LogiCORE IP Complex Multiplier v4.0

DS793 March 1, 2011 **Product Specification**

Overview

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 via 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 Virtex®-7 and Kintex™-7, Virtex®-6 and Spartan®-6 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 XtremeDSP™ slices
- Resource estimation in the Xilinx® CORE Generator® graphical user interface (GUI)
- For use with Xilinx CORE Generator tool and Xilinx System Generator for DSP 13.1

LogiCORE IP Facts	
Core Specifics	
Supported Device Family(1)	Kintex-7, Virtex-7 Virtex-6, Spartan-6
Supported User Interfaces	AXI4-Stream
Configuration	See Table 6 and Table 7
Provided with Core	
Documentation	Product Specification
Design Files	Netlist
Example Design	Not Provided
Test Bench	VHDL
Constraints File	N/A
Simulation Model	VHDL and Verilog
Tested Design Tools	
Design Entry Tools	CORE Generator 13.1 System Generator for DSP 13.1
Simulation	Mentor Graphics ModelSim 6.6d Cadence Incisive Enterprise Simulator (IES) 10.2 Synopsys VCS and VCS MX 2010.06 ISIM 13.1
Synthesis Tools	N/A
Support	
Provided by Xilinx, Inc.	

1. For the complete list of supported devices, see the release notes for this core.

Functional Description

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:

$$p_r = a_r b_r - a_i b_i$$ *Equation 1*

$$p_i = a_r b_i + a_i b_r$$ *Equation 2*

By exploiting that

$$p_r = a_r b_r - a_i b_i = a_r(b_r + b_i) - (a_r + a_i)b_i$$ *Equation 3*

$$p_i = a_r b_i + a_i b_r = a_r(b_r + b_i) + (a_i - a_r)b_r$$ *Equation 4*

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

Pinout

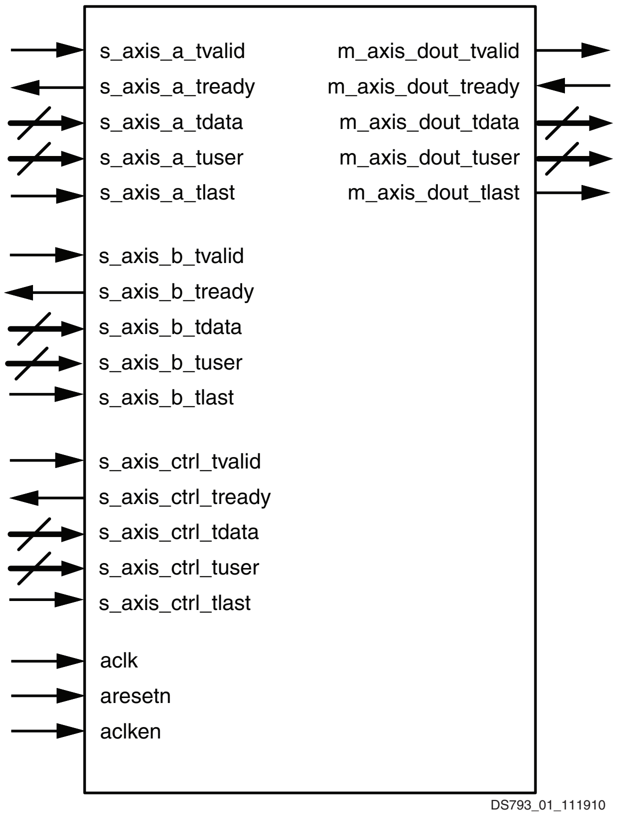

Figure 1: **Core Schematic Symbol**

This section describes the Complex Multiplier core ports as shown in Figure 1 and described in Table 1.

Table 1: **Core Signal Pinout**

Name	Direction	Optional	Description
aclk	Input	no	Rising-edge clock
aclken	Input	yes	Active-high clock enable (optional)
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 XCO parameters.

CORE Generator Graphical User Interface

The Complex Multiplier core GUI has a number of fields to set parameter values for the particular instantiation required. This section provides a description of each GUI 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 "_".

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.
- **Has TLAST:** Select whether the channel has TLAST. To ease system design, the core will pass 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 will pass 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.

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.
- **Has TLAST:** Select whether the channel has TLAST. To ease system design, the core will pass 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 will pass 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.

Multiplier Construction Options: Allows the choice of using LUTs (slice logic) to construct the complex multiplier, or using XtremeDSP 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 will generally use 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.

Input Behavior: Selects between Blocking and Non blocking behavior for the AXI4-Stream Interfaces. See Non Blocking Mode and Blocking Mode for greater detail.

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 will be 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 will be enabled. Bit 0 of the TDATA field of this channel determines the particular type of rounding for the operation in question. See the Rounding section for further details.

Channel CTRL Options: The control channel exists to supply the bit which will determine 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 via the core.

- **Has TLAST:** Select whether the channel has TLAST. To ease system design, the core will pass 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 will pass 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.

Output TLAST Behavior

- TLAST Behavior: Determines which of the input channels' TLAST or which combination of input channel TLASTs will be 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 will be set such that the core is fully pipelined for maximum performance. Manual allows user-selectable minimum latency. When the value set is less than the fully pipelined latency, performance will drop. When the value set is larger than fully pipelined, the core will delay the output via an SRL. Note that with Blocking Input Behavior 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 will be enabled by this signal.
- **ARESETn:** Enables the active low synchronous clear (`aresetn`) pin on the core. All registers in the core will be 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`.

Resource Estimation Tab: Clicking on the tab below the GUI symbol displays an estimate of the XtremeDSP 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.

Using the Complex Multiplier IP Core

The CORE Generator GUI performs error-checking on all input parameters. Resource estimation and latency information are also available.

Several files are produced when a core is generated, and customized instantiation templates for Verilog and VHDL design flows are provided in the .veo and .vho files, respectively. For detailed instructions, see the CORE Generator software documentation.

Simulation Models

The core has a number of options for simulation models:

- VHDL behavioral model in the xilinxcorelib library
- VHDL UniSim structural model
- Verilog UniSim structural model

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

Xilinx recommends that simulations utilizing UniSim-based structural models are 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 models might produce incorrect results if simulation with a resolution other than 1 ps. See the "Register Transfer Level (RTL) Simulation Using Xilinx Libraries" section in *Synthesis and Simulation Design Guide* for more information. This document is part of the ISE® Software Manuals set available at www.xilinx.com/support/documentation/dt_ise.htm.

XCO Parameters

Table 2 defines valid entries for the XCO parameters. Parameters are not case sensitive. Default values are displayed in bold.

Xilinx strongly suggests that XCO parameters are not manually edited in the XCO file; instead, use the CORE Generator GUI to configure the core and perform range and parameter value checking.

Table 2: **XCO Parameters**

XCO Parameter	Valid Values
component_name	ASCII text using characters: a to z, A to Z, 0 to 9 and '_' ; starting with a letter
APortWidth	8 - 63 (default value is **16**)
HasATLAST	**false**, true
HasATUSER	**false**, true
ATUSERWidth	**1** to 256
BPortWidth	8 - 63 (default value is **16**)
HasBTLAST	**false**, true
HasBTUSER	**false**, true
BTUSERWidth	**1** to 256
MultType	Use_LUTs, **Use_Mults**
OptimizeGoal	**Resources**, Performance
FlowControl	Blocking**, Non-Blocking**
OutputWidth	APortWidth+BPortWidth (default value is **32**)
RoundMode	**Truncate**, Random_Rounding
HasCTRLTLAST	**false**, true
HasCTRLTUSER	**false**, true
CTRLTUSERWidth	**1** to 256
OutTLASTBehv	**Null,** Pass_A_TLAST, Pass_B_TLAST, Pass_CTRL_TLAST, OR_all_TLASTs, AND_all_TLASTs
LatencyConfig	**Automatic**, Manual
MinimumLatency	0 to 59 (default value and range depend on configuration of other XCO parameters)
ACLKEN	**false**, true
ARESETn	**false**, true

Demonstration Test Bench

When the core is generated using CORE Generator, 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 Complex Multiplier core. Either the behavioral model or the netlist can be simulated within the demonstration test bench.

- Behavioral model: Ensure that the CORE Generator project options are set to generate a behavioral model. After generation, this creates a behavioral model wrapper named `<component_name>.vhd`. Compile this file into the work library (see your simulator documentation for more information on how to do this).
- Netlist: 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 option was not set, generate a netlist using the netgen program, for example in Unix:

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

  Compile the netlist into the work library (see your simulator documentation for more information on how to do this).

Compile the demonstration test bench into the work library. 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:

- Instantiate the core
- Generate two input data tables containing complex sinusoids of different frequencies
- Generate a clock signal
- Drive the core's clock enable and reset input signals (if present)
- Drive the core's input signals to demonstrate core features (see below for details)
- Checks that the core's 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's input signals to demonstrate the features and modes of operation of the core. The Complex Multiplier is treated as a mixer that is combining 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's 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 Non-Blocking 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 master channel's TREADY signal
    - Phase 3: deprive slave channel A of valid transactions at an increasing rate by deasserting its TVALID signal
- Non-Blocking 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's 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 core's inputs 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 runtime 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.

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 CORE Generator GUI. The Complex Multiplier core may be found in the Xilinx Blockset in the Math section. The block is called "Complex Multiplier 4.0." See the System Generator for DSP Help page for the "Complex Multiplier 4.0" block for more information on parameters not mentioned here.

Page 1

Page 1 is used to specify the complex multiplier construction, optimization options and output width in a similar way to the CORE Generator GUI. Please refer to CORE Generator Graphical User Interface.

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 CORE Generator GUI. Please refer to CORE Generator Graphical User Interface.

Implementation

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

Rounding

In a DSP system, especially if the system contains feedback, the wordlength growth through the multiplier should be offset by quantizing the results. Quantization, or reduction in wordlength, 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 2 illustrates the quantization method used for truncation.

.

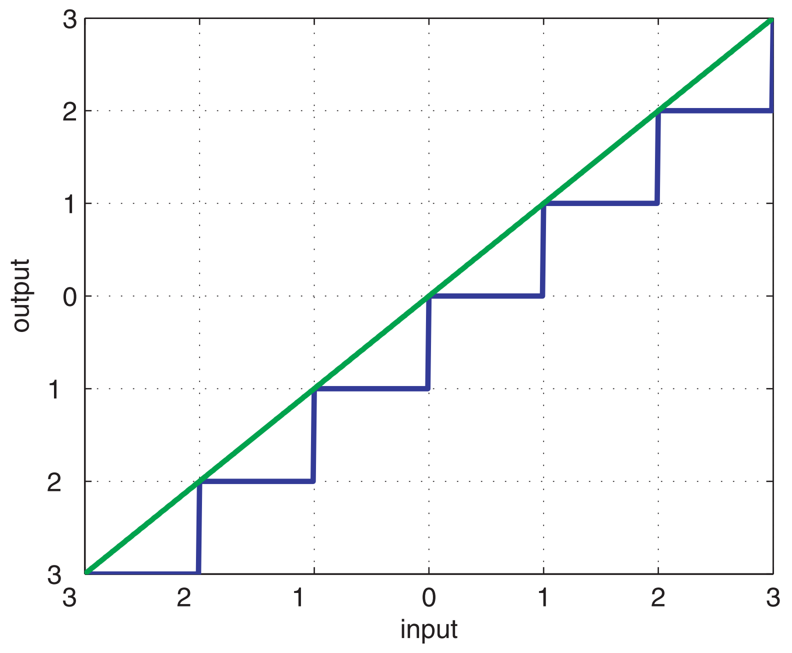

Figure 2: **Truncation**

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

$$p(e) = \begin{cases} \frac{1}{\Delta} & -\Delta < e < 0 \\ 0 & otherwise \end{cases} \qquad \textit{Equation 5}$$

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

$$m_e = \int_{-\Delta}^{0} ep(e)de = \frac{1}{\Delta}\int_{-\Delta}^{0} ede = -\frac{\Delta}{2} \qquad \textit{Equation 6}$$

$$\sigma_e^2 = \int_{0}^{\Delta} e^2 p(e)de = \frac{1}{\Delta}\int_{0}^{\Delta} e^2 de = \frac{\Delta^2}{3} \qquad \textit{Equation 7}$$

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

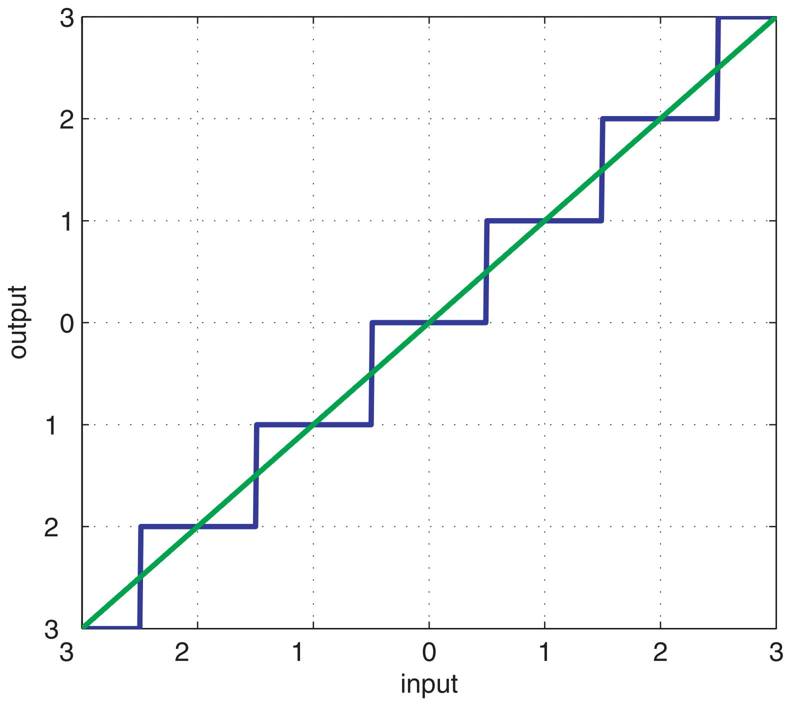

Figure 3: **Rounding**

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 8*

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 9*

$$\sigma_e^2 = \int_{-\Delta/2}^{\Delta/2} e^2 p(e)de = \frac{1}{\Delta}\int_{-\Delta/2}^{\Delta/2} e^2 de = \frac{\Delta^2}{12}$$ *Equation 10*

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.

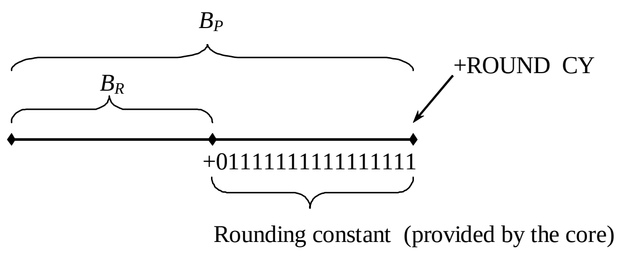

Figure 4: **Rounding to B_r Bits from B_p Bits**

To implement the rounding function shown in Figure 3, 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 4), 0.5 would be always rounded down, introducing negative bias.

If 0.5 is rounded using a static rule, the resulting quantization will always introduce bias. To avoid bias, rounding has to 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.

Table 3: **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 XtremeDSP slices are involved in the process, such as evaluation of Equation 9 or Equation 10. 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 XtremeDSP slices. Therefore an external signal should be used to feed the round carry input, through the *ROUND_CY* 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 the user 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 XtremeDSP slice implementation, the addition of the rounding constant is ’free,’ as the C port and carry-in input may be utilized. In devices without XtremeDSP slices, the addition of rounding typically requires an extra slice-based adder and an additional cycle of latency.

Hardware Implementation

Three Real Multiplier Solution

The three real multiplier implementation maps well to devices that have a pre-adder as part of the XtremeDSP slice (such as Spartan-6 and Virtex-6 FPGAs), saving slice resources.

In general, the three multiplier solution will use 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 XtremeDSP 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 device.

It will still consume 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 XtremeDSP slice implementations, it might be suitable for applications where XtremeDSP 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.

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 via 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 expected to ease use of the Complex Multiplier in a system. For example, the complex multiplier may 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 Design Reference Guide (UG761) and the AMBA 4 AXI4-Stream Protocol Version: 1.0 Specification.

Basic Handshake

Figure 5 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 will set TVALID and TREADY respectively for the next transfer appropriately.

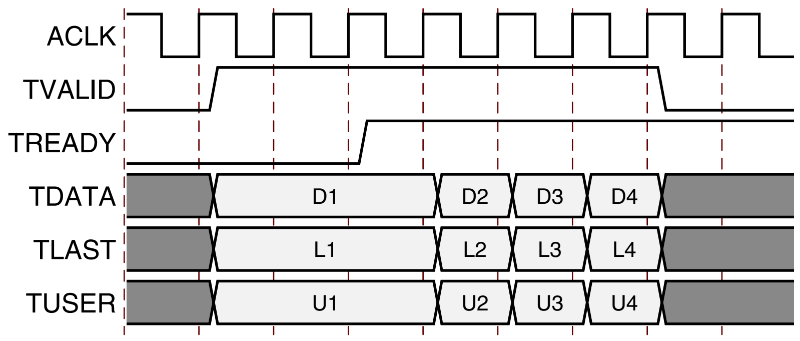

Figure 5: **Data Transfer in an AXI-Stream Channel**

Non Blocking Mode

The term 'Non-Blocking' means that lack of data on one input channel will 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 via the FlowControl parameter or GUI field. The Complex Multiplier core supports a 'non-blocking' mode in which the AXI4-Stream channels do not have TREADY, that is, they do not support back pressure. Without TREADY, blocking behavior would cause loss of data, so in the Complex Multiplier the concepts of Non-blocking and the half-handshake of interfaces without TREADY are selected or deselected by a single control. 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.

When any of the present input channels receives an active TVALID, an operation is performed. For any of the channels which did not receive a TVALID in that cycle, the last validated payload (TDATA, TLAST and TUSER) will be re-used. Zeros are used if there was no previously validated input message.

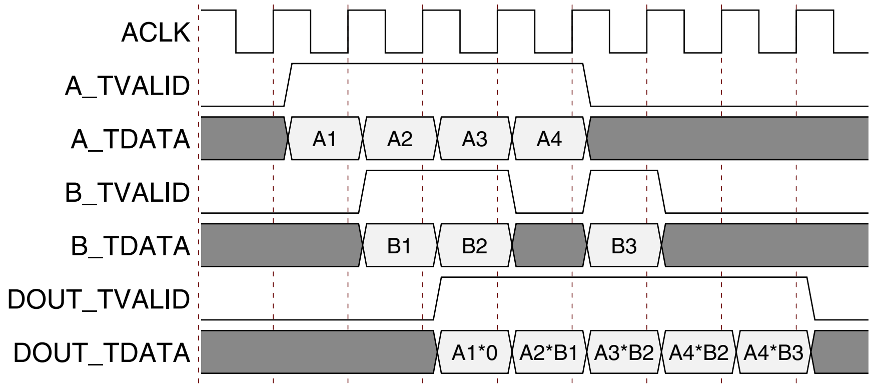

Figure 6: **Non Blocking Mode**

Blocking Mode

The term 'Blocking' means that operation execution will 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 Non-blocking behavior is selected via 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 will occur and the result will become available on the output. If the output is prevented from off-loading data because TREADY is low then data will accumulate in the output buffer internal to the core. When this output buffer is nearly full the core will stop further operations. This prevents the input buffers from off-loading data for new operations so the input buffers will fill as new data is input. When the input buffers fill, their respective TREADYs will be 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 de-asserted until the starved channel receives some data. Figure 7 shows both blocking behavior and back-pressure. Note that 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 will eventually lead to a saturation of the core's buffers, leading the core to signal that it can no longer accept further input by de-asserting 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 will act as a queue, ensuring that the first, second, third data samples on each channel will be 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.

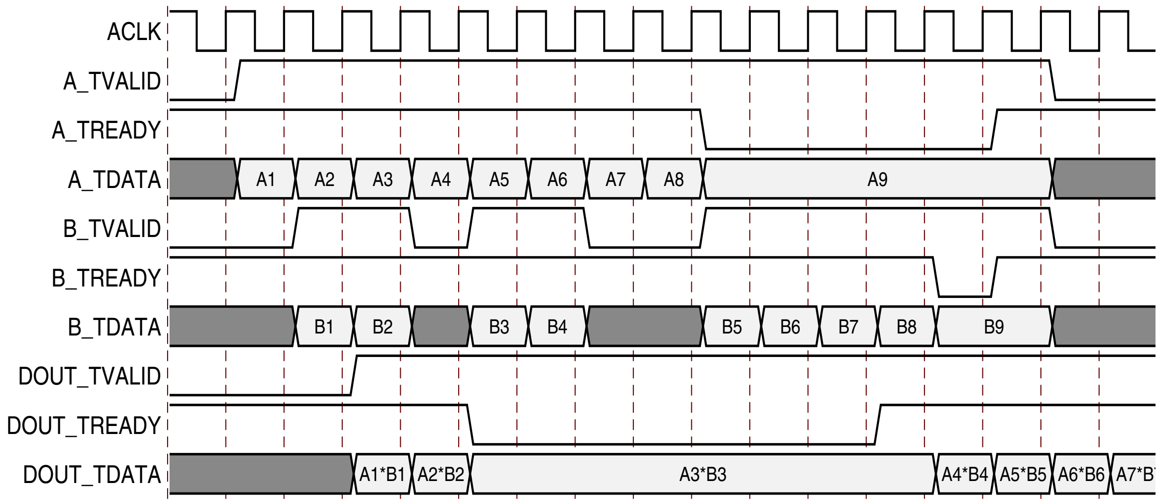

Figure 7: **Blocking Mode**

Note: This diagram is for illustration of the blocking behavior and handshake protocol. Latencies implied by the diagram may not be accurate.

TDATA Packing

Fields within an AXI4-Stream interface follow a specific naming nomenclature. See Figure 8. Normally, information pertinent to the application, complex multiplication in this case, is carried in the TDATA field. In version 4.0 the complex operand components, real and imaginary, are both passed to or from the core via the channel's 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. Note that 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 will occupy the least significant bits. The imaginary component will occupy a bit field which starts on the next byte-boundary above the real component as exemplified in the previous section. See Figure 8.

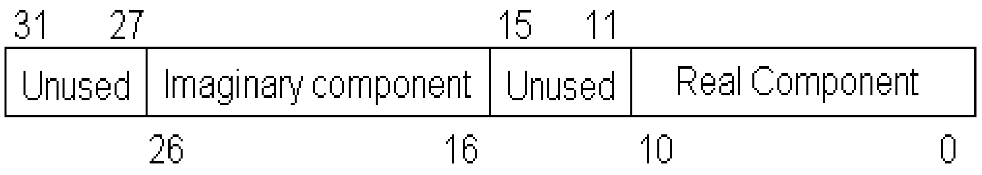

Figure 8: **TDATA Structure for A, B and DOUT Channels**

TDATA Structure for CTRL Channel

The CTRL channel exists only when Rounding has been selected and exists to convey only the rounding bit. This bit occupies bit 0 of TDATA for this channel. However, due to the byte-oriented nature of TDATA, this means that TDATA will have a width of 8 bits.

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 datastream 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 via 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 shall not have TLAST either.

TUSER Options

TUSER for each input channel is optional. Each has user-selectable width. These fields are simply concatenated, without any byte-orientation or padding, to form the output channel TUSER field. The TUSER field from channel A will form 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 will be 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`) will have the bits of B_TUSER (`s_axis_b_tuser`) suitably delayed in positions 3 downto 0 with CTRL_TUSER (`s_axis_ctrl_tuser`) bits, suitably delayed, in positions 13 downto 4.

Migrating to Complex Multiplier v4.0 from Earlier Versions

XCO Parameter Changes

The CORE Generator core update functionality may be used to update an existing XCO file from v3.1 to Complex Multiplier v4.0, but it should be noted that the update mechanism alone will not create a core compatible with v3.1. See Instructions for Minimum Change Migration. Complex Multiplier v4.0 has additional parameters for AXI4-Stream support. The following table shows the changes to XCO parameters from version 3.1 to version 4.0.

Table 4: **XCO Parameter Changes from v3.1 to v4.0**

Version 3.1	Version 4.0	Notes
APortWidth	APortWidth	Unchanged, but note that the parameter no longer directly indicates the width of the port carrying the A operand
BPortWidth	BPortWidth	Unchanged, but note that the parameter no longer directly indicates the width of the port carrying the Boperand
MultType	MultType	Unchanged
OptimizeGoal	OptimizeGoal	Unchanged
OutputWidthHigh	OutputWidth	Note 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	New to version 4.0
	HasATUSER	New to version 4.0
	ATUSERWidth	New to version 4.0
	HasBTLAST	New to version 4.0
	HasBTUSER	New to version 4.0
	BTUSERWidth	New to version 4.0
	HasCTRLTLAST	New to version 4.0
	HasCTRLTUSER	New to version 4.0
	CTRLTUSERWidth	New to version 4.0
	FlowControl	New to version 4.0
	OutTLASTBehv	New to 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 will be those output. The remaining bits will either be truncated or rounded according to RoundMode. In other words OutputWidthHigh has been deprecated and is now fixed at (APortWidth +BPortWidth).

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

Port Changes

The following table details the changes to port naming, additional or deprecated ports and polarity changes from v3.1 to v4.0.

Table 5: **Port Changes from Version 3.1 to Version 4.0**

Version 3.1	Version 4.0	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)	

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.

Latency Changes

The latency of Complex Multiplier v4.0 will be different compared to v3.1 and greater in general. The update process cannot account for this and guarantee equivalent performance.

Importantly, when in Blocking Mode, the latency of the core will be variable, so only the minimum possible latency can be determined. When in Non-Blocking Mode, the latency of the core for equivalent performance will be one greater than that for the equivalent configuration of v3.1.

Instructions for Minimum Change Migration

To configure the Complex Multiplier v4.0 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 will default to false and may 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 may simply be ignored.

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

Performance - To achieve equivalent performance to v3.1 MinimumLatency should be set to Latency+1 (except Latency=-1). Alternatively, set MinimumLatency to be the same as Latency for v3.1, but there may be drop in performance. Resource allocation for this configuration will be approximately (APortWidth+BPortWidth) flipflops more than for v3.1.

Performance and Resource Usage

Table 6 through Table 7 provide performance and resource usage information for a number of different core configurations.

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 preceding "characterization" registers and represent the true logic used by the core to implement a single Complex Multiplier. LUT counts include SRL16s or SRL32s (according to device family) and LUTs used as route-throughs.

The map options used were: `"map -pr b -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 Virtex-6 FPGA test cases in Table 6 used an XC6VLX75T-FF784 (-1 speed grade) device and ISE software speed file version "PRELIMINARY 1.08 2010-07-20."

Table 6: **Virtex-6 FPGA Performance and Resource Usage**

Complex Multiplier Configuration	Maximum Clock Frequency (MHz)	LUT/FF Pairs	LUT6s	FFs	XtremeDSP Slices
16x16 Use Mults Resources, NonBlocking	450	74	58	131	3
16x16 Use Mults Resources, Blocking	450	265	223	358	3
16x16 Use Mults Resources, NonBlocking, with 32 bit TUSER and TLAST	450	203	64	230	3
16x16 Use Mults Resources, Blocking, with 32 bit TUSER and TLAST	450	472	344	556	3
16x16 Use Mults Resources, Non-Blocking, Truncated	450	87	46	131	3
16x16 Use Mults Resources, Non-blocking, Rounded (has CTRL channel)	450	97	40	133	3
16x16 Use Mults Performance	450	50	18	67	4
16x16 Use LUTs Resources	413	1143	1042	1147	0
32x16 Use Mults Resources	450	280	126	362	6
32x16 Use Mults Performance	450	138	44	179	8
32x16 Use LUTs Resources	365	2140	2025	2192	0
32x32 Use Mults Resources	399	478	331	655	12
32x32 Use Mults Performance	450	301	127	417	16
32x32 Use LUTs Resources	294	3892	3739	3828	0

The Spartan-6 FPGA test cases in Table 7 used an XC6SLX45T-FGG484 (-2 speed grade) device and ISE software speed file version "PRODUCTION 1.11 2010-07-20."

Table 7: **Spartan-6 FPGA Performance and Resource Usage**

Complex Multiplier Configuration	Maximum Clock Frequency (MHz)	LUT/FF Pairs	LUT6s	FFs	XtremeDSP Slices
16x16 Use Mults Resources, Nonblocking	294	98	34	131	3
16x16 Use Mults Resources, Blocking	259	232	174	358	3
16x16 Use Mults Resources, Nonblocking, with 32 bit TUSER and TLAST	308	164	69	230	3
16x16 Use Mults Resources, Blocking, with 32 bit TUSER and TLAST	246	350	258	556	3
16x16 Use Mults Resources, Nonblocking, Truncating	301	91	42	131	3
16x16 Use Mults Resources, Nonblocking, Rounding (has CTRL channel)	308	96	39	133	3
16x16 Use Mults Performance	301	63	6	67	4
16x16 Use LUTs Resources	210	1134	1045	1132	0
32x16 Use Mults Resources	202	276	134	362	6
32x16 Use Mults Performance	231	158	56	209	8
32x16 Use LUTs Resources	189	2224	1921	2254	0
32x32 Use Mults Resources	175	484	346	643	12
32x32 Use Mults Performance	246	336	126	449	16
32x32 Use LUTs Resources	161	4015	3724	4021	0

Support

Xilinx provides technical 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 further 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
- Bug Fixes
- Known Issues

Ordering Information

This LogiCORE IP module is included at no additional cost with the Xilinx ISE Design Suite software and is provided under the terms of the Xilinx End User License Agreement. Use the CORE Generator software included with the ISE Design Suite to generate the core. For more information, please visit the core page.

Please contact your local Xilinx sales representative for pricing and availability of additional Xilinx LogiCORE modules and software. Information about additional Xilinx LogiCORE modules is available on the Xilinx IP Center.

References

1. IP Release Notes Guide (XTP025)
2. *Xilinx AXI Reference Guide* (UG761) available at www.xilinx.com/support/ip_documentation/ug761_axi_reference_guide.pdf.
3. AMBA 4 AXI4-Stream Protocol Version: 1.0 Specification
4. Synthesis and Simulation Design Guide

List of Acronyms

Acronym	Spelled Out
AXI	Advanced eXtensible Interface
DC	Direct Current
DSP	Digital Signal Processing
FF	Flip-Flop
FPGA	Field Programmable Gate Array
GUI	Graphical User Interface
I/O	Input/Output
IP	Intellectual Property
ISE	Integrated Software Environment
LUT	Lookup Table
MHz	Mega Hertz
ND	New Data
PDF	Probability Density Function
RTL	Register Transfer Level
SRL	Shift Register LUT
VHDL	VHSIC Hardware Description Language (VHSIC an acronym for Very High-Speed Integrated Circuits)
XCO	Xilinx CORE Generator core source file
XST	Xilinx Synthesis Technology

Revision History

Date	Version	Description of Revisions
09/21/10	1.0	First release of the core with AXI interface support. The previous release of this document was ds291.
03/01/11	1.1	Support added for Virtex-7 and Kintex-7. ISE Design Suite 13.1

Notice of Disclaimer