



# CORDIC Core User Guide

---

**UG-CORE-CORDIC-v1.3**  
**November 2024**  
[www.efinixinc.com](http://www.efinixinc.com)



# Contents

<b>Introduction.....</b>	<b>3</b>
<b>Features.....</b>	<b>3</b>
<b>Device Support.....</b>	<b>4</b>
<b>Resource Utilization and Performance.....</b>	<b>4</b>
<b>Release Notes.....</b>	<b>6</b>
<b>Functional Description.....</b>	<b>7</b>
Ports.....	8
CORDIC Operation.....	12
<b>IP Manager.....</b>	<b>13</b>
<b>Customizing the CORDIC.....</b>	<b>14</b>
<b>CORDIC Testbench.....</b>	<b>14</b>
cos and sin Testbench.....	15
arcsin and arccos Testbench.....	15
arctan Testbench.....	16
cosh and sinh Testbench.....	16
tanh Testbench.....	17
exp Testbench.....	17
ln Testbench.....	17
sqrt Testbench.....	18
<b>Revision History.....</b>	<b>19</b>

# Introduction

CORDIC algorithm is a hardware-efficient algorithm that uses rotations to calculate a wide range of elementary functions. The CORDIC core applies the CORDIC algorithm to calculate trigonometric functions, hyperbolic functions, square roots, exponentials, and natural logarithms.

Use the IP Manager to select IP, customize it, and generate files. The CORDIC core has an interactive wizard to help you set parameters. The wizard also has options to create a testbench and/or example design targeting an Efinix<sup>®</sup> development board.

## Features

- Supports 8 CORDIC functions:
  - Sine and cosine
  - Arcsine and arccosine
  - Arctangent
  - Hyperbolic sine and hyperbolic cosine
  - Hyperbolic tangent
  - Exponential
  - Natural log
  - Square root
- Verilog HDL RTL and simulation testbench

# Device Support

*Table 1: CORDIC Core Device Support*

FPGA Family	Supported Device
Trion	All
Titanium	All
Topaz	All

## Resource Utilization and Performance



**Note:** The resources and performance values provided are based on some of the supported FPGAs. These values are just guidance and can change depending on the device resource utilization, design congestion, and user design.

*Table 2: Titanium Resource Utilization and Performance*

FPGA	Mode	Logic and Adders	Flip-flops	Memory Blocks	DSP Blocks	f <sub>MAX</sub> (MHz) <sup>(1)</sup>	Efinity <sup>®</sup> Version <sup>(2)</sup>
Ti60 F225 C4	arcsin and arccos	1,279	132	0	0	290	2022.2
	arctan	984	96	0	0	333	
	exp	1,990	231	0	2	92	
	ln	1,186	168	0	2	319	
	sin and cos	1,474	169	0	0	335	
	sinh and cosh	2,364	217	0	2	93	
	sqrt	1,667	169	0	2	273	
	tanh	1,740	228	0	0	321	

*Table 3: Trion<sup>®</sup> Resource Utilization and Performance*

FPGA	Mode	Logic Utilization (LUTs)	Registers	Memory Blocks	Multipliers	f <sub>MAX</sub> (MHz) <sup>(1)</sup>	Efinity <sup>®</sup> Version <sup>(2)</sup>
T20 BGA256 C4	arcsin and arccos	1,279	132	0	0	84	2022.2
	arctan	984	96	0	0	86	
	exp	2,123	237	0	2	28	
	ln	1,189	168	0	2	89	

<sup>(1)</sup> Using default parameter settings.

<sup>(2)</sup> Using Verilog HDL.

FPGA	Mode	Logic Utilization (LUTs)	Registers	Memory Blocks	Multipliers	$f_{MAX}$ (MHz) <sup>(1)</sup>	Efinity <sup>®</sup> Version <sup>(2)</sup>
	sin and cos	1,474	169	0	0	103	
	sinh and cosh	2,287	222	0	2	26	
	sqrt	1,548	168	0	2	82	
	tanh	1,523	229	0	0	88	

# Release Notes

You can refer to the IP Core Release Notes for more information about the IP core changes. The IP Core Release Notes are available on the [Efinity Downloads](#) page under each Efinity software release version.



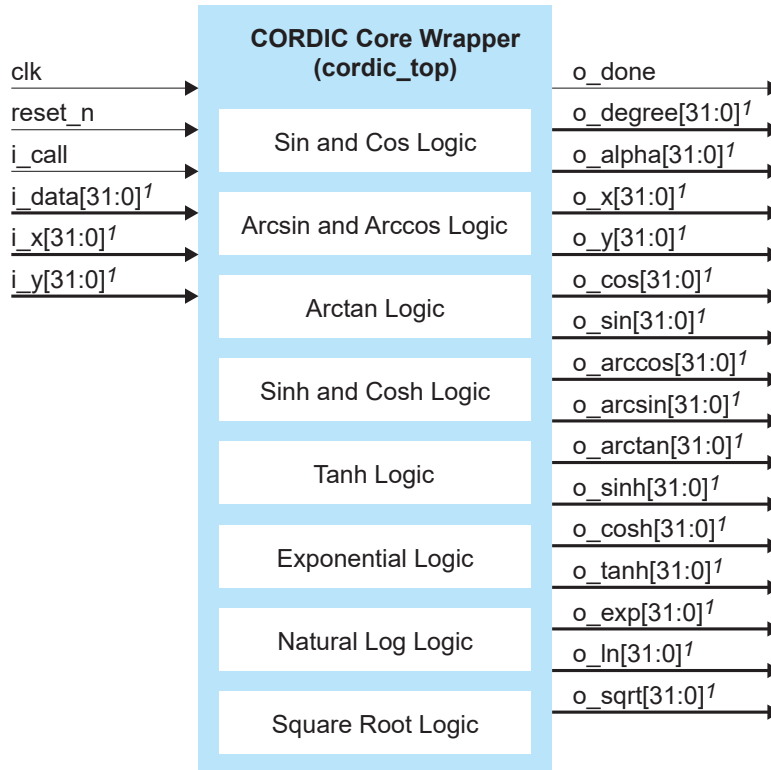
**Note:** You must be logged in to the Support Center to view the IP Core Release Notes.

---

# Functional Description

The CORDIC core consists of CORDIC logic that performs the supported algorithms. Other than `clk`, `reset_n`, `i_call`, and `o_done`, the ports vary depending on which CORDIC mode you select in the IP Manager.

Figure 1: CORDIC Core Block Diagram



1. Applicable ports are based on the selected CORDIC mode.

## Ports

**Table 4: Common Ports**

These signals are applicable to all CORDIC core modes. Refer to the subsequent tables for mode-specific ports.

Port	Direction	Description
clk	Input	Core operating clock.
reset_n	Input	Active low core reset.
i_call	Input	Valid signal for input. Assert this signal high together with reset to start operation. Deassert this signal when o_done is high. Assert this signal again when there is a new input to calculate. Optionally, you can leave this signal high to continue operation but send the new input when o_done is high. Refer to the specific CORDIC mode input ports.
o_done	Output	Valid signal for output. The core asserts this signal when operation is complete the output available at the output ports. Refer to the specific CORDIC mode output ports.

**Table 5: Sin and Cos Mode Ports**

Port	Direction	Description
i_data[31:0]	Input	Signed input data in degree. Valid range from 0 to 360.
o_cos[31:0]	Output	Signed results of cosine function. Divide the value by 65536 for a floating-point result. Precision: $\pm 0.0001$
o_sin[31:0]	Output	Signed results of sine function. Precision: $\pm 0.0001$ Divide the value by 65536 for a floating-point result.
o_deg[31:0]	Output	Not applicable.
o_x[31:0]	Output	Similar to o_cos output.
o_y[31:0]	Output	Similar to o_sin output.

**Table 6: Arcsin and Arccos Mode Ports**

Port	Direction	Description
i_data[31:0]	Input	Signed input data. Valid range from -1 to 1. Multiply the value by 65536 before input.
o_arccos[31:0]	Output	Signed results of arccosine function. Divide the value by 65536 for a floating-point result in degree. Precision: $\pm 0.0001$ degree
o_arcsin[31:0]	Output	Signed results of arcsinine function. Divide the value by 65536 for a floating-point result in degree. Precision: $\pm 0.0001$ degree
o_deg[31:0]	Output	Not applicable.
o_x[31:0]	Output	Similar to o_arccos output.
o_y[31:0]	Output	Similar to o_arcsin output.

**Table 7: Arctan Mode Ports**

Port	Direction	Description
i_x[31:0]	Input	Signed x-axis input data. Valid range from -255 to 255. Multiply the value by 65536 before input.
i_y[31:0]	Input	Signed y-axis input data. Valid range from -255 to 255. Multiply the value by 65536 before input.
o_arctan[31:0]	Output	Signed results of arctangent function. Divide the value by 65536 for a floating-point result in degree. Precision: $\pm 0.0001$ degree
o_deg[31:0]	Output	Similar to o_arctan output.
o_x[31:0]	Output	Not applicable.
o_y[31:0]	Output	Not applicable.

**Table 8: Sinh and Cosh Mode Ports**

Port	Direction	Description
i_data[31:0]	Input	Signed input data. Valid range from -3.142 to 3.142. Multiply the value by 65536 before input.
o_cosh[31:0]	Output	Signed results of hyperbolic cosine function. Divide the value by 65536 for a floating-point result. Precision: $\pm 0.0001$
o_sinh[31:0]	Output	Signed results of hyperbolic sine function. Divide the value by 65536 for a floating-point result. Precision: $\pm 0.0001$
o_deg[31:0]	Output	Not applicable.
o_x[31:0]	Output	Similar to o_cosh output.
o_y[31:0]	Output	Similar to o_sinh output.

**Table 9: Tanh Mode Ports**

Port	Direction	Description
i_data[31:0]	Input	Signed input data. Valid range from -1.13 to 1.13. Multiply the value by 65536 before input.
o_tanh[31:0]	Output	Signed results of hyperbolic tangent function. Divide the value by 65536 for a floating-point result. Precision: $\pm 0.0001$
o_cosh[31:0]	Output	Signed results of hyperbolic cosine function. Divide the value by 65536 for a floating-point result. Precision: $\pm 0.0001$
o_sinh[31:0]	Output	Signed results of hyperbolic sine function. Divide the value by 65536 for a floating-point result. Precision: $\pm 0.0001$
o_alpha[31:0]	Output	Similar to o_tanh output.
o_x[31:0]	Output	Similar to o_cosh output.
o_y[31:0]	Output	Similar to o_sinh output.

**Table 10: Exponential Mode Ports**

Port	Direction	Description
i_data[31:0]	Input	Signed input data. Valid range from -10 to 10. Multiply the value by 65536 before input.
o_exp[31:0]	Output	Signed results of exponential function. Divide the value by 65536 for a floating-point result. Precision: $\pm 0.0001$
o_alpha[31:0]	Output	Similar to o_exp output.
o_x[31:0]	Output	Not applicable.
o_y[31:0]	Output	Not applicable.

**Table 11: Natural Log (ln) Mode Ports**

Port	Direction	Description
i_data[31:0]	Input	Signed input data. Valid range from 0 to 30000. Multiply the value by 65536 before input.
o_ln[31:0]	Output	Signed results of ln function. Divide the value by 65536 for a floating-point result. Precision: $\pm 0.0001$
o_alpha[31:0]	Output	Similar to o_ln output.
o_x[31:0]	Output	Not applicable.
o_y[31:0]	Output	Not applicable.

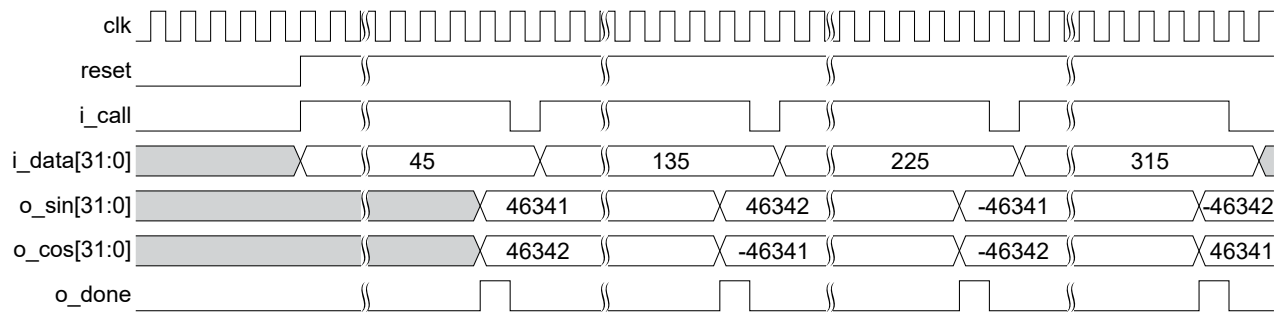
**Table 12: Square Root Mode Ports**

Port	Direction	Description
i_data[31:0]	Input	Signed input data. Valid range from 0 to 30000. Multiply the value by 65536 before input.
o_sqrt[31:0]	Output	Signed results of ln function. Divide the value by 65536 for a floating-point result. Precision: $\pm 0.0004$
o_alpha[31:0]	Output	Similar to o_sqrt output.
o_x[31:0]	Output	Not applicable.
o_y[31:0]	Output	Not applicable.

## CORDIC Operation

Assert the `reset_n` and `i_call` signals to start the CORDIC core operation. The core then takes in the value in the input port and performs the CORDIC algorithm. The result is available when the `o_done` signal is high. De-assert the `i_call` signal when the `o_done` signal is high, and assert it again when there is a new input to calculate. The following waveform illustrates an example of a cos and sin mode operation.

*Figure 2: cos and sin CORDIC Operation Example Waveform*



The output ports for a specific CORDIC function depends on the CORDIC mode you select in the IP Manager. See [Ports](#) on page 8 for more details about the ports available in each CORDIC mode.

# IP Manager

The Efinity® IP Manager is an interactive wizard that helps you customize and generate Efinity® IP cores. The IP Manager performs validation checks on the parameters you set to ensure that your selections are valid. When you generate the IP core, you can optionally generate an example design targeting an Efinity development board and/or a testbench. This wizard is helpful in situations in which you use several IP cores, multiple instances of an IP core with different parameters, or the same IP core for different projects.



**Note:** Not all Efinity IP cores include an example design or a testbench.

## Generating the CORDIC Core with the IP Manager

The following steps explain how to customize an IP core with the IP Configuration wizard.

1. Open the IP Catalog.
2. Choose **Arithmetic** > **CORDIC** core and click **Next**. The **IP Configuration** wizard opens.
3. Enter the module name in the **Module Name** box.



**Note:** You cannot generate the core without a module name.

4. Customize the IP core using the options shown in the wizard. For detailed information on the options, refer to the *Customizing the CORDIC* section.
5. (Optional) In the **Deliverables** tab, specify whether to generate an IP core example design targeting an Efinity® development board and/or testbench. These options are turned on by default.
6. (Optional) In the **Summary** tab, review your selections.
7. Click **Generate** to generate the IP core and other selected deliverables.
8. In the **Review configuration generation** dialog box, click **Generate**. The Console in the **Summary** tab shows the generation status.



**Note:** You can disable the **Review configuration generation** dialog box by turning off the **Show Confirmation Box** option in the wizard.

9. When generation finishes, the wizard displays the **Generation Success** dialog box. Click **OK** to close the wizard.

The wizard adds the IP to your project and displays it under **IP** in the Project pane.

## Generated Files

The IP Manager generates these files and directories:

- **<module name>\_define.vh**—Contains the customized parameters.
- **<module name>\_tpl.v**—Verilog HDL instantiation template.
- **<module name>\_tpl.vhd**—VHDL instantiation template.
- **<module name>.v**—IP source code.
- **settings.json**—Configuration file.
- **<kit name>\_devkit**—Has generated RTL, example design, and Efinity® project targeting a specific development board.
- **Testbench**—Contains generated RTL and testbench files.

# Customizing the CORDIC

The core has parameters so you can customize its function. You set the parameters in the **General** tab of the core's IP Configuration window.

Table 13: CORDIC Core Parameters (General Tab)

Parameter	Options	Description
Cordic Mode Selection	sin and cos, arcsin and arccos, arctan, sinh and cosh, tanh, exp, ln, sqrt	Selects CORDIC function. Default: sin and cos

## CORDIC Testbench

You can choose to generate the testbench when generating the core in the IP Manager Configuration window. To generate testbench, the **Optional Signals** option must be enabled.



**Note:** You must include all **.v** files generated in the **/testbench** directory in your simulation.



**Important:** Efinix tested the testbench generated with the default parameter options only.

Efinix provides a simulation script for you to run the testbench quickly using the Modelsim software. To run the Modelsim testbench script, run `vsim -do modelsim.do` in a terminal application. You must have Modelsim installed on your computer to use this script.

The example design includes simulation testbenches which simulate a CORDIC example. The IP Manager generates the following testbench file based on the cordic mode parameter selected:

- **tb\_cos\_sin.v**—sin and cos function
- **tb\_arccos\_arcsin.v**—arcsin and arccos function
- **tb\_arctan.v**—arctan function
- **tb\_cosh\_sinh.v**—sinh and cosh function
- **tb\_tanh.v**—tanh function
- **tb\_exp.v**—exponential function
- **tb\_ln.v**—natural log function
- **tb\_sqrt.v**—square root function

Each testbench takes a set of predefined input, convert it to a fixed-point format, perform the selected CORDIC operation, and output the results in fixed-point format. You have to convert the output to a floating-point format by dividing it by 65536.

## cos and sin Testbench

The testbench calculates the cosine and sine values for  $0^\circ$ ,  $30^\circ$ ,  $45^\circ$ ,  $60^\circ$ ,  $135^\circ$ ,  $180^\circ$ ,  $225^\circ$ ,  $300^\circ$ , and  $330^\circ$ . The testbench outputs the following:

```
# Loading sv_std.std
# Loading work.tb_sin_cos (fast)
# time<=143 i_data<= 0 o_cos<= 65539(1.000046) o_sin<= -3(-0.000046)
# time<=187 i_data<= 30 o_cos<= 56759(0.866074) o_sin<= 32769(0.500015)
# time<=231 i_data<= 45 o_cos<= 46342(0.707123) o_sin<= 46341(0.707108)
# time<=275 i_data<= 60 o_cos<= 32769(0.500015) o_sin<= 56759(0.866074)
# time<=319 i_data<= 135 o_cos<= -46341(-0.707108) o_sin<= 46342(0.707123)
# time<=363 i_data<= 180 o_cos<= -65539(-1.000046) o_sin<= 3(0.000046)
# time<=407 i_data<= 255 o_cos<= -16961(-0.258804) o_sin<= -63305(-0.965958)
# time<=451 i_data<= 300 o_cos<= 32769(0.500015) o_sin<= -56759(-0.866074)
# time<=495 i_data<= 330 o_cos<= 56759(0.866074) o_sin<= -32769(-0.500015)
# ---PASSED---
```



**Note:** The output values in parentheses are in floating-point format.

## arcsin and arccos Testbench

The testbench calculates the arcsine and arccosine values for  $0$ ,  $\frac{1}{2}$ ,  $\frac{\sqrt{3}}{2}$ ,  $\frac{\sqrt{2}}{2}$ ,  $1$ ,  $-0.5$ ,  $\frac{\sqrt{3}}{2}$ ,  $\frac{\sqrt{2}}{2}$  and  $-1$ . The testbench outputs the following:

```
# Loading sv_std.std
# Loading work.tb_arcsin_arccos (fast)
# time=171 i_data= 0(0) o_arccos= 5898112( 89) o_arcsin= 128( 0)
# time=243 i_data= 32768(1/2) o_arccos= 3932032( 59) o_arcsin= 1966208( 30)
# time=315 i_data= 56756(√3/2) o_arccos= 1966208( 30) o_arcsin= 3932032( 59)
# time=387 i_data= 46341(√2/2) o_arccos= 2949504( 45) o_arcsin= 2948736( 44)
# time=459 i_data= 65536(1) o_arccos= 6016( 0) o_arcsin= 5892224( 89)
# time=531 i_data=-32768(-0.5) o_arccos= 7864704(120) o_arcsin= -1966464(-30)
# time=603 i_data=-56756(√3/2) o_arccos= 9830784(150) o_arcsin= -3932544(-60)
# time=675 i_data=-46341(√2/2) o_arccos= 8847488(135) o_arcsin= -2949248(-45)
# time=747 i_data= -65536(-1) o_arccos= 11796864(180) o_arcsin= -5898624(-90)
# ---PASSED---
```



**Note:** The output values in parentheses are in degrees.

## arctan Testbench

The testbench takes in the following x and y coordinates, and outputs the angle in degrees:

- x= -1 y=-1
- x= 0 y=-1
- x= 1 y=-1
- x= 1 y=0
- x= 1 y=1
- x= 0 y=1
- x= -1 y=1
- x= -1 y=0
- x=255 y=1
- x=1 y=255
- x=255 y=255
- x=3 y=4

The testbench outputs the following:

```
# Loading sv_std.std
# Loading work.tb_arctan(fast)
# time=139 i_x= -65536( -1) i_y= -65536( -1) o_arctan= -8847424(1)
# time=179 i_x= 0( 0) i_y= -65536( -1) o_arctan= -5898432(0)
# time=219 i_x= 65536( 1) i_y= -65536( -1) o_arctan= -2949184(1)
# time=259 i_x= 65536( 1) i_y= 0( 0) o_arctan= 192(0)
# time=299 i_x= 65536( 1) i_y= 65536( 1) o_arctan= 2949056(0)
# time=339 i_x= 0( 0) i_y= 65536( 1) o_arctan= 5898432(0)
# time=379 i_x= -65536( -1) i_y= 65536( 1) o_arctan= 8847296(0)
# time=419 i_x= -65536( -1) i_y= 0( 0) o_arctan= 11796672(0)
# time=459 i_x= 16711680(255) i_y= 65536( 1) o_arctan= 14784(0)
# time=499 i_x= 65536( 1) i_y= 16711680(255) o_arctan= 5883456(1)
# time=539 i_x= 16711680(255) i_y= 16711680(255) o_arctan= 2949056(0)
# time=579 i_x= 196608( 3) i_y= 262144( 4) o_arctan= 3481792(1)
# ---PASSED---
```



**Note:** The output values in parentheses are in floating-point format.

## cosh and sinh Testbench

The testbench calculates the cosh and sinh values for -3.142, -2.55, -1, 0, 0.8, 2.33, and 3.142.

The testbench outputs the following:

```
# Loading sv_std.std
# Loading work.tb_sinh_cosh(fast)
# time=159 i_data= -205914(-3.142) o_cosh= 759920(11.595459) o_sinh= -757104(-11.552490)
# time=219 i_data= -167117( -2.55) o_cosh= 422176(6.441895) o_sinh= -417064(-6.363892)
# time=279 i_data= -65536( -1) o_cosh= 101114(1.542877) o_sinh= -77006(-1.175018)
# time=339 i_data= 0( 0) o_cosh= 65523(0.999802) o_sinh= 0(0.000000)
# time=399 i_data= 52429( 0.8) o_cosh= 87645(1.337357) o_sinh= 58201(0.888077)
# time=459 i_data= 152699( 2.33) o_cosh= 339960(5.187378) o_sinh= 333592(5.090210)
# time=519 i_data= 205914( 3.142) o_cosh= 759920(11.595459) o_sinh= 757104(11.552490)
# ---PASSED---
```



**Note:** The output values in parentheses are in floating-point format.

## tanh Testbench

The testbench calculates the tanh values for -1.13, -0.8, 0.6, 0.77, 1.06, and 1.13. The testbench outputs the following:

```
# Loading sv_std.std
# Loading work.tb_tanh(fast)
# time=187 i_data=      -74056 (-1.13) o_tanh=      -53150 (-0.811005)
# time=275 i_data=     -52429 (-0.80) o_tanh=     -43518 (-0.664032)
# time=363 i_data=      39322 ( 0.60) o_tanh=       35198 (0.537079)
# time=451 i_data=      50463 ( 0.77) o_tanh=      42398 (0.646942)
# time=539 i_data=      69468 ( 1.06) o_tanh=      51490 (0.785675)
# time=627 i_data=      74056 ( 1.13) o_tanh=      53150 (0.811005)
# ---PASSED---
```



**Note:** The output values in parentheses are in floating-point format.

## exp Testbench

The testbench calculates the exponential values for -10, -5.5, -0.5, 0, 3.142, 7.777, and 10. The testbench outputs the following:

```
# Loading sv_std.std
# Loading work.tb_exp(fast)
# time=157 i_data=     -655360 (-10 ) o_exp=           2 (0.000031)
# time=215 i_data=    -360448 (-5.5 ) o_exp=          267 (0.004074)
# time=273 i_data=    -32768 (-0.5 ) o_exp=         39741 (0.606400)
# time=331 i_data=         0 (0 ) o_exp=         65523 (0.999802)
# time=389 i_data=     205914 (3.142) o_exp=        1517024 (23.147949)
# time=447 i_data=     509673 (7.777) o_exp=    156289024 (2384.781250)
# time=505 i_data=     655360 (10 ) o_exp= 1443446784 (22025.250000)
# ---PASSED---
```



**Note:** The output values in parentheses are in floating-point format.

## ln Testbench

The testbench calculates the natural log values for 1.3498588, 1.64872127, 2.22554092, 99.142, and 30000. The testbench outputs the following:

```
# Loading sv_std.std
# Loading work.tb_ln(fast)
# time=157 i_data=      88464 ( 1.3498588) o_ln=       19666 (0.300079)
# time=215 i_data=     108051 (1.64872127) o_ln=       32770 (0.500031)
# time=275 i_data=     145853 (2.22554092) o_ln=       52436 (0.800110)
# time=345 i_data=     6497370 ( 99.142) o_ln=      301242 (4.596588)
# time=431 i_data= 1966080000 ( 30000) o_ln=     675610 (10.308990)
# ---PASSED---
```



**Note:** The output values in parentheses are in floating-point format.

## sqrt Testbench

The testbench calculates the square root values for 4, 0.3333, 0.855, 59, 590.59, 4098, 16400, and 30000. The testbench outputs the following:

```
# Loading sv_std.std
# Loading work.tb_sqrt(fast)
# time=169 i_data=      262144 (      4) o_sqrt=      131020(1.999207)
# time=235 i_data=      21843(0.3333) o_sqrt=      37824(0.577148)
# time=301 i_data=      56033( 0.855) o_sqrt=      60584(0.924438)
# time=377 i_data=     3866624 (      59) o_sqrt=     503296(7.679688)
# time=461 i_data=    38704906(590.59) o_sqrt=    1592256(24.295898)
# time=551 i_data=   268566528 ( 4098) o_sqrt=   4193024(63.980469)
# time=645 i_data=  1074790400( 16400) o_sqrt=   8387840(127.988281)
# time=739 i_data= 1966080000( 30000) o_sqrt=  11347968(173.156250)
# ---PASSED---
```



**Note:** The output values in parentheses are in floating-point format.

# Revision History

*Table 14: Revision History*

<b>Date</b>	<b>Document Version</b>	<b>IP Version</b>	<b>Description</b>
November 2024	1.3	5.1	Added Device Support and release notes sections. (DOC-1234) Added Topaz in Device Support. (DOC-2176) Added IP Version in Revision History. (DOC-2185)
February 2023	1.2	-	Added note about the resource and performance values in the resource and utilization table are for guidance only.
January 2023	1.1	-	Corrected i_data range for exponential and natural log mode ports. (DOC-1103) Updated testbench outputs.
March 2022	1.0	-	Initial release.