CORDIC v6.0

LogiCORE IP Product Guide

Vivado Design Suite

PG105 October 4, 2017

XILINX
ALL PROGRAMMABLE™

Table of Contents

Introduction

This Xilinx® LogiCORE™ IP core implements a generalized coordinate rotational digital computer (CORDIC) algorithm.

Features

- Functional configurations
- Optional coarse rotation module to extend the range of CORDIC from the first quadrant (+Pi/4 to - Pi/4 Radians) to the full circle
- Optional amplitude compensation scaling module to compensate for the output amplitude scale factor of the CORDIC algorithm
- Output rounding modes: Truncation, Round to Pos Infinity, Round to Pos/Neg Infinity, and Round to Nearest Even
- Word serial architectural configuration for small area
- Parallel architectural configuration for high throughput
- Control of the internal add-sub precision
- Control of the number of add-sub iterations
- X and Y data formats: Signed Fraction, Unsigned Fraction, and Unsigned Integer
- Phase data formats: Radian, Pi Radian
- Fully synchronous design using a single clock

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

Notes:

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

Overview

The CORDIC core implements a generalized coordinate rotational digital computer (CORDIC) algorithm, initially developed by Volder [Ref 1] to iteratively solve trigonometric equations, and later generalized by Walther [Ref 2] to solve a broader range of equations, including the hyperbolic and square root equations. The CORDIC core implements the following equation types:

- Rectangular <-> Polar Conversion
- Trigonometric
- Hyperbolic
- Square Root

Two architectural configurations are available for the CORDIC core:

- A fully parallel configuration with single-cycle data throughput at the expense of silicon area
- A word serial implementation with multiple-cycle throughput but occupying a small silicon area

A coarse rotation is performed to rotate the input sample from the full circle into the first quadrant. (The coarse rotation stage is required as the CORDIC algorithm is only valid over the first quadrant). An inverse coarse rotation stage rotates the output sample into the correct quadrant.

The CORDIC algorithm introduces a scale factor to the amplitude of the result, and the CORDIC core provides the option of automatically compensating for the CORDIC scale factor.

The CORDIC algorithm can be used to solve several functions as described above. These functions take different combinations of Cartesian and polar operands. The operands X_IN and Y_IN are input using the S_AXIS_CARTESIAN channel and the PHASE_IN operand is input using the S_AXIS_PHASE input.

Feature Summary

- Vector rotation (polar to rectangular)
- Vector translation (rectangular to polar)
- Sin and Cos
- Sinh and Cosh
- Atan
- Atanh
- Square root

Applications

The CORDIC core can be used to implement any of the general purpose functions listed in Feature Summary.

Licensing and Ordering

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

Product Specification

The equations used to define the CORDIC are detailed in Functional Description in Chapter 3.

Performance

The latency and throughput of the core is influenced by the selection of Parallel or Serial Architecture. The resulting basic latency and throughput are described in Parallel Architectural Configuration and Word Serial Architectural Configuration, though it should be noted that latency is affected by the form of AXI4-Stream protocol selected. The CORDIC user interface in the Vivado® Integrated Design Environment (IDE) shows the latency for the selected configuration. It should be stated that when AXI blocking mode is selected, latency should not be a primary design consideration, because the AXI protocol manages data traffic dynamically.

Two architectural configurations are available for the CORDIC core:

- Parallel, with single-cycle data throughput and large silicon area
- Word Serial, with multiple-cycle throughput and a smaller silicon area.

This choice is independent of choices relating to AXI4-Stream behavior.

Parallel Architectural Configuration

The CORDIC algorithm requires approximately one shift-addsub operation for each bit of accuracy. A CORDIC core with a parallel architectural configuration implements these shift-addsub operations in parallel using an array of shift-addsub stages.

A parallel CORDIC core with N bit output width has a latency of N cycles and produces a new output every cycle. The implementation size of this parallel circuit is directly proportional to the internal precision times the number of iterations.

Word Serial Architectural Configuration

The CORDIC algorithm requires approximately one shift-addsub operation for each bit of accuracy. A CORDIC core implemented with the word serial architectural configuration,

implements these shift-addsub operations serially, using a single shift-addsub stage and feeding back the output.

A word serial CORDIC core with N bit output width has a latency of N cycles and produces a new output every N cycles. The implementation size this iterative circuit is directly proportional to the internal precision.

Resource Utilization

For details about performance, visit Performance and Resource Utilization.

Port Descriptions

A block diagram of the CORDIC core is presented in Figure 2-1.

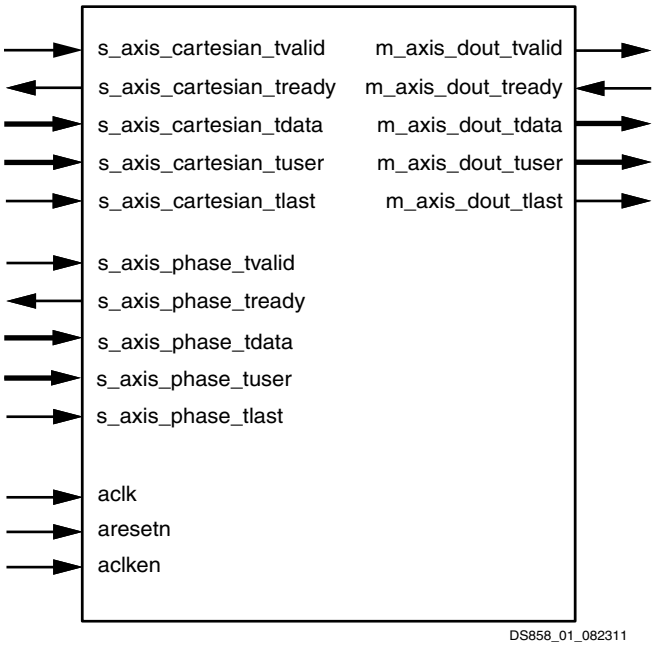

Figure 2-1: **CORDIC Symbol and Pinout**

Table 2-1 describes the port names in Figure 2-1.

Table 2-1: **Core Pinout**

Port Name	Direction	Description
aclk	In	Clock. Active rising edge.
ACLKEN	In	Clock Enable. Active-High
ARESETn	In	Synchronous Reset. Active-Low. ARESETn must be active for at least 2 clock cycles when asserted.
s_axis_cartesian_tvalid	In	Handshake signal for channel S_AXIS_CARTESIAN.(1)
s_axis_cartesian_tready	Out	Handshake signal for channel S_AXIS_CARTESIAN.(1)
s_axis_cartesian_tdata[A-1:0]	In	Depending on Functional Configuration, this port has one or two subfields; X_IN and Y_IN. These are the Cartesian operands. Each subfield is Input_Width bits wide, padded to the next byte width before being concatenated. See TDATA Packing.
s_axis_cartesian_tuser[B-1:0]	In	Data on this port is delayed with the same latency as tdata and appear on m_axis_dout_tuser.(1)
s_axis_cartesian_tlast	In	tlast is not used by the core, but is combined with s_axis_phase_tlast, or passed untouched to m_axis_dout_tlast according to TLAST_Behavior.
s_axis_phase_tvalid	In	Handshake signal for channel S_AXIS_PHASE.(1)
s_axis_phase_tready	Out	Handshake signal for channel S_AXIS_PHASE.(1)
s_axis_phase_tdata[C-1:0]	In	This port has one subfield, PHASE_IN. It is the polar operand. The subfield is Input_Width bits wide, padded to the next byte width.
s_axis_phase_tuser[D-1:0]	In	Data on this port is delayed with the same latency as tdata and appear on m_axis_dout_tuser.(1)
s_axis_phase_tlast	In	tlast is not used by the core, but is combined with s_axis_cartesian_tlast, or passed untouched to m_axis_dout_tlast according to TLAST_Behavior.
m_axis_dout_tvalid	Out	Handshake signal for channel M_AXIS_DOUT. (1)
m_axis_dout_tready	In	Handshake signal for channel M_AXIS_DOUT.(1)
m_axis_dout_tdata[E-1:0]	Out	Depending on Functional Configuration this port contains the following subfields; X_OUT, Y_OUT, PHASE_OUT. Each subfield is Output_Width bits wide, padded to the next byte width before concatenation.
m_axis_dout_tuser[F-1:0]	Out	This port contains the values input to s_axis_cartesian_tuser and/or s_axis_phase_tuser delayed by the same latency as for tdata.
m_axis_dout_tlast	Out	This port outputs s_axis_cartesian_tlast, s_axis_phase_tlast or some combination of the two delayed by the same latency as for tdata.

Notes:

1. For AXI4-Stream details see Protocol Description–AXI-4 Stream.

Width constants A to F are arbitrary values, determined by the CORDIC Vivado IDE parameters. Many pins are optional. Input channels are absent if the function selected does not require the operands carried by the channel in question. For example, the Square Root function does not require PHASE_IN, so S_AXIS_PHASE is not present for this function.

Data Inputs and Outputs

The set of data input ports and output `tdata` subfields for a particular functional configuration are automatically determined by the Vivado IDE, as shown in Table 2-2.

Table 2-2: **Input/Output Subfields vs. Functional Configuration**

	S_AXIS_CARTESIAN		S_AXIS_PHASE	M_AXIS_DOUT		
Function	**XIN**	**YIN**	**PHASE_IN**	**XOUT**	**YOUT**	**PHASE_OUT**
Rotate	1	1	1	1	1	0
Translate	1	1	0	1	0	1
Sin and Cos	0	0	1	1	1	0
ArcTan	1	1	0	0	0	1
Sinh and Cosh	0	0	1	1	1	0
ArcTanh	1	1	0	0	0	1
Square Root	1	0	0	1	0	0

Notes:

1. A 1 indicates that the subfield and parent channel are present. A 0 indicates that the subfield is absent. If all subfields of a channel are absent, the channel is also absent. The X_IN operand, if present, is in the least significant bit positions of S_AXIS_CARTESIAN. SImilarly, X_OUT is in the least significant position of M_AXIS_DOUT, with Y_OUT in the next significant position and PHASE_OUT in the most significant position. Where one or more is missing, the remaining operands shift down in bit position. For example, for Translate with an Output_Width of 8, XOUT is [7:0] and PHASE_OUT is [15:8] of m_axis_dout_tdata.

Designing with the Core

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

Clocking

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

Resets

The CORDIC core uses a single, optional reset input called `ARESETn`. This signal is active-Low and must be active or inactive 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–AXI-4 Stream

The conversion to AXI4-Stream interfaces brings standardization and enhances interoperability of Xilinx® LogiCORE™ IP solutions. Other than general control signals such as `aclk`, `ACLKEN` and `ARESETn`, all inputs and outputs to the CORDIC core are conveyed using AXI4-Stream channels. A channel consists of `tvalid` and `tdata` always, plus several optional ports and fields. In the CORDIC core, 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 CORDIC core operates on the operands contained in the `tdata` fields and outputs the result in the `tdata` field of the output channel. The CORDIC core does not use inputs, `tuser` and `tlast` as such, but the core provides the facility to convey these fields with the same latency as for `tdata`. This facility of passing `tlast` and `tuser` from input to output is intended to ease use of the CORDIC core in a system. For example, the CORDIC core might operate on streaming packetized data. In this example, the core could be configured to pass the `tlast` of the packetized data channel, thus saving the system designer the effort of constructing a

bypass path for this information. For more information about AXI4-Stream Interfaces see [Ref 3] and [Ref 4].

Basic Handshake

Figure 3-1 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.

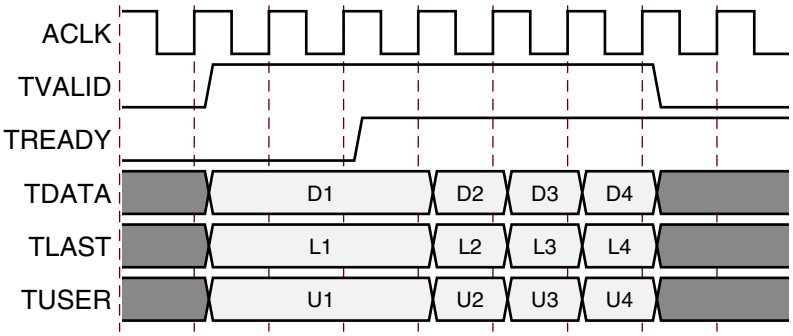

Figure 3-1: **Data Transfer in an AXI-Stream Channel**

Non Blocking Mode

The CORDIC core provides a mode intended to ease the migration from previous, non-AXI versions of this core. The term "NonBlocking" is used to indicate that lack of data on one input channel does not cause incoming data on the other channel to be buffered. Also, back pressure from the output is not possible because in NonBlocking mode the output channel does not have a `tready` signal. The full flow control of AXI4-Stream is not always required. Blocking or NonBlocking behavior is selected using the flow_control parameter or GUI field. 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 migrating their design to this version from a pre-AXI version with minimal change.

When all of the present input channels receive an active `tvalid` (and `tready`, if present, is asserted), an operation is validated and the output `tvalid` (suitably delayed by the latency of the core) is asserted to qualify the result. This is to allow a minimal migration from previous versions. If one channel receives `tvalid` and the other does not, an operation does not occur, even if `tready` is present and asserted. Unlike Blocking mode (which is fully AXI4-Stream compliant) valid transactions on an individual channel can be ignored in NonBlocking mode. For performance, `ARESETn` is registered internally, which delays its

action by one clock cycle. The effect is that the core is still reset and does not accept input in the cycle following the deassertion of ARESETn. tvalid is also inactive on the output channel for this cycle.

Figure 3-2 shows the NonBlocking mode in operation. For simplicity of illustration, the latency of the core is zero. As indicated by s_axis_cartesian_tready and s_axis_phase_tready (which are ultimately the same signal), the core can accept data on every third cycle. Data A1 in the Cartesian channel is ignored because s_axis_phase_tvalid is deasserted. Data inputs A2 and B1 are accepted because both tvalids and tready are asserted.

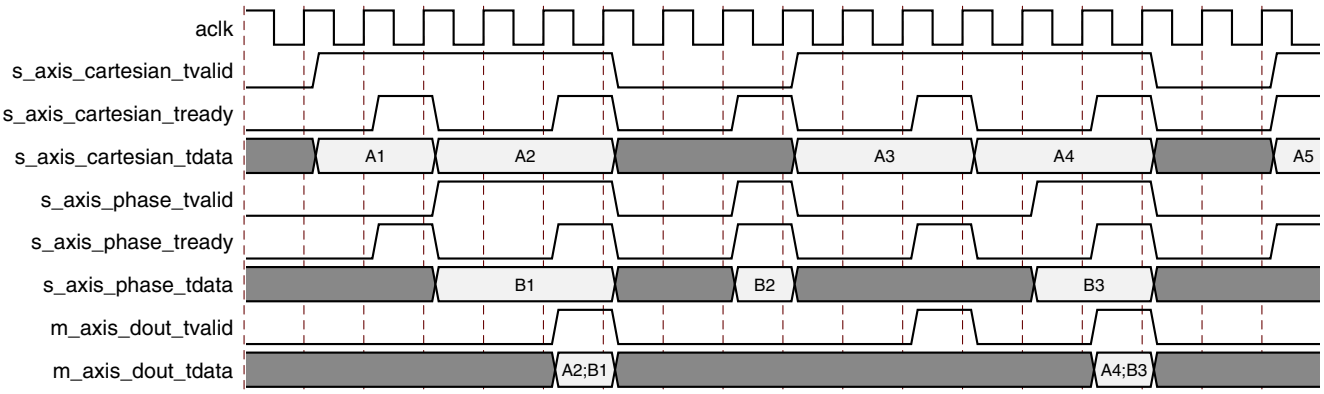

Figure 3-2: **NonBlocking Mode**

Blocking Mode

The term 'Blocking' means that each channel with tready buffers data for use. 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 flow_control 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 CORDIC core has one or two 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 m_axis_dout_tready is low, data accumulates in the output buffer internal to the core. When this output buffer is nearly full the core stops further operations. This prevents the input buffers from off-loading data for new operations so the input buffers fill as new data is input. When the input buffers fill, their respective treadys (s_axis_cartesian_tready and s_axis_phase_tready) are deasserted to prevent further input. This is the normal action of back pressure. The two input channels are tied, as each must receive validated data before an operation can proceed. As an additional blocking mechanism, one input channel does not receive validated data while the other does. 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 empty channel receives some data.

Figure 3-3 shows both blocking behavior and back pressure. The first data on channel S_AXIS_CARTESIAN is paired with the first data on channel S_AXIS_PHASE, the second with the second, and so on. This demonstrates the 'blocking' concept. The channel names S_AXIS_CARTESIAN and S_AXIS_PHASE are used conceptually. Either can be taken to mean the Cartesian or phase channel. Figure 3-3 further shows how data output is delayed not only by latency, but also by the handshake signal `m_axis_dout_tready`. This is 'back pressure'. Sustained back pressure on the output along with data availability on the inputs eventually leads to a saturation of the core buffers, causing 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 Figure 3-3 shows, 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.

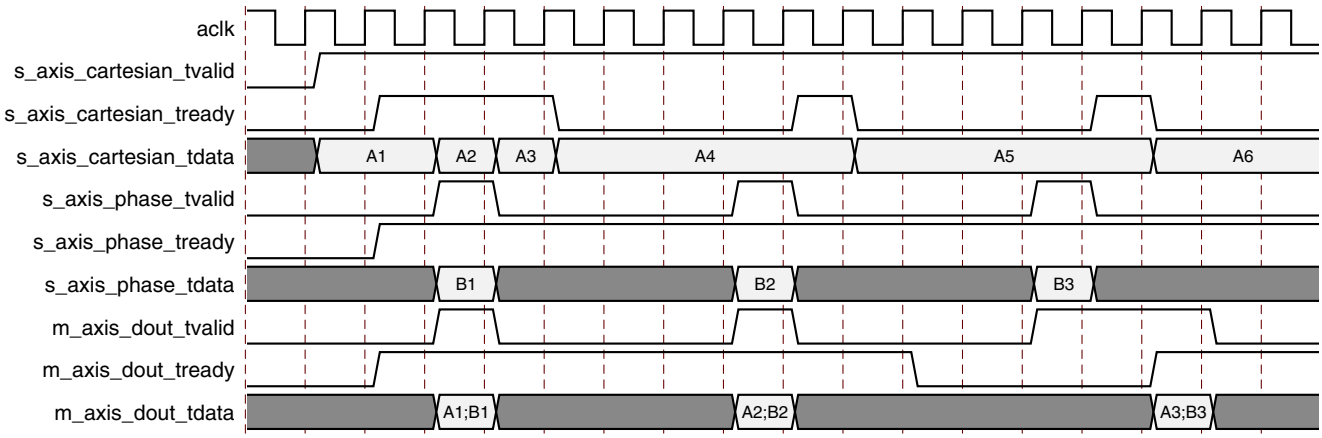

Figure 3-3: **Blocking Mode**

TDATA Packing

Fields within an AXI4-Stream interface follow a specific nomenclature. In this core the operands are both passed to or from the core over the `tdata` port of the channel. To ease interoperability with byte-oriented protocols, each subfield within `tdata` that could be used independently is first extended, if necessary, to fit a bit field which is a multiple of 8 bits. 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 use additional resources.

TDATA Structure for Cartesian Channel

Input channel `s_axis_cartesian` carries the two scalar operands X_IN (real) and Y_IN (imaginary) byte-aligned in the `tdata` field. Each operand occupies the least significant position in the bytes occupied. The `tdata` port width itself is the minimum number of bytes required to hold the operands (see Figure 3-4).

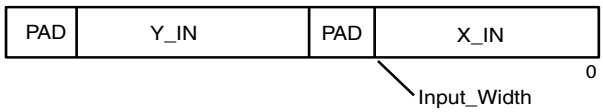

Figure 3-4: **TDATA Structure for Cartesian Channel**

TDATA Structure for Phase Channel

Input channel `s_axis_phase` carries the operand PHASE_IN in the `tdata` field. The `tdata` port width itself is the minimum multiple of bytes wide required to contain the operand (see Figure 3-5).

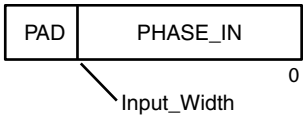

Figure 3-5: **TDATA Structure for Phase Channel**

TDATA Structure for Output (DOUT) Channel

The structure of `m_axis_dout_tdata` is more complex. This port can contain several combinations of output subfields X_OUT, Y_OUT and PHASE_OUT, depending on the Functional Selection parameter. The possible formats are shown with the corresponding functional selections in Figure 3-6.

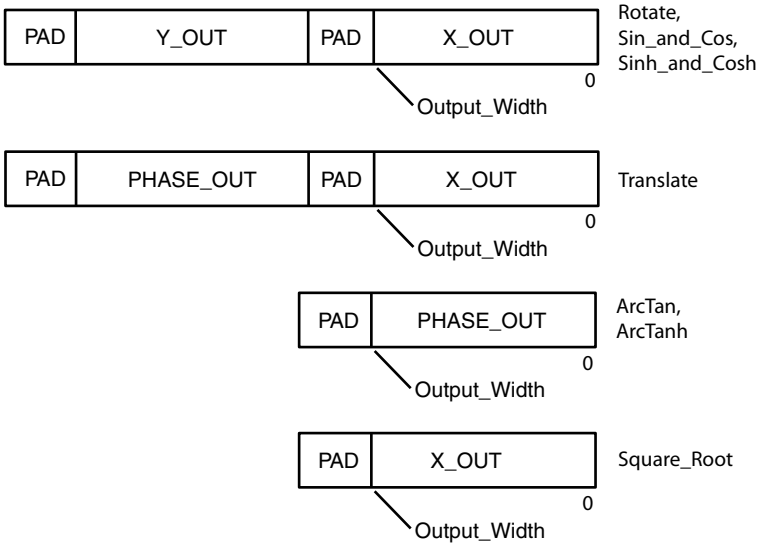

Figure 3-6: **TDATA Structure for Output (DOUT) Channel**

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 CORDIC core operates on a per-sample basis where each operation is independent. Because of this, there is no need for tlast on any of the input (s_axis) channels. The tlast and tuser signals are supported on each input channel as an optional aid to system design for implementations in which the data stream being passed through the CORDIC core has some packetization or ancillary field, but is not relevant to the CORDIC. The facility to pass tlast and/or tuser removes the burden of matching latency to the tdata path (which can be variable) through the CORDIC core.

TLAST Options

tlast for each input channel is optional. When present, each can be passed through the CORDIC. When more than one channel has tlast enabled, the core 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 the Cartesian channel occupies the least significant position, followed by the tuser field from the phase channel.

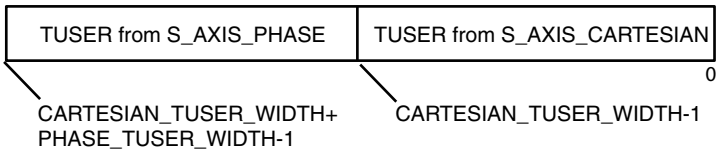

Figure 3-7: **TUSER for Cartesian and Phase Channel**

Functional Description

The CORDIC algorithm was initially designed to perform a vector rotation, where the vector (X,Y) is rotated through the angle θ yielding a new vector (X',Y').

Vector Rotation Equation

$$a) \quad X' = (\cos(\theta) \times X - \sin(\theta) \times Y) \qquad \textit{Equation 3-1}$$

$$b) \quad Y' = (\cos(\theta) \times Y + \sin(\theta) \times X)$$

$$c) \quad \theta' = 0$$

The CORDIC algorithm performs a vector rotation as a sequence of successively smaller rotations, each of angle atan(2^{-i}), known as *micro-rotations*. Equation 3-2 shows the expression for the i^{th} iteration where i is the iteration index from 0 to n.

Expression for the i^{th} microrotation

$$x_{i+1} = x_i - \alpha_i \cdot y_i \cdot 2^{-i} \quad \text{2a)} \qquad \textit{Equation 3-2}$$

$$y_{i+1} = y_i + \alpha_i \cdot x_i \cdot 2^{-i} \quad \text{2b)}$$

$$\theta_{i+1} = \theta_i + \alpha_i \cdot \text{atan}(2^{-i})) \quad \text{2c)}$$

α_i = (+ or -) 1, where α_i is the direction of rotation.

See CORDIC Scale Factor or Vector Translation for details on selecting α_i. Each micro-rotation stage can be expressed as a simple shift and add/subtract operation. Equation 3-3 shows the Vector rotation expression for the n^{th} iteration. Vector rotation expressed as a series of 'n' micro-rotations

$$X' = \prod_{i=1}^{n} \cos(\text{atan}(2^{-i}))(X_i - \alpha_i Y_i 2^{-i}) \quad \text{3a)} \qquad \textit{Equation 3-3}$$

$$Y' = \prod_{i=1}^{n} \cos(\text{atan}(2^{-i}))(Y_i + \alpha_i X_i 2^{-i}) \quad \text{3b)}$$

$$\theta' = \sum_{i=1}^{n} \theta - (\alpha_i \cdot \text{atan}(2^{-i})) \quad \text{3c)}$$

α_i = (+ or -) 1.

The CORDIC algorithm can be used to generate either a vector rotation or a vector translation.

Vector Rotation

Vector rotation rotates the vector (X, Y) through the angle θ to yield a new vector (X',Y'), as shown in Figure 3-8.

Vector rotation is performed by selecting α_i, such that θ' converges towards zero; that is, when θ_{i-1} >= 0, α_i is set to -1 and when θ_{i-1} < 0, α_i is set +1.

Vector Rotation Equations

$$X' = Z_i \times (\cos(\theta) \times X - \sin(\theta) \times Y) \quad \text{4a)} \qquad \textit{Equation 3-4}$$

$$Y' = Z_i \times (\cos(\theta) \times Y + \sin(\theta) \times X) \quad \text{4b)}$$

4c) $\theta' = 0$

$$Z_i = \frac{1}{\prod_{i=1}^{n} \cos(\tan(2^{-i}))}$$

Polar to Rectangular Translation

When the vector rotation functional configuration is selected, the input vector (X_IN, Y_IN) is rotated by the input angle, θ, using the CORDIC algorithm. This generates the scaled output vector, Z_i * (X', Y'), shown in Figure 3-8.

The input subfields, X_IN, Y_IN and PHASE_IN, are limited to the ranges given in Table 3-1 when coarse rotation is set. Inputs outside these ranges produce unpredictable results. See Input/Output Data Representation for more information about the CORDIC binary data formats.

An optional coarse rotation module is provided to extend the range of the input subfields, X_IN, Y_IN and PHASE_IN, to the full circle. For this functional configuration, the coarse rotation module is selected by default, but can be manually deselected. See Advanced Configuration Parameters for more information. When this option is not set, inputs must be constrained to lie in the first quadrant, -Pi/4 to + Pi/4.

An optional compensation scaling module is provided to compensate for the CORDIC scale factor Z_i. For this functional configuration, the compensation scaling module is selected by default, but can be manually deselected. See Advanced Configuration Parameters for more information.

A Polar to Rectangular Translation can be implemented by setting the functional configuration to vector rotation, the input vector to (Mag, 0), and the rotation angle to θ, shown in Figure 3-9.

Vector rotation is linear with respect to magnitude; thus the user can scale the input/output range; that is:

if (X, Y) rotated by angle θ = (X', Y') then
K*(X, Y) rotated by angle θ = K*(X', Y').

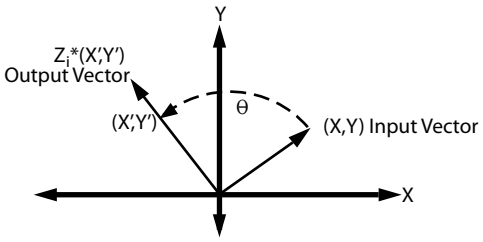

Figure 3-8: **Vector Rotation**

Table 3-1: **Vector Rotation I/O**

Signal	Range	Description
X_IN	-1 <= X_IN<=1	Input X Coordinate
Y_IN	-1 <= Y_IN<=1	Input Y Coordinate
PHASE_IN	-Pi <= PHASE_IN <= Pi	Input Rotation Angle
X_OUT	-Sqrt(2) <= X_OUT<= Sqrt(2)	Output X Coordinate * Z
Y_OUT	-Sqrt(2) <= Y_OUT<= Sqrt(2)	Output Y Coordinate * Z

Example 1: Vector Rotation

The input vector, (Xin, Yin), and the output vector, (Xout, Yout) are expressed as a pair of fixed-point twos complement numbers with an integer width of 2 bits (1QN format). The input rotation angle, Pin radians, is also expressed as a fixed-point twos complement number but with an integer width of 3 bits (2QN format). See the Input/Output Data Representation section for further information on the CORDIC binary data formats.

In this example, the input/output width is set to 10 bits and the output vector (Xout, Yout) is scaled to compensate for the CORDIC scale factor.

Xin: "0010110101" => 00.10110101 => 0.707

Yin: "0001000000" => 00.01000000 => 0.25

Pin: "1100110111" => 110.0110111 => -Pi/2

Xout: "0001000001" => 00.01000001 => 0.25

Yout: "1101001011" => 11.01001011 => -0.707

Vector Translation

Vector translation rotates the vector (X_IN,Y_IN) around the circle until the Y component equals zero as illustrated in Figure 3-9. The outputs from vector translation are the magnitude, X', and phase, θ', of the input vector (X,Y).

Vector translation is performed by selecting α_i such that Y' converges towards zero; that is, when $Y_{i-1} >= 0$, α_i is set to -1 and when $Y_{i-1} < 0$, α_i is set +1.

Vector Translation Equations:

$$5a) \quad X' = Z_i \times \sqrt{(X^2 + Y^2)} \qquad \textit{Equation 3-5}$$

$$5b) \quad Y' = 0$$

$$5c) \quad \theta' = \text{atan}\left(\frac{X}{Y}\right)$$

$$Z_i = \frac{1}{\prod_{i=1}^{n} \text{acos}(\text{atan}(2^{-i}))}$$

Rectangular to Polar Translation

When the vector translational functional configuration is selected, the input vector (X_IN,Y_IN) is rotated using the CORDIC algorithm until the Y component is zero. This generates the scaled output magnitude, Z_i * Mag(X_IN,Y_IN), and the output phase, Atan(Y_IN/X_IN), shown in Figure 3-9.

The inputs, X_IN and Y_IN, are limited to the ranges given in Table 3-2 when coarse rotation is set. Inputs outside these ranges produce unpredictable results. See Input/Output Data Representation for more information about CORDIC binary data formats.

An optional coarse rotation module is provided to extend the range of inputs, X_IN and Y_IN, to the full circle. For this functional configuration, the coarse rotation module is selected by default, but can be manually deselected. See Advanced Configuration Parameters for more information. When this option is not set, inputs must be constrained to lie in the first quadrant, -Pi/4 to + Pi/4.

An optional compensation scaling module is provided to compensate for the CORDIC scale factor Z_i. For this functional configuration, the compensation scaling module is selected by default, but can be manually deselected. See Advanced Configuration Parameters for more information.

A rectangular to polar translation can be implemented by setting functional configuration to vector translation, and the input vector to (X,Y), shown in Figure 3-9.

Vector translation is linear with respect to magnitude, making the input/output range scalable:

> if vector (X_IN, Y_IN) is translated to (X', θ'), then
> vector K*(X_IN, Y_IN) is translated to K*(X', θ').

The phase angle of a zero length vector, (0,0), is indeterminate and the output phase angle generated by the core is unpredictable.

The accuracy of the phase output from the CORDIC vector translation algorithm is limited by the number of significant magnitude bits of the input vector (X_IN, Y_IN). See Customizing and Generating the Core for more information.

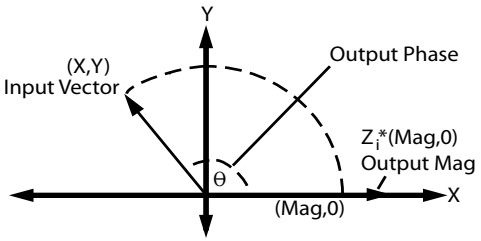

Figure 3-9: **Vector Translation (Polar to Rectangular)**

Example 2: Vector Translation

Table 3-2: **Vector Translation I/O**

Signal	Range	Description
X_IN	-1 <= X_IN <= 1	Input X Coordinate
Y_IN	-1 <= Y_IN <= 1	Input Y Coordinate
X_OUT	0 <= X_OUT <= Sqrt(2)	Output Magnitude * Z
PHASE_OUT	-Pi <= Phase Out <= Pi	Output Phase

The individual input vector elements, (X_IN, Y_IN), and the output magnitude, X_OUT, are expressed as fixed-point twos complement numbers with an integer width of 2 bits (1QN format). The output phase angle, PHASE_OUT radians, is expressed as a fixed-point twos complement number with an integer width of 3 bits (2QN format).

In this example the input/output width is set to 10 bits and the output X_OUT is scaled to compensate for the CORDIC scale factor.

X_IN: "0010110101" => 00.10110101 => 0.707

Y_IN: "0001000000" => 00.01000000 => 0.25

X_OUT: "0011000000" => 00.11000000 => 0.75

PHASE_OUT: "0000101011" => 000.0101011 => 0.336

CORDIC Scale Factor

The outputs of the CORDIC algorithm, Equation 3-4 and Equation 3-5, are equivalent to a vector rotation or vector translation scaled by a constant Z_i. The constant Z_i is known as the CORDIC scale factor.

$$Z_i = \frac{1}{\prod_{i=1}^{n} \cos(\operatorname{atan}(2^{-i}))}$$ *Equation 3-6*

The Taylor series expansion of acos (atan (2^{-i})) is $(1 + 2^{-2i})^{-1/2}$. Hence, the constant Z_i can be expressed as

$$Z_i = \prod_{i=1}^{n} (1 + 2^{-2i})^{1/2} \qquad \textit{Equation 3-7}$$

The CORDIC scale factor, Z_i, is only dependent on the number of iterations, n. Only functional configurations Rotate, Translate, Rectangular to Polar, and Polar to Rectangular are affected by the CORDIC scale factor. When these functional configurations are selected, the CORDIC core provides the option of multiplying by 1 / Z_i to cancel out the scaling factor. See Advanced Configuration Parameters for more information.

Output Quantization Error

The Output Quantization Error can be split into two components; the Output Quantization Error due to the Input Quantization (OQEIQ), and the Output Quantization Error due to Internal Precision (OQEIP).

OQEIQ is due to the half LSB of quantization noise on the X_IN,Y_IN and PHASE_IN inputs. In a vector rotation this input quantization noise results in OQEIQ of a half LSB on both the X_OUT and Y_OUT outputs. In a vector translation this input quantization noise results in OQEIQ of a half LSB on the X_OUT output; however, OQEIQ on the phase output is dependent on the ratio (Y_IN/ X_IN). Thus for small X_IN inputs the effect of input quantization noise on OQEIQ is greatly magnified.

OQEIP is due to the limited precision of internal calculations. In the CORDIC core the default internal precision is set such that the accumulated OQEIP is less than 1/2 the OQEIQ. The internal precision can be manually set to (Input_Width + Output_Width + $\log_2$(Output_Width)). This reduces OQEIP to a half LSB (the phase is calculated to full precision regardless of the magnitude input vector).

The Output Quantization Error, for a CORDIC core with default internal precision, is dominated by OQEIQ. OQEIQ can only be reduced by increasing the number of significant magnitude bits in the input vector (X_IN,Y_IN). Increasing the internal precision or zero padding X_IN and Y_IN inputs only affects OQEIP and has minimal effect on the total output quantization error.

The effect of input quantization and internal quantization on the CORDIC phase output quantization error is illustrated in the following examples:

Example 1a: The quantization error in phase output for a small input vector, (Xin_small, Yin_small).

Xin_small: “0000000001” => 1/256.

Yin_small: “0000000001” => 1/256.

Vector translation with no input quantization:

Xin_ideal: "0000000001" => 1/256.

Yin_ideal: "0000000001" => 1/256.

Pout_ideal: "0001100100" => 0.79.

Output quantization error due to the input quantization:

Xin_Quant = Xin_small - 1/2 LSB and Yin_Quant = Yin_small + 1/2 LSB.

Xin_Quant: "00000000001" => 1/512.

Yin_Quant: "00000000011" => 3/512.

Pout_Quant: "0010100000" =>1.25.

OQEIQ = abs(abs(Pout_Quant) - abs(Pout_Ideal)).

OQEIQ = "0000111100" => 0.47.

Output quantization error due to the internal precision:

Xin_cordic: "0000000001" => 1/256.

Yin_cordic: "0000000001" => 1/256.

Pout_cordic: "0001111010" => 0.95.

OQEIP = abs(abs(Pout_cordic) - abs(Pout_Ideal)).

OQEIP = "0000010110" => 0.17.

Example 1b: Quantization error in phase output for a large input vector, (Xin_large, Yin_large).

Xin_large: "0100000000" => 256/256.

Yin_large: "0100000000" => 256/256.

Vector translation with no input quantization:

Xin_ideal: "0100000000" => 256/256.

Yin_ideal: "0100000000" => 256/256.

Pout_ideal: "0001100100" => 0.79.

Output quantization error due to the input quantization:

Xin_Quant = Xin_large - 1/2 LSB and Yin_Quant = Yin_small + 1/2 LSB.

Xin_Quant: "00111111111" => 511/512.

Yin_Quant: "01000000001" => 513/512.

Pout_Quant: "0001100101" =>0.79.

OQEIQ = abs(abs(Pout_Quant) - abs(Pout_Ideal)).

OQEIQ = "0000000001" => 0.00.

Output quantization error due to the internal precision:

Xin_cordic: "0100000000" => 256/256.

Yin_cordic: "0100000000" => 256/256.

Pout_cordic: "0001100100" => 0.79.

OQEIP = abs(abs(Pout_cordic) - abs(Pout_Ideal)).

OQEIP = "0000000000" => 0.00

Sin and Cos

When the Sin and Cos functional configuration is selected, the unit vector is rotated by input angle, θ, using the CORDIC algorithm. This generates the output vector (Cos(θ), Sin(θ)).

The input PHASE_IN is limited to the range given in Table 3-3 when coarse rotation is set. Inputs outside this range produce unpredictable results. See Input/Output Data Representation for more information about CORDIC binary data formats.

An optional coarse rotation module is provided to extend the range of input angle, θ, to the full circle. For this functional configuration, the coarse rotation module is selected by default, but can be manually deselected. See Advanced Configuration Parameters for more information. When this option is not set, inputs must be constrained to lie in the first quadrant, -Pi/4 to + Pi/4.

The compensation scaling module is disabled for the Sin and Cos functional configuration as it is internally pre-scaled to compensate for the CORDIC scale factor.

Table 3-3: **Sin and Cos**

Signal	Range	Description
PHASE_IN	-Pi <= PHASE_IN <= Pi	Input Angle θ
X_OUT	-1 <= X_OUT <= 1	Output Cos(θ)
Y_OUT	-1 <= Y_OUT <= 1	Output Sin(θ)

Example 3: Sin and Cos

The input angle, PHASE_IN, is expressed as a fixed-point twos complement number with an integer width of 3 bits (2QN format). The output vector, (X_OUT, Y_OUT), is expressed as a pair of fixed-point twos complement numbers with an integer width of 2 bits (1QN format).

In this example the input/output width is set to 10 bits.

PHASE_IN: "0001100100" => 000.1100100 => 0.781

X_OUT: "0010110110" => 00.10110110 => 0.711

Y_OUT: "0010110100" => 00.10110100 => 0.703

Sinh and Cosh

When the Sinh Cosh functional configuration is selected, the CORDIC algorithm is used to move the vector (1,0) through hyperbolic *angle*, p, along the hyperbolic curve shown in Figure 3-10. The hyperbolic angle represents the log of the area under the vector (X, Y) and is unrelated to a trigonometric angle. This generates the output vector (X_OUT = Cosh(PHASE_IN), Y_OUT = Sinh(PHASE_IN)).

The input hyperbolic angle, PHASE_IN, is limited to the range given in Table 3-4. Inputs outside this range produce unpredictable results. See Input/Output Data Representation for more information about CORDIC binary data formats.

The coarse rotation module is disabled for the Sinh and Cosh functional configuration, as it does not apply to hyperbolic transformations. The compensation scaling module is disabled for the Sinh and Cosh functional configuration, as it is internally pre-scaled to compensate for the CORDIC hyperbolic scale factor.

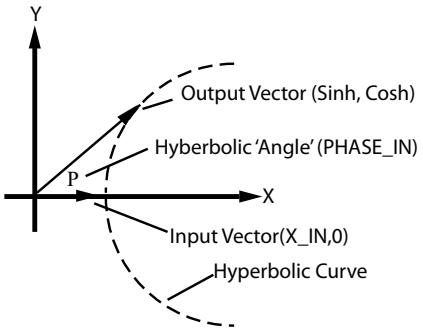

Figure 3-10: **Hyperbolic Sinh Cosh**

Table 3-4: **Sinh and Cosh**

Signal	Range	Description
PHASE_IN	-Pi/4 <= PHASE_IN <= Pi/4	Input Hyperbolic Angle
X_OUT	1 <= X_OUT < 2	Output Cosh
Y_OUT	-2 <= Y_OUT < 2	Output Sinh

Example 4: Sinh and Cosh

The input hyperbolic angle, Pin, is expressed as a fixed-point twos complement number with an integer width of 3 bits (2QN format). The output vector, (X_OUT, Y_OUT), is expressed as a pair of fixed-point twos complement numbers with an integer width of 2 bits (1QN format).

In this example the input/output width is set to 10 bits.

PHASE_IN: "0001001110" => 000.1001110 => 0.609

X_OUT: "0100110001" => 01.00110001 => 1.191

Y_OUT: "0010100110" => 00.10100110 => 0.648

ArcTan

When the ArcTan functional configuration is selected, the input vector (X_IN,Y_IN) is rotated (using the CORDIC algorithm) until the Y component is zero. This generates the output angle, Atan(Y_IN/X_IN).

The inputs, X_IN and Y_IN, are limited to the ranges given in Table 3-5 when coarse rotation is set. Inputs outside these ranges produce unpredictable outputs. See Input/Output Data Representation for more information about CORDIC binary data formats.

An optional coarse rotation module is provided to extend the range of inputs X_IN and Y_IN to the full circle. For this functional configuration, the coarse rotation module is selected by default, but can be manually deselected. See Advanced Configuration Parameters for more information. When this option is not set, inputs must be constrained to lie in the first quadrant, -Pi/4 to + Pi/4.

The compensation scaling module is disabled for the ArcTan functional configuration as no magnitude data is output. The ArcTan of a zero length vector, (0,0), is indeterminate and the output angle generated by the core is undefined.

The accuracy of the output angle from the CORDIC vector translation algorithm is limited by the number of significant magnitude bits of the input vector (X_IN, Y_IN). See Output Quantization Error for more information.

Table 3-5: **ArcTan**

Signal	Range	Description
X_IN	-1 <= X_IN <=1	Input X Coordinate
Y_IN	-1 <= Y_IN <=1	Input Y Coordinate
PHASE_OUT	-Pi <= PHASE_OUT <= Pi	Output Angle

Example 5: ArcTan

The input vector (X_IN, Y_IN) is expressed as a pair of fixed-point twos complement numbers with an integer width of 2 bits (1QN format). The output angle, Pout radians, is expressed as a fixed-point twos complement number with an integer width of 3 bits (2QN format).

In this example, the input/output width is set to 10 bits.

X_IN: "0010100000" => 00.10100000 => 0.625

Y_IN: "0010000000" => 00.10000000 => 0.500

PHASE_OUT: "0001010110" => 000.1010110=> 0.672

ArcTanh

When the ArcTanh functional configuration is selected, the CORDIC algorithm is used to move the input vector (X_IN,Y_IN) along the hyperbolic curve (Figure 3-11) until the Y component reaches zero. This generates the hyperbolic "angle," Atanh(Y_IN/X_IN). The hyperbolic *angle* represents the log of the area under the vector (X_IN,Y_IN) and is unrelated to a trigonometric angle.

The inputs, X_IN and Y_IN, are limited to the ranges given in Table 3-6. Inputs outside these ranges produce unpredictable outputs. Additionally, Y_IN must be less than or equal to (4/5 * X_IN) or the CORDIC algorithm does not converge. See Input/Output Data Representation for more information about CORDIC binary data formats.

The coarse rotation module is disabled for the ArcTanh functional configuration, as it does not apply to hyperbolic transformations.

The compensation scaling module is disabled for the ArcTanh functional configuration as no output magnitude data is output.

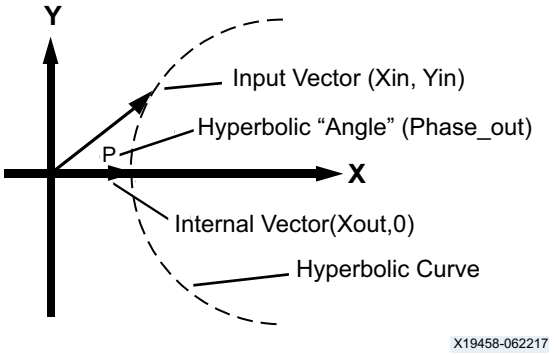

Figure 3-11: **Hyperbolic ArcTan**

Table 3-6: **ArcTanh**

Signal	Range	Description
X_IN	0 < X_IN <2	Input X Coordinate
Y_IN	-2 <= Y_IN < 2 -X_IN * 4/5 <= Y_IN <= X_IN * 4/5	Input Y Coordinate
PHASE_OUT	-Pi/2 <= PHASE_OUT<= Pi/2	Output Hyperbolic Angle

Example 6: ArcTanh

The input vector, (X_IN, Y_IN), is expressed as a pair of fixed-point twos complement numbers with an integer width of 2 bits (1QN format). The output, Pout, is expressed as a fixed-point twos complement number with an integer width of 3 bits (2QN format).

In this example, the input/output width is set to 10 bits.

X_IN: “0111111111” => 01.11111111 => 0.998

Y_IN: “0101111011” => 01.01111011 => 0.740

PHASE_OUT: “0001111010” => 000.1111010 => 0.953

Square Root

When the square root functional configuration is selected, a simplified CORDIC algorithm is used to calculate the positive square root of the input. The input, X_IN, and the output, X_OUT, are always positive and are both expressed as either unsigned fractions or unsigned integers. When the data format is set to Unsigned Fraction, X_IN is limited to the range: 0 <= X_IN < +2. When data format is set to Unsigned Integer, X_IN is limited to the range: 0 <= X_IN < 2**Input_Width, and the output width is determined automatically based on the input width. See Input/Output Data Representation for more information about CORDIC binary data formats.

The coarse rotation module is disabled because coarse rotation is not required for the Square Root functional configuration. The compensation scaling module is disabled because no output compensation is required for the Square Root functional configuration.

Table 3-7: **Square Root**

Signal	Range	Description
X_IN	Unsigned Fraction: $0 \le X\_IN < +2$ Unsigned Integer: $0 \le X\_IN < 2^{Input\_Width}$	Input X Value
X_OUT	Unsigned Fraction: $0 \le X\_OUT < +2$ Unsigned Integer: $0 \le X\_OUT < 2^{[int(Input\_Width/2) + 1]}$	Output Square Root

Example 7a: Square Root - Unsigned Fraction

The input, X_IN, and output, X_OUT, are expressed as an unsigned fixed-point number with an integer width of 1 bit.

In this example the input/output width is set to 10 bits.

X_IN: “0000100000” => 0.000100000 => 1/16

X_OUT: “0010000000” => 0.010000000 => 1/4

Example 7b: Square Root - Unsigned Integer

The input, X_IN, is expressed as an unsigned integer. The output, X_OUT, is expressed as an unsigned integer. In this example the input width is set to 10 bits so the output width is automatically set to 6 bits.

X_IN: “0000100000” => 32

X_OUT: “000110” => 6

Input/Output Data Representation

The CORDIC algorithm is used for a variety of operations. Depending on the operation in question, the input and output can be in Cartesian pair, Polar pair or Scalar form. The following sections describe the representation of values in these various forms.

Cartesian Operands and Results

The `s_axis_cartesian_tdata` subfields are: X_IN, Y_IN. The `m_axis_dout_tdata` subfields are X_OUT and Y_OUT.

For Functional Configurations, Rotate, Translate, Sin, Cos and Atan, the Cartesian operands and results are represented using fixed-point twos complement numbers with an integer width of 2 bits. The integer width is fixed regardless of the word width; the remainder of the bits are used for the fractional portion of the number. Using the Q Numbers Format this representation is described as 1QN where N = word width - 2. It can also be described as Fix(N+2)_N using the System Generator Fix format.

Input operands, X_IN and Y_IN, must be in the range: -1 <= input data signal <= 1. Input data outside this range produces undefined results.

Using a 10-bit word width, +1 and -1 are represented as:

"0100000000" => 01.00000000 => +1.0

"1100000000" => 11.00000000 => - 1.0

For the Square Root Functional Configuration, the Data Signals, X_IN and X_OUT, are both represented in either Unsigned Fractional or Unsigned Integer data format.

The input operand, X_IN, must be in the range: 0 <= X_IN < +2 when data format is set to Unsigned Fraction or in the range 0 <= X_IN < 2**Input_Width when data format is set to Unsigned Integer.

When Unsigned Fractional data format has been selected the Data Signals are represented using a unsigned fixed-point number with an integer with of 1 bit. The integer width is fixed and the remainder of the word is used to represent the fractional portion of the number. Using the System Generator Fix format this representation is described as UFix(N+1)_N, where is the number of fractional bits being used and is defined as N = word width -1. The Q Number format is used to represent signed twos complement numbers and is therefore not suitable to describe the representation format used by the square root function.

Phase Signals

The `s_axis_phase_tdata` Phase operand is PHASE_IN. The `m_axis_dout_tdata` phase output is called PHASE_OUT. The phase signals are always represented using a fixed-point twos complement number with an integer width of 3 bits. As with the data signals the integer width is fixed and any remaining bits are used for the fractional portion of the number. The Phase Signals require an increased integer width to accommodate the increased range of values they must represent when the Phase Format is set to Radians.

When Phase Format is set to Radians, PHASE_IN must be in the range: -Pi <= (PHASE_IN) <= Pi. PHASE_IN outside this range produce undefined results.

In 2Q8, or Fix11_7, format values, +Pi and -Pi are:

"01100100100" => 011.00100100 => +3.14

"10011011100" => 100.11011100 => - 3.14

When Phase Format is set to Scaled Radians PHASE_IN must be in the range: -1 <= (PHASE_IN) <= +1. PHASE_IN outside this range produce undefined results.

In 2Q7, or Fix10_7 format values, +1 and -1 are represented as:

"0010000000" => 001.0000000 => +1.0

"1110000000" => 111.0000000 => - 1.0

Q Numbers Format

An XQN format number is an 1+X+N bit twos complement binary number; a sign bit followed by X integer bits followed by an N bit mantissa (fraction). XQN format can be used to express numbers in the range (-2^X) to ($2^X - 2^{(-N)}$). An equivalent notation using the System Generator Fix format, defined as Fix***word_length_fractional_length***, would be Fix(1+X+N)_N.

A number using Q15 format is equivalent to a number using Fix16_15 representation, and a number in 1Q15 format is equivalent to a number using Fix17_15 representation.

Table 3-8 and Table 3-9 contain examples of XQN Format Numbers.

Table 3-8: **1QN Format Data: Example of a 1Q7 (or Fix9_7) Format Number**

	(Sign) Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
+1	0	1	0	0	0	0	0	0	0
-1	1	1	0	0	0	0	0	0	0
+Pi/4	0	0	1	1	0	0	1	0	0
-Pi/4	1	1	0	0	1	1	0	1	1
			Fractional Bits						

Table 3-9: **2QN Format Phase: Example of a 2Q6 (or Fix9_6) Format Number**

	(Sign) Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	BIt 0
+1	0	0	1	0	0	0	0	0	0
-1	1	1	1	0	0	0	0	0	0
+Pi	0	1	1	0	0	1	0	0	1
-Pi	1	0	0	1	1	0	1	1	1
				Fractional Bits					

Mapping Different Data Formats

Rotate, Translate, Sin, Cos and Atan Functional Configurations

For functional configurations, Rotate, Translate, Sin, Cos and Atan it is possible to map alternative Data Signal formats to the fixed integer width fractional number used by the CORDIC core.

When the input and output width differ, care must be taken to re-interpret the CORDIC output.

Example 8a develops Example 2: Vector Translation to demonstrate a possible remapping.

Example 8a

The Vector Translation function determines the magnitude and phase angle of a given input vector (X_IN, Y_IN). The input and output width is set to 10 bits. The standard CORDIC data representation is Fix10_8, the alternative format being mapped onto the input of the CORDIC is Fix10_1.

X_IN value: "0010110101"

Table 3-10: **Example 8: Mapping an Alternative Data Format onto the X_IN input**

	Sign Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Decimal Value
Binary Value	0	0	1	0	1	1	0	1	0	1	
Fix10_8 weighting	-2^{1}	2^{0}	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}	0.707
Fix10_1 weighting	-2^{8}	2^{7}	2^{6}	2^{5}	2^{4}	2^{3}	2^{2}	2^{1}	2^{0}	2^{-1}	90.5

Y_IN value: "0001000000"

Table 3-11: **Example 8: Mapping an Alternative Data Format onto the Y_IN input**

	Sign Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Decimal Value
Binary Value	0	0	1	0	0	0	0	0	0	0	
Fix10_8 weighting	-2^{1}	2^{0}	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}	0.25
Fix10_1 weighting	-2^{8}	2^{7}	2^{6}	2^{5}	2^{4}	2^{3}	2^{2}	2^{1}	2^{0}	2^{-1}	32

MATLAB® software is used to generate the expected results. Firstly the magnitude and phase angle for the standard CORDIC input format 1Q8, or Fix10_8 is generated:

```
>> a=0.707+0.25j

>> magnitude = abs(a)

magnitude = 0.7499
```

>> phase_angle = angle(a)

phase_angle = **0.3399**

Secondly using the mapped input format, 9Q1 or Fix10_1:

>> b=90.5+32j

>> magnitude = abs(b)

magnitude = **95.9909**

>> phase_angle = angle(b)

phase_angle = **0.3399**

The CORDIC output is:

X_OUT value: "0011000000"

PHASE_OUT value: "0000101011"

Table 3-12 and Table 3-13 demonstrate the output value of the CORDIC being interpreted using the two data representation formats.

Table 3-12: **Example 8: X_OUT Interpretation**

	Sign Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Decimal Value
Binary Value	0	0	1	1	0	0	0	0	0	0	
Fix10_8 weighting	-2^{1}	2^{0}	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}	0.75
Fix10_1 weighting	-2^{8}	2^{7}	2^{6}	2^{5}	2^{4}	2^{3}	2^{2}	2^{1}	2^{0}	2^{-1}	96

Table 3-13: **Example 8: PHASE_OUT Interpretation**

	Sign Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Decimal Value
Binary Value	0	0	0	0	1	0	1	0	1	1	
Fix10_7 weighting	-2^{2}	2^{1}	2^{0}	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}	0.336

Example 8b

If the output width is less than the input width, the CORDIC reduces the fractional width of the result. When the data output, X_OUT, is being re-interpreted to an alternative data format, the value must be scaled appropriately.

Table 3-14 demonstrates how the resulting decimal value might change when the output width is reduced to 8 bits.

Table 3-14: **Example 8b: X_OUT Interpretation with Reduced Output Width**

	Sign Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Decimal Value
Binary Value	0	0	1	1	0	0	0	0	
Fix8_6 weighting	-2^{1}	2^{0}	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	0.75
Fix8_0 weighting	-2^{7}	2^{6}	2^{5}	2^{4}	2^{3}	2^{2}	2^{1}	2^{0}	48

A similar situation arises when the output width is greater than the input width. In this case, the CORDIC increases the fractional width of the result. When the data output is being re-interpreted to a data format with no fractional bits, this results in an increased magnitude. This output then needs to be scaled appropriately.

Square Root Functional Configuration

For the Square Root functional configuration it is also possible to map other data formats onto the data format of the CORDIC but it might be necessary to re-interpret and scale the output.

Example 9 modifies Example 7a: Square Root - Unsigned Fraction.

Example 9

X_IN value: "00001000"

Table 3-15: **Example 9: Mapping an Alternative Data Format onto the X_IN input**

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Decimal Value
Binary Value	0	0	0	0	1	0	0	0	
UFix8_7 weighting	2^{0}	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}	0.0625
UFix8_1 weighting	2^{6}	2^{5}	2^{4}	2^{3}	2^{2}	2^{1}	2^{0}	2^{-1}	4
UFix8_0 weighting	2^{7}	2^{6}	2^{5}	2^{4}	2^{3}	2^{2}	2^{1}	2^{0}	8

The expected output values for each input format are as follows:

UFix8_7 format: sqrt(0.0625) = **0.25**

UFix8_1 format: sqrt(4) = **2**

UFix8_0 format: sqrt(8) = **2.8284**

The CORDIC output is:

X_OUT value: "00100000"

Table 3-16 demonstrates the output value directly interpreted in each of the input formats.

Table 3-16: **X_OUT Direct Interpretation**

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Decimal Value
Binary Value	0	0	1	0	0	0	0	0	
UFix8_7 weighting	2^{0}	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}	0.25
UFix8_1 weighting	2^{6}	2^{5}	2^{4}	2^{3}	2^{2}	2^{1}	2^{0}	2^{-1}	16
UFix8_0 weighting	2^{7}	2^{6}	2^{5}	2^{4}	2^{3}	2^{2}	2^{1}	2^{0}	32

Table 3-16 shows that if the output value is directly interpreted in the alternative data format the wrong decimal value is determined. The output value must be scaled correctly.

The output scaling is determined as follows.

The CORDIC core calculates the square root of input values in the range 0 <= X_IN < 2.

$$Y = \sqrt{X} \qquad \textit{Equation 3-8}$$

The alternative data format represents values in the range 0<= X_IN < 2^{N+1} and the requirement is to calculate:

$$Y_{alt} = \sqrt{X_{alt}} \qquad \textit{Equation 3-9}$$

Interpreting X_{alt} using the standard CORDIC data format scales the input by 2^{-N}, shown in Table 3-15.

$$Y = \sqrt{2^{-N} \cdot X_{alt}}$$
$$Y = 2^{(-N)/2} \cdot \sqrt{X_{alt}} \qquad \textit{Equation 3-10}$$

As Table 3-16 shows, directly re-interpreting the CORDIC output in the alternative data formats results in an incorrect decimal value. This is due to the scale factor introduced by the remapping of the input and the square root function. This scaling factor introduced is shown in Equation 3-10, $2^{-N/2}$.

The corrected results are shown:

UFix8_1 weighting: $16/2^{(6/2)}$ = **2**

UFix8_0 weighting: $32/2^{(7/2)}$ = **2.8284**

When N is even the scaling factor is an integer power of two. This can be applied by right shifting the CORDIC output, X_OUT, by N/2. The example using the UFix8_1 format demonstrates this with a scaling factor of 2^{-3} = 1/8.

When N is odd the scaling factor is not an integer power of two. This introduces an additional output scaling factor of $\sqrt{2}$. The example using UFix8_0 demonstrates this with a scaling factor of $2^{-7/2} = 2^{-3.5}$.

This could be implemented by first scaling the output by a right shift of 4 and then multiplying by $\sqrt{2}$. A more efficient way would be to translate the $\sqrt{2}$ scaling to the input of the square root function.

This is demonstrated in Equation 3-11 where $2^{-N/2}=2^{-M-(1/2)}$.

$$Y = 2^{(-M-1/2)} \cdot \sqrt{X_{alt}}$$
$$Y = 2^{-M} \cdot \sqrt{2^{-1} \cdot X_{alt}}$$

Equation 3-11

The scaling becomes a simple divide by 2, or right shift, of the input, X_IN, before applying it to the square root function. Followed by scaling the output, X_OUT, by 2^{-M}.

An input value of 8 is used for the UFix8_0 formatting example. Divided by 2 this gives 4. Table 3-15 shows that 4 maps to 1/32 in the CORDIC input range.

$$\sqrt{1/32} = 0.17678 = 0.0010110$$

Table 3-16 shows that the CORDIC output value, 0.0010110, maps to a decimal value of 22 in UFix8_0 formatting. Applying the output scaling of 2^{-3}, or 1/8, gives **2.75**. The loss in accuracy is due to representing $\sqrt{1/32}$ using only 8 bits. If the full accuracy result is used and then re-interpreted to the alternative data format (Fix8_0) and then scaled, the correct result is obtained; for example:

$$\sqrt{1/32} \times 2^{7} \times 2^{-3} = 2.8284$$

Design Flow Steps

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

- *Vivado Design Suite User Guide: Designing IP Subsystems using IP Integrator* (UG994) [Ref 5]
- *Vivado Design Suite User Guide: Designing with IP* (UG896) [Ref 6]
- *Vivado Design Suite User Guide: Getting Started* (UG910) [Ref 7]
- *Vivado Design Suite User Guide: Logic Simulation* (UG900) [Ref 8]

Customizing and Generating the Core

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

If you are customizing and generating the core in the Vivado IP integrator, see the *Vivado Design Suite User Guide: Designing IP Subsystems using IP Integrator* (UG994) [Ref 5] for detailed information. IP integrator might auto-compute certain configuration values when validating or generating the design. To view the parameter value you can run the `validate_bd_design` command in the Tcl console.

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

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

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

The CORDIC GUI in the Vivado IDE contains two information tabs and two pages to configure the core.

Tab 1 & 2: IP Symbol and Implementation Details

The IP Symbol tab displays the core pinout.

The Implementation Details tab displays the core latency and resource usage. The block RAM and Multiplier/DSP Slice resources are only utilized when Compensation Scaling is selected.

Component Name: Used as the base name of the output files generated for the core. Names must begin with a letter and be composed from the following characters: a to z, 0 to 9, and "_."

Page 1 - Configuration Options

Used to configure the functional selection and architecture of the CORDIC core.

- **Functional Selection:** The functional selections available are Rotate, Sin and Cos, ArcTan, Square Root, Translate, Sinh and Cosh and ArcTanh. See Functional Description for more information on each of the supported functions. In general, X_IN, Y_IN, X_OUT and Y_OUT express signed binary numbers of 1QN format and PHASE_IN and PHASE_OUT express signed binary numbers of 2QN format. When Square Root is selected, two new data formats are available: Unsigned Integer and Unsigned Fraction. For details about CORDIC binary data formats, see Input/Output Data Representation.
- **Architectural Configuration:** Two architectural configurations are available for the CORDIC core, Parallel and Word Serial. See Performance for more details.
- **Pipelining Mode:** The CORDIC core provides three pipelining modes: None, Optimal, and Maximum. The choice of pipelining mode is based on the selection of Functional Configuration and Architectural Configuration. Unavailable pipelining modes are greyed out in the CORDIC GUI.
  - **None**: the CORDIC core is implemented without pipelining.
  - **Optimal**: the CORDIC core is implemented with as many stages of pipelining as possible without using any additional LUTs.
  - **Maximum**: the CORDIC core is implemented with a pipeline after every shift-add sub stage.
- **Data Format**: The CORDIC core provides three formats for expressing the X and Y components of data samples:
  - **Signed Fraction**: Default setting. The X and Y inputs and outputs are expressed as fixed-point twos complement numbers with an integer width of 2 bits. Example: 11100000 represents the value -0.5.
  - **Unsigned Fraction**: The X and Y inputs and outputs are expressed as unsigned fixed-point number with an integer with of 1 bit.
    Available only for Square Root functional configuration. Example: 11100000 represents the value +1.75.

- **Unsigned Integer**: The X and Y inputs and outputs express unsigned integers. Available only for Square Root functional configuration. Example: 11100000 represents the value +224.

- **Phase Format:** The CORDIC core provides two Phase Format options:
  - **Radians**: The phase is expressed as a fixed-point twos complement numbers with an integer width of 3 bits, in radian units.
    Example: 01100000 represents the value 3.0 radians.
  - **Scaled Radians**: The phase is expressed as fixed-point twos complement numbers with an integer width of 3 bits, with Pi-radian units. One scaled-radian equals Pi * 1 radians.
    Example: 11110000 represents the value -0.5 * Pi radians.

See Input/Output Data Representation for more information about CORDIC binary data formats.

Input / Output Options:

The CORDIC core provides four input / output common configuration options.

- **Input Width**: Input Width controls the widths of the input ports, X_IN, Y_IN and PHASE_IN. The Input Width can be configured in the range 8 to 48 bits.
- **Register Inputs**: Selects if the input signals X_IN, Y_IN, PHASE_IN are registered.
- **Output Width**: Output Width controls the widths of the output ports, X_OUT, Y_OUT, PHASE_OUT. The Output Width can be configured in the range 8 to 48 bits.
- **Register Outputs**: Selects if the output signals, X_OUT, Y_OUT, PHASE_OUT are registered.

- **Round Mode:** The CORDIC core provides four rounding modes. Table 4-1 illustrates the behavior of the different Rounding modes.
  - **Truncate**: The X_OUT, Y_OUT, and PHASE_OUT outputs are truncated.
  - **Positive Infinity**: The X_OUT, Y_OUT, and PHASE_OUT outputs are rounded such that 1/2 is rounded up (towards positive infinity). It is equivalent to the MATLAB® function floor(x+0.5).
  - **Pos Neg Infinity**: The outputs X_OUT, Y_OUT, and PHASE_OUT are rounded such that 1/2 is rounded up (towards positive infinity) and -1/2 is rounded down (towards negative infinity). It is equivalent to the MATLAB function round(x).
  - **Nearest Even**: The X_OUT, Y_OUT, and PHASE_OUT outputs are rounded toward the nearest even number such that a 1/2 is rounded down and 3/2 is rounded up.

Table 4-1: **Rounding Modes**

Table 4-2:	Truncate	Pos Neg Infinity	Positive Infinity	Nearest Even
1.50	1	2	2	2
1.00	1	1	1	1
0.50	0	1	1	0
0.25	0	0	0	0
0.00	0	0	0	0
- 0.25	-1	0	0	0
- 0.50	-1	-1	0	-1
- 0.75	-1	-1	-1	-1

- **Advanced Configuration Parameters**
  - **Iterations:** Controls the number of internal add-sub iterations to perform.When Iterations is set to zero, the number of iterations performed is determined by the required accuracy of the output. By default, Iterations is set to zero, thus the number of iterations is automatically determined. In this case, for all operations except square root, the basic number of iterations set is the output width. This is then modified as follows; for **Coarse Rotation** set, subtract 2; for hyperbolic functions with the iterations number of 4 or more, add 1 and for greater than or equal to 13, add a further 1.
    For square root, the number of iterations set when automatic is selected is again the output width but modified as follows: **Round Mode** set to positive infinity, add 1; **Round Mode** set to negative infinity or round to nearest even, add 2.
  - **Precision**: Configures the internal precision of the add-sub iterations. When Precision is set to zero, internal precision is determined automatically based on the required accuracy of the output and the number of internal iterations. By default, Precision is set to zero, thus the internal precision is automatically determined. When Precision is set to (Input_Width + Output_Width + $\log_2$(Output_Width)) the output phase is precise to the full output width regardless of input magnitude. However, the output phase accuracy is still limited by the OQEIQ component of Output Quantization Error and by the number of Iterations of the CORDIC Micro-Rotation block.
    For all operations other than square root, the internal precision set when Precision is set to zero is the Output_Width plus the $\log_2$ rounded up of the number of iterations, for example, the $\log_2$ rounded up of 8 is 3 and of 9 is 4.
    For square root, the precision is the Output_Width plus 1 for **Round Mode** positive or negative infinity and plus 2 for round to nearest even.
  - **Coarse Rotation**: Controls the instantiation of the coarse rotation module. Instantiation of the coarse rotation module is the default for the functional configurations: Vector rotation, Vector translation, Sin and Cos, and ArcTan. If Coarse Rotation is turned off for these functions, the input/output range is limited to the first quadrant (-Pi/4 to + Pi/4). Coarse rotation is not required for the Sinh and Cosh, ArcTanh, and Square Root configurations. The standard CORDIC

algorithm operates over the first quadrant. Coarse Rotation extends the CORDIC operational range to the full circle by rotating the input sample into the first quadrant and inverse rotating the output sample back into the appropriate quadrant.

- **Compensation Scaling:** Controls the compensation scaling module used to compensate for CORDIC magnitude scaling. CORDIC magnitude scaling affects the Vector Rotation and Vector Translation functional configurations. It does *not* affect the Sin, Cos, Sinh, Cosh, ArcTan, ArcTanh and Square Root functional configurations. For the latter configurations, compensation scaling is set to No Scale Compensation. CORDIC magnitude scaling is a side effect of the CORDIC algorithm. The magnitude outputs, X and Y, are generated scaled by the CORDIC scale factor, Z_i. The compensation scaling module compensates for the effect of CORDIC magnitude scaling by scaling the outputs, X and Y, by $1/Z_i$.
  - **No Scale Compensation**: The outputs X and Y are not compensated and are generated, scaled by the ratio Z_i.
  - **LUT Based**: The outputs X and Y are compensated using a LUT-based Constant Coefficient Multiplier.
  - **BRAM:** The outputs X and Y are compensated using a block RAM-based Constant Coefficient Multiplier.
  - **Embedded Multiplier**: The outputs X and Y are compensated using the DSP Slice.

Page 2 - AXI4-Stream Options

Used to configure the AXI4-Stream interfaces.

Cartesian Channel Options:

- **Has TLAST:** Selects optional port `s_axis_cartesian_tlast`
- **HAS TUSER:** Selects optional port `s_axis_cartesian_tuser`
- **TUSER Width:** Determines width of `s_axis_cartesian_tuser`

Phase Channel Options:

- **Has TLAST:** Selects optional port `s_axis_phase_tlast`
- **HAS TUSER:** Selects optional port `s_axis_phase_tuser`
- **TUSER Width:** Determines width of `s_axis_phase_tuser`

Flow Control: Selects Blocking or NonBlocking behavior of AXI4-Stream channels for the whole core.

Optimize Goal: Selects between performance and resources as the goal of optimization. Specifically in AXI4-Stream implementation, selecting Performance can lead to a larger

output buffer, but performance similar to DSP Slices. Selecting Resources limits the size of the output buffer, but might result in a lower maximum achievable clock frequency.

Output has TREADY: Selects optional port `m_axis_dout_tready`. With this option, the core might be stalled by backpressure and so needs an output buffer (internally). Without this option, the core might not be stalled and does not require an output buffer which results in a smaller design.

Output TLAST Behavior: Selects the logic combination of input `tlasts` to become `m_axis_dout_tlast`. When neither input `tlast` is selected this is forced to Null and `m_axis_dout_tlast` is not present. When only one is selected, `m_axis_dout_tlast` exists and outputs the delayed input `tlast`. When both input `tlasts` are selected, the output, suitably delayed can be selected as either input, or a logical OR of the inputs, or a logical AND of the inputs.

Optional Pins

- ACLKEN: Selects optional port `ACLKEN`. This is provided primarily for ease of migration. It is not recommended when designing with AXI4-Stream Blocking modes.
- ARESETn: Selects optional port `ARESETn`. `ARESETn` is active-Low and must be asserted for a minimum of two `aclk` cycles to reset the core.

User Parameters

Table 4-3 shows the relationship between the fields in the Vivado IDE (described in Customizing and Generating the Core) and the User Parameters (which can be viewed in the Tcl console).

Table 4-3: **Vivado IDE Parameter to User Parameter Relationship**

GUI Parameter	User Parameter	Default Value
Functional Selection	functional_selection	Rotate
Architectural Configuration	architectural_configuration	Parallel
Pipelining Mode	pipelining_mode	Maximum
Data Format	data_format	SignedFraction
Phase Format	phase_format	Radians
Input Width	input_width	16
Output Mode	output_width	16
Round Mode	round_mode	Truncate
Iterations	iterations	0
Precision	precision	0
Coarse Rotation	coarse_rotation	True
Compensation Scaling	compensation_scaling	No_Scale_Compensation

Table 4-3: **Vivado IDE Parameter to User Parameter Relationship *(Cont'd)***

GUI Parameter	User Parameter	Default Value
Cartesian Channel Options: Has TUSER	cartesian_has_tuser	False
Cartesian Channel Options: TUSER Width	cartesian_tuser_width	1
Cartesian Channel Options: Has TLAST	cartesian_has_tlast	False
Phase Channel Options: Has TUSER	phase_has_tuser	False
Phase Channel Options: TUSER Width	phase_tuser_width	1
Phase Channel Options: Has TLAST	phase_has_tlast	False
Flow Control	flow_control	NonBlocking
Optimize Goal	optimize_goal	Resources
Output has TREADY	out_tready	False
Output TLAST Behavior	out_tlast_behv	Null
ACLKEN	aclken	False
ARESETN	aresetn	false

Output Generation

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

System Generator for DSP

This section details the parameters that differ from the CORDIC GUI in the Vivado IDE. See Customizing and Generating the Core for more information about all other parameters. The CORDIC core can be found in the Xilinx® Blockset in the Math section. The block is called "CORDIC v6.0". See the System Generator for DSP Help page for the "CORDIC v6.0" block for more information on parameters not mentioned here. The System Generator for DSP GUI provides the same parameters as the CORDIC GUI in the Vivado IDE.

Implementation

See the *System Generator for DSP User Guide* (UG640) [Ref 9] for information about the FPGA Area Estimation parameter.

Constraining the Core

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

Required Constraints

This section is not applicable for this IP core.

Device, Package, and Speed Grade Selections

This section is not applicable for this IP core.

Clock Frequencies

This section is not applicable for this IP core.

Clock Management

This section is not applicable for this IP core.

Clock Placement

This section is not applicable for this IP core.

Banking

This section is not applicable for this IP core.

Transceiver Placement

This section is not applicable for this IP core.

I/O Standard and Placement

This section is not applicable for this IP core.

Simulation

For comprehensive information about Vivado® simulation components, as well as information about using supported third party tools, see the *Vivado Design Suite User Guide: Logic Simulation* (UG900) [Ref 8].

To simulate the core, generate the core simulation model and demonstration test bench. Ensure that the demonstration test bench is the top-level entity in the simulation options.

IMPORTANT: *For cores targeting 7 series or Zynq-7000 devices, UNIFAST libraries are not supported. Xilinx IP is tested and qualified with UNISIM libraries only.*

Synthesis and Implementation

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

C Model

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

Features

- Bit accurate with CORDIC core
- Available for 64-bit Linux platforms
- Available for 64-bit Windows platforms
- Supports all features of the CORDIC 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 5-1 and Table 5-2 list the contents of each ZIP file.

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

File	Description
cordic_v6_0_bitacc_cmodel.h	Header file which defines the C model API
libIp_cordic_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 5-2: **C Model ZIP File Contents: Windows**

File	Description
cordic_v6_0_bitacc_cmodel.h	Header file which defines the C model API
libIp_cordic_v6_0_bitacc_cmodel.dll	Model dynamically linked library
libIp_cordic_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

Installation instructions for Linux and Windows operating systems are described in this section.

Linux

- Unpack the contents of the ZIP file.
- Ensure that the directory where the `libIP_cordic_v6_0_bitacc_smodel.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_cordic_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 `cordic_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 CORDIC C model are listed in Table 5-3.

Table 5-3: **C Model Data Types**

Name	Type	Description
xip_real	Double	Base type for scalar inputs and outputs (magnitude and phase).
xip_complex	Struct {re,im} xip_real	Base type for Cartesian inputs and outputs.
xip_array_real	Struct	Structure to hold scalar input or scalar output.
xip_array_complex	Struct	Structure to hold Cartesian (complex) data for input or output from the CORDIC core.
xip_uint	Unsigned Int	Used for configuration parameter of integer or Boolean type. For Boolean: 0=false 1=true
xip_cordic_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_cordic_v6_0_status but used for functions which are not core-specific.
xip_cordic_v6_0_config	Struct	The configuration of the core itself. The members of this structure are listed in the cordic_v6_0_bitacc_cmodel.h file. The names closely match the same names in XCI files. The cordic_v6_0_bitacc_cmodel.h file also contains #defined values for all.
xip_cordic_v6_0	Struct	Type defined which C (not C++) can use as a handle (pointer) to a C++ object – the C model itself.

The xip_array_complex and xip_array_real 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 CORDIC C model functions.

Data Values

The CORDIC core input and output fields are in standard twos complement binary form with widths between 8 and 48 bits.

The CORDIC C model expects data to be in the C type *double*, equal to the raw value of the twos complement bit vector input to the HDL. The parameters, DataFormat and PhaseFormat determine how the value is interpreted by the C model just as the equivalent parameters used to configure the HDL core determine how the HDL core interprets input binary vectors.

For example, the 8 bit vector 11110000 input to the core should be input to the C model as -16, regardless of the value of parameters for data format or phase format.

Functions

There are several accessible C model functions.

Information Functions

Table 5-4 lists the information functions. The prototypes for these functions can be found in the C model header file.

Table 5-4: **Information Functions**

Name	Return	Arguments	Description
xip_cordic_get_version	Const char*	Void	Return the CORDIC C model version as a null terminated string. For v6.0, this is '6.0'.
xip_cordic_v6_0_get_default_config	xip_cordic_v6_0_status	xip_cordic_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 5-5.

Table 5-5: **Initialization Functions**

Name	Return	Arguments	Description
xip_cordic_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_cordic_v6_0_destroy	xip_cordic_v6_0_status	Pointer to xip_cordic_v6_0 (C model itself)	Deallocates memory owned by C model and destroys C model itself.
xip_cordic_v6_0_get_config	xip_cordic_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.
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 real or complex.

Execution Functions

The run time functions of the C model are described in Table 5-6.

Table 5-6: **Execution Functions**

Name	Return	Arguments	Description
xip_cordic_v6_0_data_do	xip_cordic_v6_0_status	Pointer to C model, function selection, pointer to magnitude in structure, pointer to phase in structure, pointer to Cartesian input structure, pointer to magnitude out structure, pointer to phase output structure, pointer to Cartesian output structure, number of samples.	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_cordic_v6_0_rotate	xip_cordic_v6_0_status	Pointer to C model, pointer to Cartesian data in structure, pointer to phase data in structure, pointer to Cartesian data out structure, number of samples	Simplified variant of xip_cordic_v6_0_data_do specifically for the rotate function.
xip_cordic_v6_0_translate	xip_cordic_v6_0_status	Pointer to C model, pointer to Cartesian data in structure, pointer to magnitude data out structure, pointer to phase data out structure, number of samples	Simplified variant of xip_cordic_v6_0_data_do specifically for the translate function.
xip_cordic_v6_0_sin_cos	xip_cordic_v6_0_status	Pointer to C model, pointer to phase data in structure, pointer to Cartesian data out structure, number of samples	Simplified variant of xip_cordic_v6_0_data_do specifically for the sin_cos function.
xip_cordic_v6_0_atan	xip_cordic_v6_0_status	Pointer to C model, pointer to Cartesian data in structure, pointer to phase data out structure, number of samples	Simplified variant of xip_cordic_v6_0_data_do specifically for the atan function.
xip_cordic_v6_0_sinh_cosh	xip_cordic_v6_0_status	Pointer to C model, pointer to phase data in structure, pointer to Cartesian data out structure, number of samples	Simplified variant of xip_cordic_v6_0_data_do specifically for the sinh_cosh function.
xip_cordic_v6_0_atanh	xip_cordic_v6_0_status	Pointer to C model, pointer to Cartesian data in structure, pointer to phase data out structure, number of samples	Simplified variant of xip_cordic_v6_0_data_do specifically for the atanh function.
xip_cordic_v6_0_sqrt	xip_cordic_v6_0_status	Pointer to C model, pointer to magnitude data in structure, pointer to magnitude data out structure, number of samples	Simplified variant of xip_cordic_v6_0_data_do specifically for the sqrt function.
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 real or complex.

Compiling

Compilation of user code requires access to the `cordic_v6_0_bitacc_cmodel.h` header file and the header files of the MPIR dependent libraries, `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 `cordic_v6_0_bitacc_cmodel.h` header file must be included first, because it defines some symbols that are used in the MPIR header files. The `cordic_v6_0_bitacc_cmodel.h` header file includes the MPIR header files, so these do not need to be explicitly included in source code that uses the C model. When compiling on Windows, the symbol `NT` must be defined, either by a compiler option, or in user source code before the `cordic_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 MPIR libraries. 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_cordic_v6_0_bitacc_cmodel.so`

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

```
-L. -Wl,-rpath,. -lIp_cordic_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. -lIpcordic_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_cordic_v6_0_bitacc_cmodel.dll`

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

- `libgmp.lib`
- `libIp_cordic_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. This 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/. A pre-compiled MPIR library is provided with the C model, using the following version:

- MPIR 2.6.0

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

GMP and MPIR in particular 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 that was used to compile the provided libraries. Source code and compilation scripts for any version of the libraries can be obtained from the GMP [Ref 10] and MPIR [Ref 11] websites.

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. The comments assist in understanding the code.

Test Bench

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

Demonstration Test Bench

When the core is generated in the Vivado® IDE, 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

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

Demonstration Test Bench in Detail

The demonstration test bench performs the following tasks:

- Instantiates the core
- Generates stimulus data sets for each input channel. Both sets are rotating phasors
- Generates a clock signal
- Drives the clock enable and reset input signals of the core (if present)
- Drives the input signals of the core to demonstrate core features
- Checks that the core output signals obey AXI protocol rules (data values are not checked in order to keep the test bench simple)
- Provides signals showing the separate fields of AXI `tdata` and TUSER signals

The demonstration test bench drives the input signals of the core to demonstrate the features and modes of operation of the core. The CORDIC core is driven with two simple data sets (phasors of different periods) to stimulate the core with a wide range of positive and negative values, including zero. The input data is pre-generated and stored in data tables, and the test bench drives the core data inputs with the ramp 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 back pressure by deasserting the `tready` signal of the master channel
  - Phase 3: deprive a single slave channel 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 a single slave channel 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 inputs of the core with different data or to perform different operations. Input data is pre-generated in the `create_ip_cartesian_table` and `create_ip_phase_table` functions and stored in the IP_cartesian_DATA and IP_phase_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_cartesian_TABLE and T_IP_phase_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.

Upgrading

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

Migrating to the Vivado Design Suite

For information on migrating to the Vivado Design Suite, see the *ISE to Vivado Design Suite Migration Guide* (UG911) [Ref 12].

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.

The Vivado Design Suite core upgrade functionality can be used to upgrade an existing XCO/XCI file from v4.0 or v5.0 to CORDIC v6.0. See Instructions for Minimum Change Migration (v4.0 to v6.0). There are no changes of functionality, port or configuration from v5.0 to v6.0.

Parameter Changes

Table A-1 shows the parameter changes from version 4.0 to version 6.0.

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

Version 4.0	Version 6.0	Notes
Component_Name	Component_Name	Unchanged
Functional_Selection	Functional_Selection	Unchanged
Architectural_Configuration	Architectural_Configuration	Unchanged
Pipelining_Mode	Pipelining_Mode	Unchanged
Data_Format	Data_Format	Unchanged

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

Version 4.0	Version 6.0	Notes
Phase_Format	Phase_Format	Unchanged
Input_Width	Input_Width	Unchanged
Register_Inputs	Register_Inputs	Unchanged
Output_Width	Output_Width	Unchanged
Register_Inputs	Register_Inputs	Unchanged
Round_Mode	Round_Mode	Unchanged
Iterations	Iterations	Unchanged
Precision	Precision	Unchanged
Coarse_Rotation	Coarse_Rotation	Unchanged
Compensation_Scaling	Compensation_Scaling	Unchanged
CE	ACLKEN	Renamed only.
SCLR	ARESETn	Renamed. Note that the parameter has not changed, but the signal in question is now active-Low.
ND		Deprecated.
RDY		Deprecated.
X_OUT		Deprecated. If X_OUT is not connected, unused logic is removed automatically.
Y_OUT		Deprecated. If Y_OUT is not connected, unused logic is removed automatically.
Phase_Output		Deprecated. If PHASE_OUT is not connected, unused logic is removed automatically.
	cartesian_has_tuser	New addition in v5.0
	cartesian_tuser_width	New addition in v5.0
	cartesian_has_tlast	New addition in v5.0
	phase_has_tuser	New addition in v5.0
	phase_tuser_width	New addition in v5.0
	phase_has_tlast	New addition in v5.0
	flow_control	New addition in v5.0
	optimize_goal	New addition in v5.0
	out_tready	New addition in v5.0
	out_tlast_behv	New addition in v5.0

Port Changes

Table A-2: **Port Changes from Version v4.0 to v6.0**

Version 4.0	Version 6.0	Notes
CLK	aclk	Rename only
CE	ACLKEN	Rename only
SCLR	ARESETn	Rename and change of sense (now active-Low). Note that ARESETn should be asserted for a minimum of 2 cycles.
ND		Deprecated. However, this is analogous to the tvalid signals. See Instructions for Minimum Change Migration (v4.0 to v6.0).
RFD		Deprecated. However, this is analogous to the tready signals. See Instructions for Minimum Change Migration (v4.0 to v6.0).
RDY		Deprecated. However, this is analogous to the m_axis_dout_tvalid. See Instructions for Minimum Change Migration (v4.0 to v6.0).
X_IN	s_axis_cartesian_tdata subfield	subfield of s_axis_cartesian_tdata See TDATA Packing.
Y_IN	s_axis_cartesian_tdata subfield	subfield of s_axis_cartesian_tdata. See TDATA Packing.
PHASE_IN	s_axis_phase_tdata subfield	s_axis_phase_tdata(N-1:0)
X_OUT	m_axis_dout_tdata subfield	Subfield of m_axis_dout_tdata. See TDATA Packing.
Y_OUT	m_axis_dout_tdata subfield	Subfield of m_axis_dout_tdata. See TDATA Packing.
PHASE_OUT	m_axis_dout_tdata subfield	Subfield of m_axis_dout_tdata. See TDATA Packing.

Latency Changes

With the addition of AXI4-Stream interfaces, the latency of the CORDIC core v6.0 is different compared to v4.0 for AXI Blocking mode. Latency is the same as v4.0 in v6.0 for AXI NonBlocking mode. Importantly, when in Blocking Mode, the latency of the core is variable due to the FIFO nature of the AXI4-Stream protocol, so only the minimum possible latency can be determined. Relative to v4.0, with Blocking and Output `tready` present, minimum latency is 3 cycles greater. With no output `tready`, minimum latency is increased by one cycle only. There are no latency changes from v5.0 to v6.0.

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

There are no changes of behavior, ports, or parameterization between v5.0 and v6.0. Use the following information to configure the CORDIC core v6.0 to most closely mimic the behavior of v4.0.

Parameters

- Set FlowControl to NonBlocking.

All other new parameters default to FALSE and can be ignored.

Ports

- Rename and map signals as detailed in Port Changes.
- Map ND to both `s_axis_cartesian_tvalid` and `s_axis_phase_tvalid`, if present for the function in question.
- Map RFD to `s_axis_cartesian_tready` or `s_axis_phase_tready`.
- Map RDY to `m_axis_dout_tvalid`.

Performance and resource use is unchanged compared with CORDIC v4.0 and v5.0 other than small changes due to the use of different tools

Functionality Changes

There are no functionality changes in v6.0 compared to v5.0. The addition of AXI4-Stream interfaces in v5.0 causes functionality changes compared to v4.0. See Instructions for Minimum Change Migration (v4.0 to v6.0).

Simulation Changes

Starting with CORDIC v6.0 (2013.3 version) behavioral simulation models have been replaced with IEEE 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 8].

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 CORDIC core, the Xilinx Support web page contains key resources such as product documentation, release notes, answer records, information about known issues, and links for obtaining further product support.

Documentation

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

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

Answer Records

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

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

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

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

Master Answer Record for the CORDIC Core

AR: 54497

Technical Support

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

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

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

Debug Tools

There are many tools available to address CORDIC design issues. It is important to know which tools are useful for debugging various situations.

Vivado Design Suite Debug Feature

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

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

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

See the *Vivado Design Suite User Guide: Programming and Debugging* (UG908) [Ref 13].

Reference Boards

Various Xilinx development boards support the CORDIC core. These boards can be used to prototype designs and establish that the core can communicate with the system.

- 7 series FPGA evaluation boards
  - KC705
  - KC724

C Model Reference

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

Third-Party Tools

MATLAB® can be used to debug this core.

Simulation Debug

The simulation debug flow for Mentor Graphics Questa Advanced Simulator is shown in Figure B-1. A similar approach can be used with other simulators.

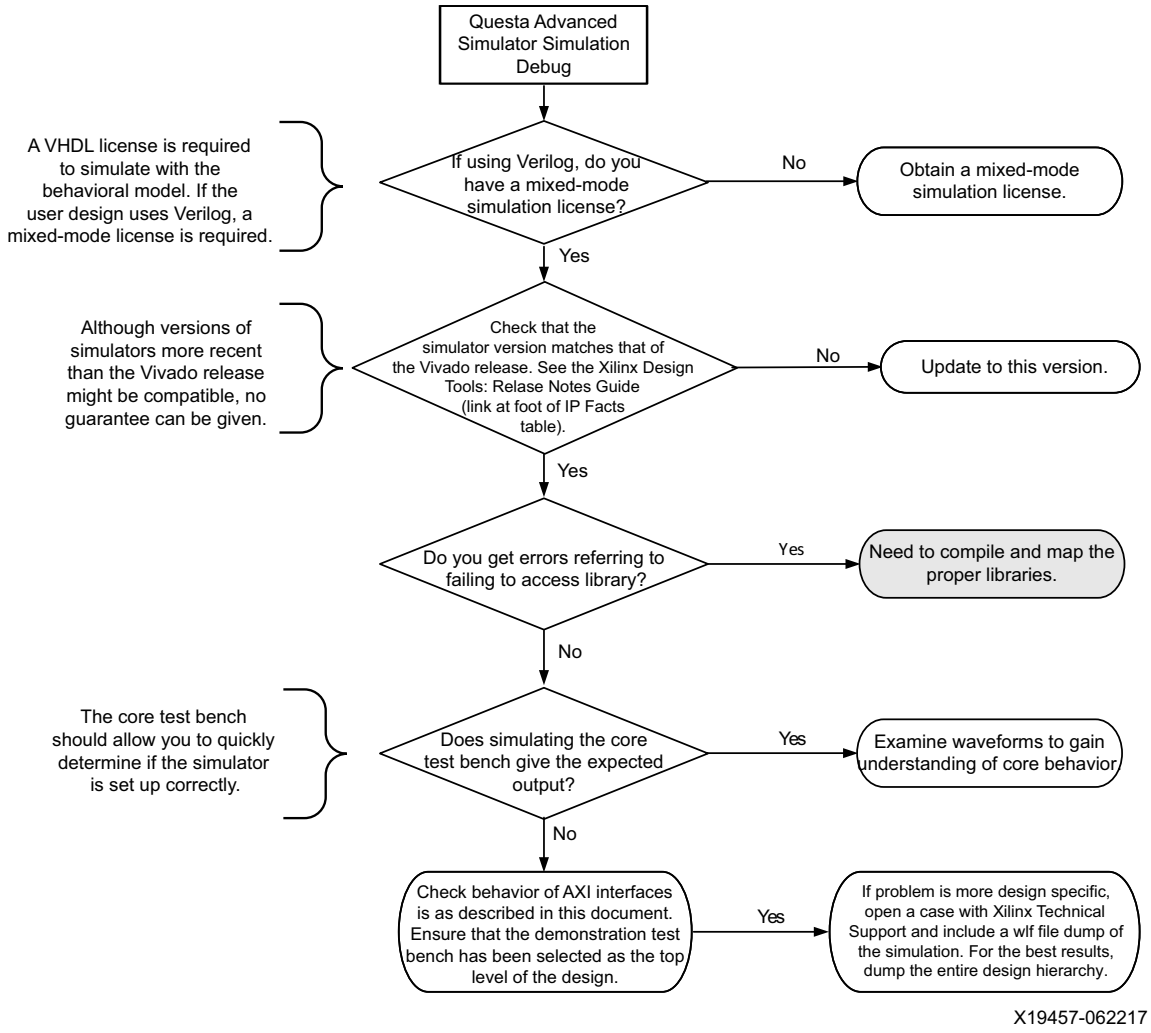

Figure B-1: **Questa Advanced Simulator Debug Flow Diagram**

AXI4-Stream Interface Debug

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

- If transmit `<interface_name>_tready` is stuck Low following the **`<interface_name>_tvalid`** input being asserted, the core cannot send data.
- If the receive **`<interface_name>`**`_tvalid` is stuck Low, the core is not receiving data.
- Check that the `ACLK` inputs are connected and toggling.
- Check that the AXI4-Stream waveforms are being followed. See Figure 3-1 to Figure 3-3.
- Check core configuration.

Additional Resources and Legal Notices

Xilinx Resources

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

Documentation Navigator and Design Hubs

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

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

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

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

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

References

These documents provide supplemental material useful with this product guide:

1. Volder, J., *The CORDIC Trigonometric Computing Technique* IRE Trans. Electronic Computing, Vol. EC-8, Sept. 1959, pp330-334.
2. Walther, J.S., *A Unified Algorithm for Elementary Functions*, Spring Joint computer conf., 1971, proc., pp379-385.
3. *AMBA® AXI4-Stream Protocol Specification* (ARM IHI 0051A)
4. *Vivado Design Suite AXI Reference Guide* (UG1037)
5. *Vivado Design Suite User Guide: Designing IP Subsystems using IP Integrator* (UG994)
6. *Vivado Design Suite User Guide: Designing with IP* (UG896)
7. *Vivado Design Suite User Guide: Getting Started* (UG910)
8. *Vivado Design Suite User Guide: Logic Simulation* (UG900)
9. *System Generator for DSP User Guide* (UG640)
10. The GNU Multiple Precision Arithmetic (GMP) Library gmplib.org
11. The GNU Multiple Precision Integers and Rationals (MPIR) library (www.mpir.org)
12. *ISE to Vivado Design Suite Migration Guide* (UG911)
13. *Vivado Design Suite User Guide: Programming and Debugging* (UG908)
14. *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
10/04/2017	6.0	Example 4: PHASE_IN decimal value updated.
10/05/2016	6.0	Example 6: ArcTanh updated.
11/18/2015	6.0	UltraScale+ device support added.
06/24/2015	6.0	• Remove references to divider signals.
04/02/2014	6.0	• Added link to resource utilization figures • Added User Parameter table (Table 4-3)
12/18/2013	6.0	Added UltraScale™ architecture support.

Date	Version	Revision
10/02/2013	6.0	Document version number advanced to match the core version number. Added Chapter 5, C Model.
03/20/2013	1.0	Initial release as a Product Guide; replaces DS858. No other documentation changes.

Please Read: Important Legal Notices