiplier does not use TLAST, but provides the facility to pass TLAST with the same latency as TDATA.
	s_axis_b_tlast	
	s_axis_ctrl_tlast	
	m_axis_dout_tlast	
	s_axis_a_tuser (E-1:0)(4)	TUSER (AXI4-Stream ancillary field for application-specific information) for each channel. The complex multiplier does not use TUSER, but provides the facility to pass TUSER with the same latency as TDATA.
	s_axis_b_tuser (F-1:0)(5)	
	s_axis_ctrl_tuser (G-1:0)(6)	
	m_axis_dout_tuser (H-1:0)(7)	

Notes:

1. N is APortWidth
2. M is BPortWidth
3. S is OutputWidth
4. E is ATUSERWidth
5. F is BTUSERWidth
6. G is CTRLTUSERWidth
7. H is calculated from the input channel TUSERWidth and HASTUSER parameters.

Functionality Changes

Latency Changes

The latency of Complex Multiplier v6.0 is different compared to v3.1 for AXI Blocking mode. The update process cannot account for this and guarantee equivalent performance. Latency is the same as v3.1 in v6.0 for AXI NonBlocking mode. There are no changes to latency from v5.0 to v6.0.

The latency of AXI NonBlocking mode is reduced by 1 from v4.0 going to v6.0 except for the case of zero latency which is still zero in v6.0. The latency of AXI Blocking mode from v4.0 to v6.0 is reduced by 6 cycles.

IMPORTANT: When in Blocking Mode, the latency of the core is variable so only the minimum possible latency can be determined.

Instructions for Minimum Change Migration (v3.1 to v6.0)

To configure the Complex Multiplier v6.0 to most closely mimic the behavior of v3.1 the translation is as follows:

Parameters - Set FlowControl to NonBlocking and OutputWidth to APortWidth+BPortWidth+1-OutputWidthLow. All other new parameters default to FALSE and can be ignored. If OutputWidthHigh was previously set to other than APortWidth+BPortWidth then translation is not directly possible, but the (APortWidth+BPortWidth-OutputWidthHigh) most significant bits of the output can be ignored.

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

Performance and resource use are mostly unchanged versus v3.1 other than small changes due to the use of a different version of tools.

Instructions for Minimum Change Migration (v4.0 to v6.0)

For AXI Blocking Mode, the minimum latency is reduced by 6 cycles. The resource cost of Blocking mode is also reduced versus v4.0. This saving is proportional to the total number of input bits. NonBlocking behavior has also been changed relative to v4.0 to further reduce the resource cost of moving to AXI from pre-AXI versions such as v3.1. Latency in v6.0 is one cycle less than for v4.0 in NonBlocking mode except for the zero latency case, which is still zero.

Instructions for Minimum Change Migration (v5.0 to v6.0)

There are no changes of parameters, ports, or functionality from v5.0 to v6.0. No changes are required other than the version change.

Simulation

Starting with Complex Multiplier v6.0 (2013.3 version), behavioral simulation models have been replaced with IEEE P1735 Encrypted VHDL. The resulting model is bit and cycle accurate with the final netlist. For more information on simulation, see the *Vivado Design Suite User Guide: Logic Simulation* (UG900) [Ref 7].

Upgrading in the Vivado Design Suite

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

Parameter Changes

No change.

Port Changes

No change.

Other Changes

No change.

Debugging

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

Finding Help on Xilinx.com

To help in the design and debug process when using the Complex Multiplier, the Xilinx Support web page (www.xilinx.com/support) contains key resources such as product documentation, release notes, answer records, information about known issues, and links for obtaining further product support.

Documentation

This product guide is the main document associated with the Complex Multiplier. This guide, along with documentation related to all products that aid in the design process, can be found on the Xilinx Support web page (www.xilinx.com/support) or by using the Xilinx Documentation Navigator.

Download the Xilinx Documentation Navigator from the Design Tools tab on the Downloads page (www.xilinx.com/download). For more information about this tool and the features available, open the online help after installation.

Answer Records

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

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

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

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

Answer Records for the Complex Multiplier

AR: 54495

Contacting 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.

To contact Xilinx Technical Support:

1. Navigate to www.xilinx.com/support.
2. Open a WebCase by selecting the WebCase link located under Additional Resources.

When opening a WebCase, include:

- Target FPGA including package and speed grade.
- All applicable Xilinx Design Tools and simulator software versions.
- Additional files based on the specific issue might also be required. See the relevant sections in this debug guide for guidelines about which file(s) to include with the WebCase.

Note: Access to WebCase is not available in all cases. Login to the WebCase tool to see your specific support options.

Debug Tools

Vivado Lab Tools

Vivado® lab tools insert logic analyzer and virtual I/O cores directly into your design. Vivado lab tools allow you to set trigger conditions to capture application and integrated block port signals in hardware. Captured signals can then be analyzed. This feature represents the functionality in the Vivado IDE that is used for logic debugging and validation of a design running in Xilinx devices in hardware.

The Vivado lab tools logic analyzer is used to interact with the logic debug LogiCORE IP cores, including:

- ILA 2.0 (and later versions)

- VIO 2.0 (and later versions)

See Vivado Design Suite User Guide: Programming and Debugging (UG908) [Ref 9].

C Model

See *Chapter 8, C Model Reference* in this guide for tips and instructions for using the provided C Model files to debug your design.

Simulation Debug

The simulation debug flow for Questa SIM is illustrated in Figure B-1. A similar approach can be used with other simulators.

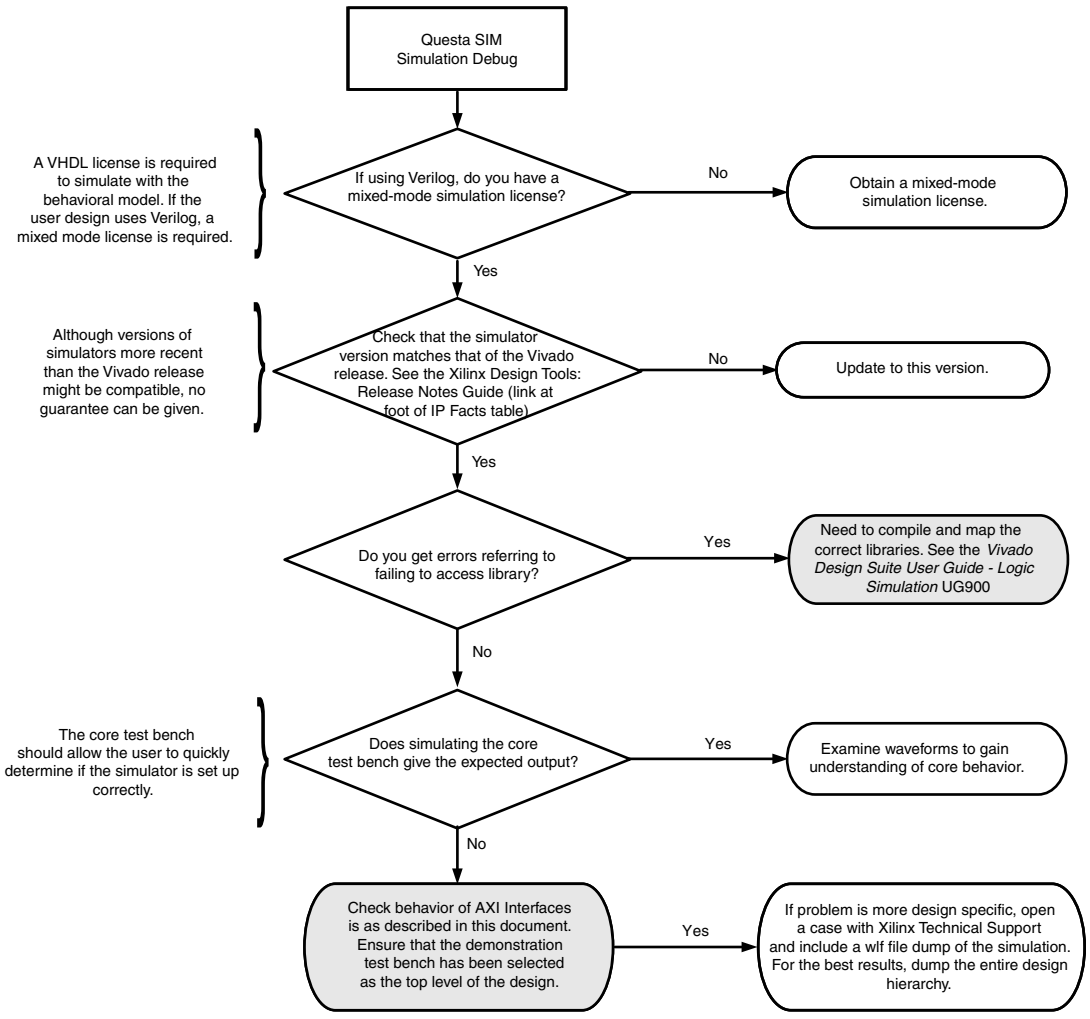

Figure B-1: **Quests SIM Debug Flow**

Interface Debug

AXI4-Stream Interfaces

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

- If transmit <*interface_name*>_tready is stuck Low following the **<***interface_name***>_tvalid** input being asserted, the core cannot send data.
- If the receive **<***interface_name***>**_tvalid is stuck Low, the core is not receiving data.
- Check that the ACLK inputs are connected and toggling.
- Check that the AXI4-Stream waveforms are being followed (see Figure 3-4).
- Check core configuration.

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.

References

These documents provide supplemental material useful with this product guide:

1. *Xilinx AXI Reference Guide* (UG761)
2. *AMBA® AXI4-Stream Protocol Specification* (ARM IHI 0051A)
3. *Vivado Design Suite User Guide: Designing IP Subsystems using IP Integrator* (UG994)
4. *Vivado Design Suite User Guide: Implementation* (UG904)
5. *Vivado Design Suite User Guide: Designing with IP* (UG896)
6. *Vivado Design Suite User Guide: Getting Started* (UG910)
7. *Vivado Design Suite User Guide - Logic Simulation* (UG900)
8. *ISE® to Vivado Design Suite Migration Guide* (UG911)
9. *Vivado Design Suite User Guide: Programming and Debugging* (UG908)
10. *System Generator for DSP User Guide* (UG640)
11. The GNU Multiple Precision Arithmetic (GMP) Library gmplib.org
12. The GNU Multiple Precision Integers and Rationals (MPIR) library www.mpir.org
13. Multiple Precision Arithmetic on Windows, Brian Gladman: http://gladman.plushost.co.uk/oldsite/computing/gmp4win.php

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
03/20/2013	1.0	Initial release as a Product Guide; replaces DS793. Debug appendix added. No other documentation changes.
06/19/2013	6.0	Added C Model Reference chapter. Document version number advanced to match the core version number.
10/02/2013	6.0	Minor updates to IP Facts table and Migrating appendix. Updates to C Model Reference chapter.
12/18/2013	6.0	Added UltraScale™ architecture support.

Notice of Disclaimer