

PathWave
FPGA 2021

PathWave FPGA User Guide

Notice

© Keysight Technologies, Inc. 2021

1400 Fountaingrove Pkwy., Santa Rosa, CA 95403-1738, United States

All rights reserved.

No part of this documentation may be reproduced in any form or by any means (including electronic storage and retrieval or translation into a foreign language) without prior agreement and written consent from Keysight Technologies, Inc. as governed by United States and international copyright laws.

Restricted Rights Legend

If software is for use in the performance of a U.S. Government prime contract or subcontract, Software is delivered and licensed as "Commercial computer software" as defined in DFAR 252.227-7014 (June 1995), or as a "commercial item" as defined in FAR 2.101(a) or as "Restricted computer software" as defined in FAR 52.227-19 (June 1987) or any equivalent agency regulation or contract clause.

Use, duplication or disclosure of Software is subject to Keysight Technologies' standard commercial license terms, and non-DOD Departments and Agencies of the U.S. Government will receive no greater than Restricted Rights as defined in FAR 52.227-19(c)(1-2) (June 1987). U.S. Government users will receive no greater than Limited Rights as defined in FAR 52.227-14 (June 1987) or DFAR 252.227-7015 (b)(2) (November 1995), as applicable in any technical data.

Table of Contents

PathWave FPGA User Guide	8
Key Features	8
Getting Started	8
Working with PathWave FPGA	8
Getting Started	9
Contents	9
Release Notes	9
2021 Hotfix	9
2021 Highlights	9
2020 Update 1.1 Highlights	10
2020 Update 1.0 Highlights	10
2020 Highlights	10
2019 Highlights	10
2018 Highlights	10
BSP Compatibility	10
Known Issues	11
System Requirements	14
Recommended Hardware Configurations	14
Software Compatibility with PathWave FPGA	14
Summary of HDL Language Support	15
Installation	15
Obtain PathWave FPGA License File	15
Download PathWave FPGA Installer	16
Install PathWave FPGA	16
Setup PathWave FPGA License	16
Node-locked License	16
Floating License	16
Launch PathWave FPGA	16
Licensing	16
About PathWave FPGA Licenses	16
Supported licensing modes	16
The Licensing Process	17
Troubleshooting	17
User Guide	18
Contents	18
Overview	18
GUI Overview	22
Keyboard and Mouse Shortcuts	23
Basic Controls	24
Adjusting the View	24
Manipulating Items	24
Undo and Redo	24
Adding Blocks	25
Connecting Ports and Interfaces	26
Connecting an Output Port to an Input Port	27
Remove and Redraw operations	28
Connecting Input Ports to a Literal Constant	30
Connection Rules	30
Ports	30
Port Size Mismatches	30
Interfaces	31
Adding and Editing Comments	32
Configuring PathWave FPGA	34
Vivado Installation Path	34
Vivado Installation Browse Button	34
IP Repositories Path List	34
IP Repositories Control Buttons	36
Theme Checkbox	36

Infer Interfaces Checkbox	36
Designing your FPGA Logic	36
Creating a New Sandbox Project	36
Sandbox Project Directory Structure	37
Source Control	37
Which files should be tracked	38
Sample .gitignore file	39
Creating a New Submodule Project	39
Project Settings Dialog	40
IP Repositories Path List	40
IP Repositories Control Buttons	40
Design Interfaces	40
Keysight Standard Interfaces	41
Introduction	41
Interface Descriptions	42
Adding a Memory Map	47
Adding a Register Bank	48
How to Create and Update a Register Bank	48
Register Bank Operation	51
Configuring Submodule Interfaces	51
Interface List	51
Component Preview	52
Interface Control Buttons	52
Name and Description	52
Interface Role	52
Category	52
Parameters	52
Optional Ports	52
Synchronous Properties	52
OK and Cancel Buttons	53
Changes to the Sandbox	53
Deciding the Address Width of an Interface	53
Addressing Information Section	54
Expand/Collapse Button	54
Exceeding the available address space	55
Registering Sandbox Interfaces	57
Symbol Names	58
Symbol Name of a Memory Mapped Interface	58
Symbol name of a Register	58
Symbol names for submodule interfaces	59
IP Catalog	59
Customizing the IP Catalog window	60
Customizing which columns are displayed	60
Customizing how the IP is grouped	61
IP icon color coding	63
PathWave FPGA IP Repository	63
Basic IP blocks	66
Connectors	84
Math	87
DSP	102
Memory	124
DSP Library IP	131
IP Repositories	164
Imported User IP	164
Importing an HDL file with Dependencies	167
Importing an HDL file without Dependencies	167
Vivado XCI (Xilinx Core Instance)	168
Invoking Vivado IP tool	168
Importing a Vivado XCI File	170
PathWave FPGA Submodule	171
Naming Conventions	171
Reserved Words	172

Name Collisions	172
IP Catalog Level.....	172
Design Canvas Level	173
Exception.....	173
Workarounds.....	173
Building your FPGA Logic.....	174
Generating the Bit File.....	174
Synthesizing and Implementing your Design inside of PathWave FPGA.....	174
Different FPGA Build options	176
Monitoring the Build	176
Exploring the Build Output.....	177
Building your Design using Vivado.....	177
Generating a Vivado Project	178
Troubleshooting	178
Drive mapping remaining after build completion	178
Generated project synthesis fails because paths are too long	178
Simulating your FPGA Logic.....	179
Simulation Testbench Design.....	179
Simulation Testbench.....	179
Simulating PathWave FPGA Standard Interfaces	179
Simulating AXI/AXI-lite Interfaces	179
Simulating AXI Streaming Interfaces	180
Simulating Mem Interfaces	181
Simulating Mem+ Interfaces	182
Simulating FDS Interfaces	184
Simulation Fileset	186
sources.json.....	186
Using Vivado's Simulation Flow	187
Test Bench Address Mapping.....	191
Contents of Address Mapping Files	192
Address Mapping Example.....	192
Using the Address Mapping File	195
Verilog	195
VHDL	195
Name Collisions.....	195
Advanced Features	196
Command Line Arguments.....	196
Migrating a design to a new BSP	198
Changing a Submodule Project Target Hardware	198
Debugging in Hardware	198
User Constraint Files	204
Glossary	205
IP Developers Guide	207
Generation of IP-XACT file	207
IP Repositories.....	207
IP Packager.....	207
Start IP Packager.....	208
Import to project	208
Welcome Page.....	208
New Button	208
Open Button	208
Recent Files List	208
Exit Button	209
Main Page.....	209
Menu button.....	209
File Buttons	209
Tabs Section.....	210
General Tab.....	210
Interfaces Tab.....	211
Port Mapping Tab.....	211
Physical Ports Tab.....	213

Parameters Tab	214
Enumerations Tab	215
Files Tab	217
IP Repository Manifest.....	218
Manifest Format.....	218
Tutorials.....	219
IP Packager Tutorial	219
Simple HDL done manually	219
Simple HDL done automatically	233
Parameterized HDL.....	239
Advanced IP Packaging	251
HVI Tutorial	269
Import Vivado High-Level Synthesis (HLS) generated IP	275
Create the Vivado HLS IP	275
Creating a Vivado HLS project.....	275
Implementing the IP function in C	278
Generating the synthesizable HLS IP.....	279
Using the Vivado HLS IP in PathWave FPGA	280
Importing the HLS IP into a project	280
Create a design using the HLS IP	281
Running design on FPGA	281
Power of Two Decimation Tutorial	282
Purpose of Tutorial	282
Requirements.....	282
Description of Decimator Design.....	282
Description of Test Software	283
Test Signal Description	283
Building the Bitfile	290
Running the C++ Example	290
Xilinx System Generator for DSP™ Tutorial	291
Appendix.....	303
Infer Interface Reference	303
CLOCK	303
NRST.....	304
AXIMM	304
AXILite.....	305
AXIS	306
MEM.....	306
Importing IP with Invalid IP-XACT	306
VHDL Support.....	307
Generics.....	307
Ports.....	307
Known Issues	308
Verilog Support.....	308
Parameters.....	308
Expressions	309
Known Issues	309
Legal	310
7-zip	310
bzip2	311
Doxygen	311
Inja	311
Lua	312
Qt	312
Xerces-C++	313
zlib.....	313
Apache License v2.0	313
Apache License.....	313
APPENDIX: HOW TO APPLY THE APACHE LICENSE TO YOUR WORK	315

GNU GPLv3	316
GNU GENERAL PUBLIC LICENSE	316
Preamble.....	316
TERMS AND CONDITIONS	317
0. Definitions.....	317
1. Source Code.....	317
2. Basic Permissions.....	318
3. Protecting Users' Legal Rights From Anti-Circumvention Law.....	318
4. Conveying Verbatim Copies.....	318
5. Conveying Modified Source Versions.....	318
6. Conveying Non-Source Forms.....	319
7. Additional Terms.....	320
8. Termination.....	321
9. Acceptance Not Required for Having Copies.....	321
10. Automatic Licensing of Downstream Recipients.....	321
11. Patents.....	322
12. No Surrender of Others' Freedom.....	322
13. Use with the GNU Affero General Public License.....	323
14. Revised Versions of this License.....	323
15. Disclaimer of Warranty.....	323
16. Limitation of Liability.....	323
17. Interpretation of Sections 15 and 16.....	323
How to Apply These Terms to Your New Programs	324
GNU LESSER GENERAL PUBLIC LICENSE	324
0. Additional Definitions.....	325
1. Exception to Section 3 of the GNU GPL.....	325
2. Conveying Modified Versions.....	325
3. Object Code Incorporating Material from Library Header Files.....	325
4. Combined Works.....	326
5. Combined Libraries.....	326
6. Revised Versions of the GNU Lesser General Public License.....	326

PathWave FPGA User Guide



Keysight PathWave FPGA Documentation

Keysight PathWave FPGA is a system-level FPGA development environment that allows you to create and deploy your custom hardware-acceleration directly into instruments.

Key Features

[Overview](#)

Getting Started

[User Guide](#)

[Release Notes](#)

Working with PathWave FPGA

[GUI Overview](#)

[Configuring PathWave FPGA](#)

[Creating a New Sandbox Project](#)

Getting Started

Contents

- [Release Notes](#)
- [System Requirements](#)
- [Installation](#)
- [Licensing](#)

Release Notes

This section contains information about previous and current releases.

2021 Hotfix

This is a maintenance release. It fixes a licensing error which exhibits as being unable to open multiple instances of PathWave FPGA when using floating licenses. With this fix, a workstation can open multiple PathWave FPGA instances using a single license.

2021 Highlights

This section provides a general overview of the new features present in release 2021:

- PathWave FPGA now uses Keysight standard licensing, and the Keysight PathWave Licensing Manager. The application licensed feature name is KF9000B. Existing projects may be opened with their previous application or this release. See [Licensing](#) for more detail.
- User projects may now have project-specific IP Repos.
- Updates to the [PathWave FPGA IP Repository](#)
 - New IP
 - BitCount - Count the bits in a word
 - Counter - Programmable up/down counter
 - FreqCnt - Frequency Counter
 - ConvertBitWidth/ConvertBitWidthStream - Convert bit widths of data
 - Prbs - Pseudo random bit sequence generator
 - Reorder/reorderStream - Change order of samples within a supersampled vector
 - ReshapeUp/ReshapeDown - Change the number of samples in a supersampled vector
 - Updated IP
 - DecimateBy5
 - DecimateBy5Complex
 - LO

2020 Update 1.1 Highlights

This section provides a general overview of the new features present in release 2020 Update 1.1:

- Fixed an issue where using Vivado 2020.X+ would fail when running FPGA builds.

2020 Update 1.0 Highlights

This section provides a general overview of the new features present in release 2020 Update 1.0:

- VLVN (Vendor-Library-Name-Version) is now used to avoid collisions with IP having the same name, but a different vendor, library, or version. See [Name Collisions](#).
- All IP are now shown in one IP catalog. You can customize how the IP is grouped and which IP information is displayed. See [IP Catalog](#).
- When importing Verilog IP into PathWave FPGA, most operators are now supported in port range expressions. See [Verilog Support](#).

2020 Highlights

This section provides a general overview of the new features present in release 2020:

- Verilog parameter support. See [Verilog Support](#).
- Enabling register stages at the sandbox boundary. See [Registering Sandbox Interfaces](#).

2019 Highlights

This section provides a general overview of the new features present in release 2019:

- Enabled re-targeting a project from one BSP to another. See [Migrating a design to a new BSP](#).
- Added hierarchical design support through Sub-modules. See [Creating a New Submodule Project](#).
- Added new IP to the included base IP. See [PathWave FPGA IP Repository](#).
- Parsing of IP-XACT 2009 enabled. Xilinx Vivado blocks will now use the interfaces present in the block.
- Created a tool for packaging HDL code into IP-XACT 2014. See [IP Packager](#).

2018 Highlights

This section provides a general overview of the new features present in release 2018, the first release of PathWave FPGA:

- PathWave FPGA is a graphical environment that provides a way to rapidly develop FPGA designs on Keysight Open FPGA hardware.
- An IP library is provided which includes Logic/Math, Memory, and DSP blocks that can be included in an FPGA design. Vivado IP blocks or custom HDL IP can also be imported and the port interfaces described using IP-XACT 2014.
- PathWave FPGA provides a design flow from schematic to bitfile generation with the press of a button.

BSP Compatibility

PathWave FPGA is compatible with all BSPs, but there are several minor issues.

The **M3202 3.73** release and the **M3302 3.64** release both contain a block called "Streamer32x2." Every time you load or create a project with one of these BSPs you will get an error dialog because PathWave FPGA also contains the same block. We recommend that you do one of the following to fix the issue.

- If you do not want to use the streamer block while using PathWave FPGA then follow these steps:
 - For the M3202 **delete** the folder: C:\Program Files\Keysight\M3202A BSP\R037300\bsp\ip\7k325\streamer32x2 and **delete** C:\Program Files\Keysight\M3202A BSP\R037300\bsp\ip\7k410\streamer32x2
 - For the M3302 **delete** the folder: C:\Program Files\Keysight\M3302A BSP\R036400\bsp\ip\7k325\streamer32x2 and **delete** C:\Program Files\Keysight\M3302A BSP\R036400\bsp\ip\7k410\streamer32x2
- If you do want to use the streamer block while using PathWave FPGA then follow these steps:
 - For the M3202 **delete** the folder: C:\Program Files\Keysight\M3202A BSP\R037300\bsp\ip\7k325\streamer32x2 and **move** C:\Program Files\Keysight\M3202A BSP\R037300\bsp\ip\7k410\streamer32x2 to an IP repository. This IP repository must be used in PathWave FPGA 2018, but must not be used in later versions of PathWave FPGA.
 - For the M3302 **delete** the folder: C:\Program Files\Keysight\M3302A BSP\R036400\bsp\ip\7k325\streamer32x2 and **move** C:\Program Files\Keysight\M3302A BSP\R036400\bsp\ip\7k410\streamer32x2 to an IP repository. This IP repository must be used in PathWave FPGA 2018, but must not be used in later versions of PathWave FPGA,

The **M3102A 1.35** release and the **M3202A 3.67** release build scripts contain hard-coded paths to the PathWave FPGA 2018 **k7z_generator.exe** program. This will cause a failure if PathWave FPGA 2018 is not installed. To fix this for this and future PathWave FPGA releases, do the following steps which may require administrator privileges:

- Navigate to the BSP script folder; this is typically C:\Program Files\Keysight\M3102A BSP\R013500\bsp\script for the **M3102A** or C:\Program Files\Keysight\M3202A BSP\R036700\bsp\script for the **M3202A**.
- Copy the sd_common_build.tcl file to a location other than C:\Program Files.
- Open the copy of sd_common_build.tcl file in a text editor.
- Change the line at or around line 437 from:
 set k7zGenerator {C:/Program Files/Keysight/PathWave FPGA 2018/k7z_generator.exe}
 to:
 set k7zGenerator [file join \$script_dir k7z_generator.exe]
- Copy the edited version of sd_common_build.tcl file back to its original location in C:\Program Files.
- Copy the following files from the PathWave FPGA 2020 Update 1.0 install folder (typically C:\Program Files\Keysight\PathWave FPGA 2020 Update 1.0) to the BSP script folder:
 - 7za.exe
 - k7z_generator.exe

Known Issues

- There is a known issue when using the M3xxx BSPs with Vivado 2020.2. This affects all versions of PathWave FPGA. When using these BSPs, use a different version of Vivado.

- IP block 'Streamer32x2b' requires Vivado 2018.1 minimum. Use the 'Streamer32x2' IP block with earlier versions of Vivado.
- There are issues when using the 'Streamer32x2' IP block and the DDR interface in the M3xxx BSPs with Vivado 2018.1. When using the 'Streamer32x2' IP block or the DDR interface in these BSPs, it is recommended to use a different version of Vivado.
- **Backward Compatibility**
 - In PathWave FPGA 2019 release or earlier, **MEM** interface was treated as having a **byte-addressing scheme**. From this release forward, MEM interfaces are using a word-addressing scheme. This change has the following impact to a project created with an earlier release and used with the current one:
 - if the project contained interface instances that were using the MEM interface but originating from a byte-addressing design interface (like **AXIMM**), the maximum acceptable value for the address width of the each instance has been lowered by 2 bits. This might cause the selected address width value to fall out of range. The user needs to manually adjust the value.
 - if the project contained a register bank originating from a MEM interface, the address offset difference between each register is reduced from 4 to 1.
 - In PathWave FPGA 2019 release or earlier, "TO range" ports were broken out into individual wires on the schematic. This behavior has been removed. This change has the following impact to a project created with an earlier release and used with the current one:
 - any connections on a "TO range" port will be lost.
- Using multiple monitors with different display scaling can result in issues with the PathWave FPGA UI. We recommend using the same scale factor for all monitors. Below are known issues, but there are likely others:
 - Window does not auto adjust when moving between monitors with different resolutions (e.g. 4K to 2K).
 - Title bar buttons do not respond to user interaction when moved from a 4K monitor to a non-4K monitor if text scaling set at 150% or above.
 - Window cuts off sections of the program on 4K monitors with text scaling set at 250% or above.
 - White border is present around maximized window on 4K monitors with text scaling set at 250% or above.
 - Changing display scaling while PathWave FPGA is running is not recommended and may not work correctly.
- **VHDL support**
 - The value range of an Integer datatype of a port is ignored. Directly importing such a file in PathWave FPGA will be completed successfully, however, the synthesis of any design that contains that IP will fail. A workaround is to create an IP-XACT file for the VHDL file using the IP Packager. Then, in the Physical Ports tab, modify the width to match the actual width required.
 - Some VHDL errors are ignored by PathWave FPGA when importing VHDL, but will fail during synthesis. Vivado is the authority on whether a VHDL file is valid, not PathWave FPGA.
 - For vector ports with a 'downto' range, the right boundary must be literal '0'. For a 'to' range, the left boundary must be literal '0'.
 - Constants or datatypes imported from another package cannot be used in the entity declaration.

- When Kactus2 is used for creating IP-XACT for a VHDL file, the VHDL entity declaration must end with "end <entity_name>" and not "end entity."
- Arrays are not supported. They may or may not load into the schematic properly, but they will not build properly.
- **Verilog support**
 - Importing Verilog IP into PathWave FPGA has a number of known limitations. It is recommended that you create IP-XACT for any Verilog IP that does not meet the following conditions. Note that only module declarations, port and parameter definitions and 'endmodule' are checked. A violation of the following conditions will produce a "Syntax Error" message when importing Verilog IP:
 - Module declarations must include at least one port definition.
 - Ports and parameters cannot have the same name differing only by case (e.g. "myPort" and "myport").
 - Tasks and functions are not supported because their ports are misinterpreted as part of the module's interface.
 - Output registers cannot be assigned an initial value in the same statement where it is defined, such as "output reg myReg = 0;"
 - Definition of port attributes is not supported, such as "(* attribute definition *) input portName,".
 - Parameters and port definitions in a module declaration may not be conditionally included using `ifdef / `endif statements and they cannot use any preprocessor variables.
 - Expressions are limited to 32-bit signed integers. For example, "'hFFFF_FFFF" is treated as -1 instead of 4294967295.
 - Size constants in expressions are ignored. For example, "4'd65" is treated as 65 instead of being truncated to 1.
 - Arrays will fail to parse and will not load.
- Arrays are not supported in ipxact, but may load without giving any errors.
- Literals are restricted to 64 bits in this release. A '1' in the uppermost bit of the 64 bits can be represented with a hexadecimal or binary representation, or a negative decimal.
- UNC paths are not supported for building FPGA bits.
 - A UNC path can be mapped to a windows drive for building, but this is discouraged due to slow FPGA build times on remote file systems.
- If you run into intermittent licensing errors using a network license server, it could be because of a short timeout. Increasing the environment variable FLEXLM_TIMEOUT to 20000000 will set the timeout to be 20 seconds.
 - If licensing errors do not stop, a local node locked license will solve the issue.
- Saving and loading from a path with unicode characters is not supported.
- IP-XACT with callouts to unused HDL files can cause FPGA builds to fail.
- Using enumeration names longer than 150 characters can cause the IP Packager to crash
- For the LO5_DC and LO5_UC library IP blocks, the tunable range for the LO frequency is limited to $f/f_s = \pm 0.4$.

System Requirements

You must ensure that your system meets the following requirements before installing PathWave FPGA.

- Administrator privileges
- Operating system that has the most recent updates and Service Packs
- License File (or Authorization Codes, or token if evaluating) or internet access

Recommended Hardware Configurations

Category	Practical Minimums	Recommended
Operating System	Windows 10, 64-bit	Windows 10, 64-bit
Hard disk	10 GB free space	100 GB free space
RAM	4 GB RAM	16 GB RAM and above
Display	1280 x 720	1920 x 1200
Software Security	USB hardware key	Wired LAN, or Wireless LAN
Test Instrument Interface	Not required	LAN
Touch User Interface	N/A	Not supported

Software Compatibility with PathWave FPGA

The following table summarizes PathWave FPGA compatibility with various versions of other software applications. However, for the latest vendor information, licensing, and downloads, please contact each vendor directly.

Vendor	Software / Feature	Release Officially Supported	May work, but not supported	Release Explicitly not-supported
<u>Xilinx</u>	Vivado, debugging, compilation of bit images.	Vivado 2017.3		prior to Vivado 2017.3
<u>CMake</u>	CMake to support to enable FPGA bit file verification	3.9 or later		prior to 3.9
<u>Kactus2</u>	We recommend using the PathWave FPGA <u>IP Packager</u> instead of Kactus 2.	3.6 or later	3.5 (note, there is a workaround documented when using parameterized HDL)	3.4
<u>Microsoft</u>	Visual Studio C++ to enable FPGA bit file verification	2017	Other versions	

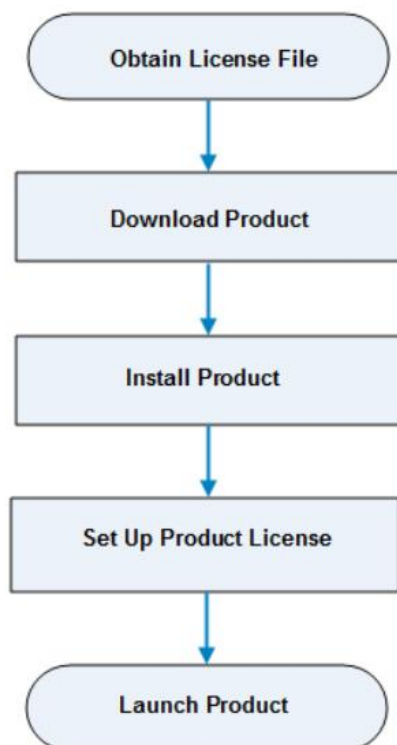
Summary of HDL Language Support

Standard	Release Officially Supported	May work, but not supported	Release Explicitly not-supported
IP-XACT	IEEE 1685-2014 , IEEE 1685-2009		
Verilog	IEEE 1364-2005		
VHDL	IEEE 1076-2002 (VHDL 2002)		IEEE 1076-2008 (VHDL 2008)

Newer versions of Xilinx Vivado might be required for Keysight Instruments (BSPs). Consult the instrument product manual for specific requirements.

Installation

PathWave FPGA can be installed on a computer running Windows by downloading the PathWave FPGA install file from <http://www.keysight.com/find/pathwave-fpga>. For the system requirement details, refer to [System Requirements](#).



Obtain PathWave FPGA License File

PathWave FPGA requires a license to run. You can either apply for an [Evaluation](#) or a [Purchased](#) license. Once the license request is approved, a license file (with .lic extension) is

sent as an email attachment. Save this file on your computer, it will be used when you run the **Keysight PathWave License Manager**.

Download PathWave FPGA Installer

Click <https://www.keysight.com/find/pathwave-fpga> to download the installer.

Install PathWave FPGA

To install PathWave FPGA, you must have system administrator privileges. Run the downloaded installer and follow the guided tour to complete the installation. If you want to do a silent install, run the installer executable from the command line as **Administrator** and use the "**--mode unattended**" command line option.

Setup PathWave FPGA License

At the end of installation, the **License Setup Wizard** starts automatically after detecting that you do not have a valid license to start PathWave FPGA. If you choose to skip the license setup, you can complete the process later by clicking **Start > Programs > Keysight PathWave License Manager > Keysight PathWave License Manager**.

Node-locked License

To setup a counted license, select the **Add a License File** option and follow the guided tour to complete the license setup process. In case of a USB dongle, attach the dongle to the USB port and invoke the **PathWave License Manager** to complete the setup process.

Floating License

To setup a floating license, select the **Specify a Remote License Server** option and follow the guided tour to complete the license setup process. Consult your license administrator for the network path of the license server.

Launch PathWave FPGA

To run PathWave FPGA, go to the **Start** menu and choose **Programs > Keysight PathWave FPGA <release_number> > Keysight PathWave FPGA <release_number>**.

Licensing

This chapter explains in more detail how to install PathWave FPGA licenses. It contains the following sections:

- About PathWave FPGA Licenses
- The Licensing Process
- Troubleshooting

About PathWave FPGA Licenses

PathWave FPGA requires one license per desktop client. PathWave FPGA releases 2021 and newer require a different license than PathWave FPGA 2020 Update 1.1, or older.

Supported licensing modes

The following types of licenses are supported:

Commercial licenses:

- Node-Locked, perpetual and 6, 12, 24, and 36 months, subscription.
- USB Portable, perpetual and 6, 12, 24, and 36 months, subscription.
- Floating/Networking, perpetual and 6, 12, 24, and 36 months, subscription.
- Transportable, perpetual and 6, 12, 24, and 36 months, subscription.

Trial licenses:

- 90 days Node-locked.

The Licensing Process

The Keysight licensing process uses the following steps:

1. Purchase and fulfillment

For most Keysight licensed product options, your entitlement certificate is sent to you as a PDF attachment via email immediately after your purchase. In some cases, you receive a paper copy of your certificate with your purchased product. The licensed product options may be software products or upgraded features of an instrument.

2. Getting a license

Using the entitlement certificate you received when you ordered, you can request your licenses on the [Keysight Software Manager](#) web site. To do this, you'll need to choose a host instrument or PC, and provide its identifying information (the Host ID) when you request your licenses. Once you begin the process, Keysight Software Manager will guide you step by step through requesting your licenses and you will receive the license files via email.

You might need to create a *myKeysight* login when you first go to the Keysight Software Manager site, and you will need to log in anytime you go to the site.

3. Installing your license

To enable the licensed software, after you receive a license file from Keysight Software Manager, you must install it on your instrument or computer or on a central licensing server accessible from your instrument or computer.

To install the license:

1. Install PathWave FPGA.
2. Use Keysight PathWave License Manager to install your license. PathWave License Manager is installed with PathWave FPGA, if not already installed. The installation process is described in the email that comes with your license. Previous versions of PathWave FPGA used a different Keysight license manager. Keysight recommends that you do not uninstall any previous Keysight license manager so that your previous Keysight products continue to function. After installing a PathWave FPGA license, it will appear in the PathWave License Manager status tab, with a Feature name of KF9000B.

Detailed documentation for administrators is available at <http://www.keysight.com/find/licensingdoc>.

Troubleshooting

By default, PathWave License Manager 2.3 saves its log files in C:\ProgramData\Keysight\Licensing\Log

User Guide

PathWave FPGA is Keysight's "Open FPGA" development environment. PathWave FPGA provides a complete FPGA design flow from design creation to gateway deployment to HW/gateway verification.

Contents

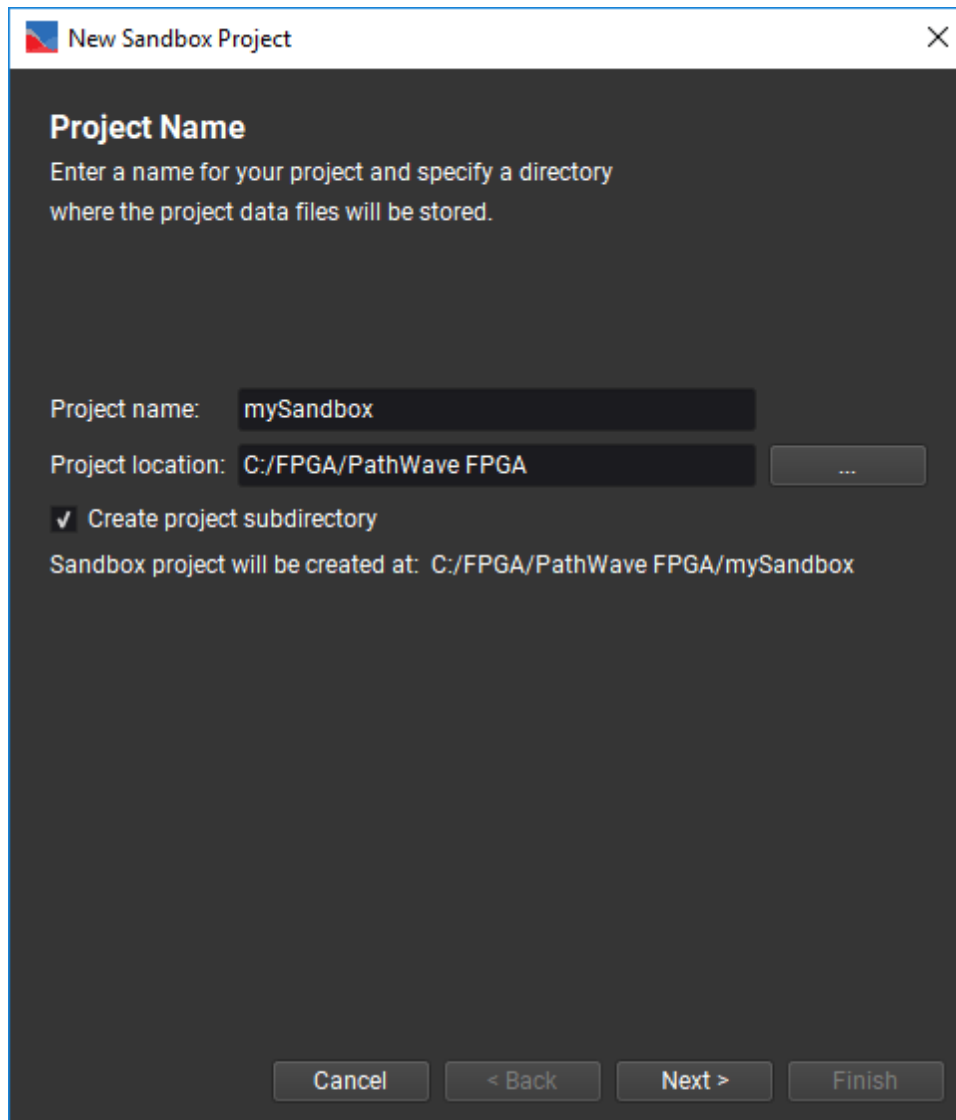
- [Overview](#)
- [GUI Overview](#)
- [Configuring PathWave FPGA](#)
- [Designing your FPGA Logic](#)
- [Building your FPGA Logic](#)
- [Simulating your FPGA Logic](#)
- [Advanced Features](#)
- [Glossary](#)

Overview

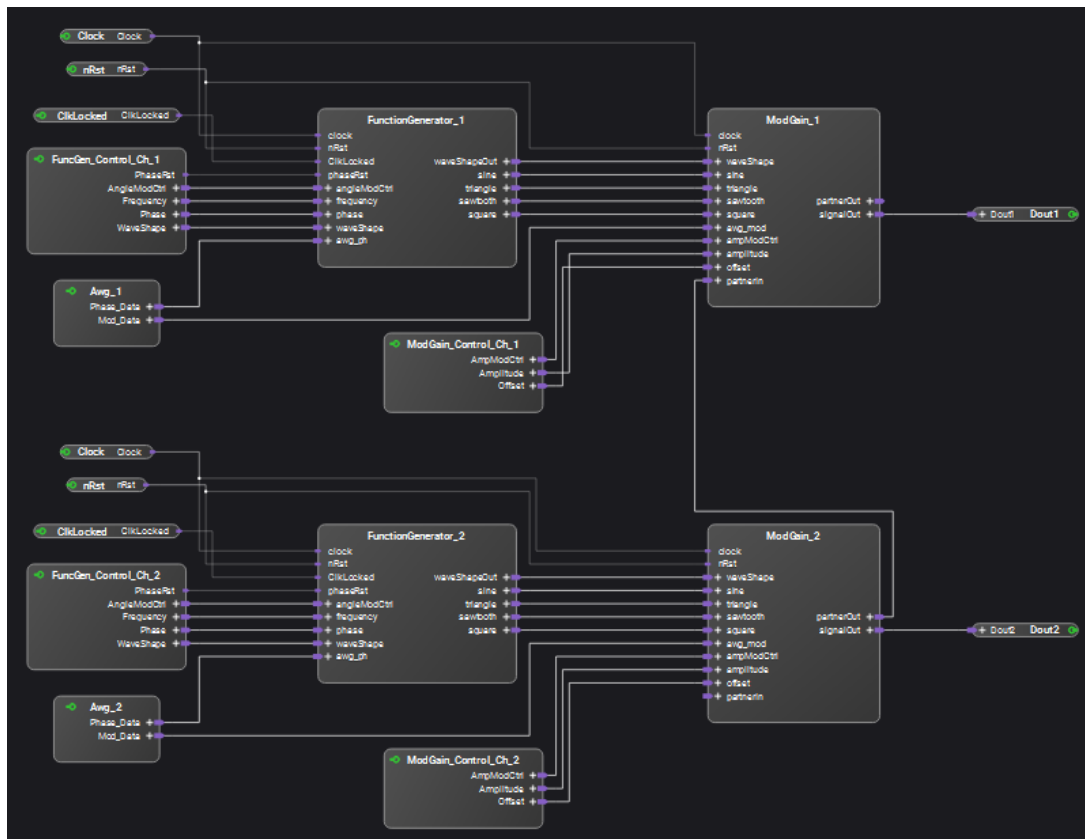
PathWave FPGA is a graphical environment that provides a way to rapidly develop FPGA designs on Keysight Open FPGA hardware. An IP library is provided which includes Logic/Math, Memory, and DSP blocks that can be included in an FPGA design. Vivado IP blocks or custom HDL IP can also be imported and the port interfaces described using IP-XACT 2014. PathWave FPGA provides a design flow from schematic to bitfile generation with the press of a button.

To get started, follow the PathWave FPGA design flow:

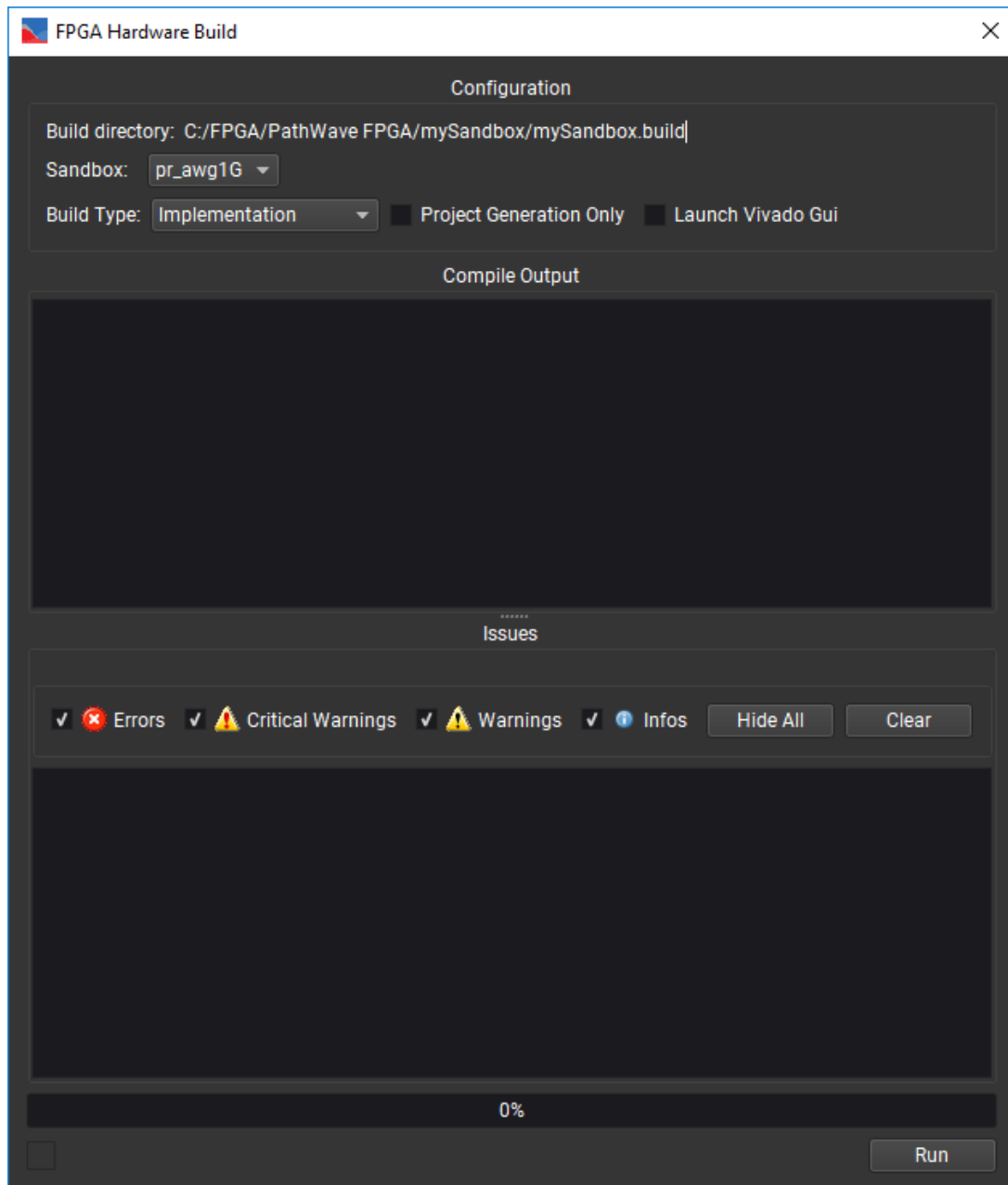
1. Start PathWave FPGA
2. Create a new project with the PathWave FPGA New Project Wizard



3. Modify the default FPGA template design by importing Vivado IP, HDL IP, or by using the PathWave FPGA IP library.

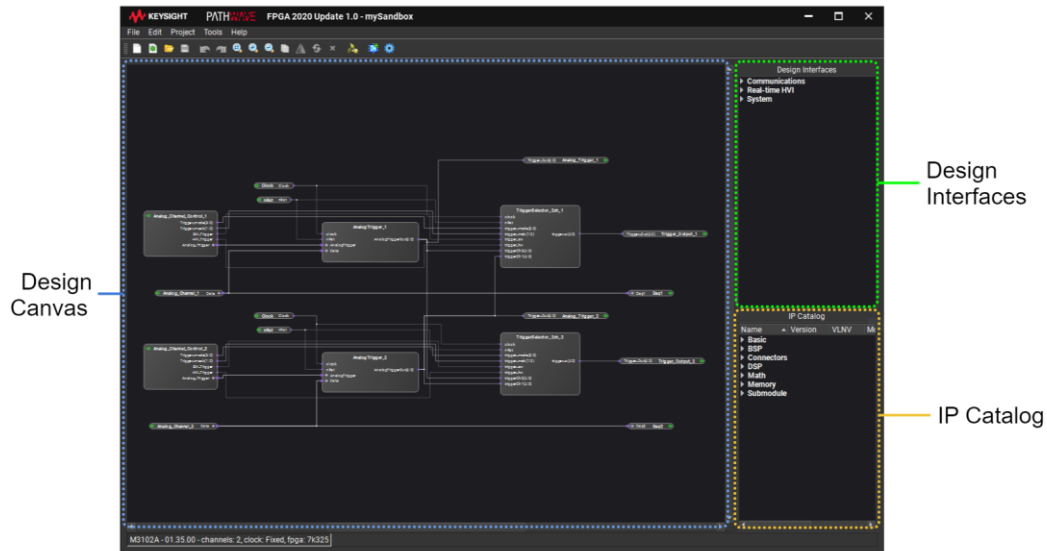










4. Compile the design into a bit image











5. Deploy your design using the instrument driver or the BSP programming API

GUI Overview



Menu/Icon/Pane	Description
File	Includes options to create a new project, open an existing project, save a project, close a project, and exit.
Edit	Includes actions to manipulate the design canvas such as undo, redo, select all, and copy.
Project	Includes an option to generate FPGA firmware and to import external blocks.
Tools	Includes the IP Packager , Vivado IP tool and configure settings.
Help	Includes link to product documentation, license, and product related information.
	Create a new sandbox project.
	Create a new submodule project.
	Open an existing project.
	Save the project.
	Undo the last operation.
	Redo the last operation that was undone.
	Fit schematic in window.
	Zoom in.

Menu/Icon/Pane	Description
	Zoom out.
	Copy.
	Flip.
	Redraw connections.
	Remove.
	Launch the Vivado IP tool.
	Add external block.
	Generate the firmware for the project.
Design Interfaces	Design Interfaces are responsible for communication between the internally configurable FPGA part (the FPGA customizable space, which a user can edit) and the rest of FPGA.
IP Catalog	IP Catalog contains the building blocks, built-in or custom , for the design canvas.

Keyboard and Mouse Shortcuts




This topic lists the operations that can be performed using keyboard and mouse shortcuts.

Function	Key Code
Add/remove item from selection	Ctrl + Left click
Abort current action	Esc
Remove selected items	Delete
Redraw connections	Ctrl + R
Zoom fit	Ctrl + F
Copy selection	Ctrl + C
Select all	Ctrl + A
Undo	Ctrl + Z
Redo	Ctrl + Y
New project	Ctrl + N
Open project	Ctrl + O
Save project	Ctrl + S

Function	Key Code
Close project	Ctrl + F4
Exit	Alt + F4

Basic Controls


Adjusting the View

Operation	Keyboard	Mouse
 Zoom In	Ctrl++	Ctrl + Mouse wheel up
 Zoom Out	Ctrl+-	Ctrl + Mouse wheel down
 Zoom Fit	Ctrl+F	
Pan		Alt + Mouse click and drag

Manipulating Items

To move an item, left-click on the item and drag it to a different location. Connections are routed automatically and can't be moved manually.

To select an item, left-click on the item. To select multiple items, left-click on an empty space and drag to select all items in a rectangle. To add or remove individual items from the selection, hold the **Ctrl** key and left-click an item. To select all items, press **Ctrl+A** or choose **Select All** from the **Edit** menu.

To copy a block or a selection, right-click the block or an item in the selection and choose Copy, then left-click to place the copy in the design. You can also press **Ctrl+C**, choose **Copy** from the **Edit** menu, or click the  **Copy** button on the toolbar.

Undo and Redo

To **Undo** an action, press **Ctrl+Z**, or choose **Undo** from the **Edit** menu, or click the  **Undo** button on the toolbar.

To **Redo** an action, press **Ctrl+Y**, or choose **Redo** from the **Edit** menu, or click the  **Redo** button on the toolbar.

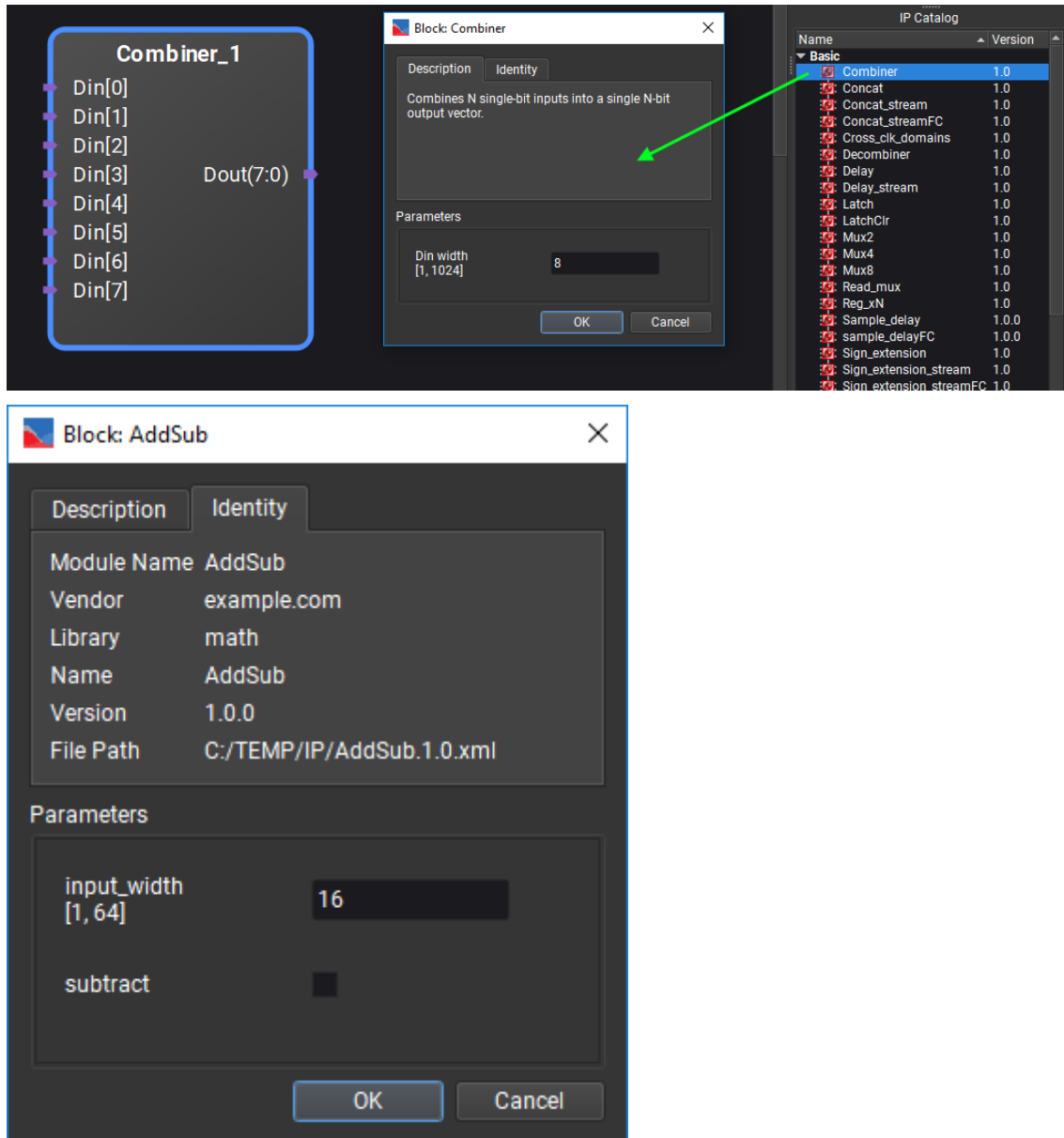
Undo is disabled after certain actions:

- Adding or removing external blocks from the IP panes. Adding or removing instances does not disable Undo.
- Adding or removing Vivado IP from the IP panes. Adding or removing instances does not disable Undo.
- Creating or removing a submodule project from the Submodules pane. Adding or removing instances does not disable Undo.
- Reloading a block
- Changing a blocks file

Adding Blocks

A hardware project is created by combining blocks from the IP Catalog on the right side of the [user interface](#). The blocks can be selected, dragged into the project, configured, and connected to other blocks in the project.

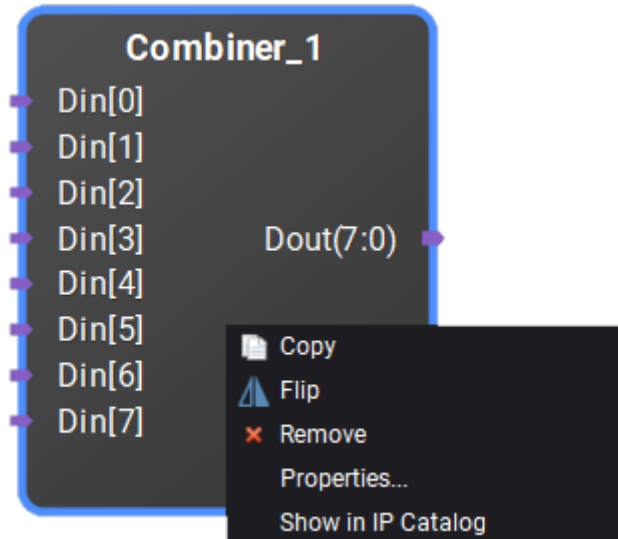
To add a block, either double-click the entry or drag it onto the Design Canvas. In the properties dialog, you can review the description and change the values of parameters.



The Identity tab shows the identifying information about the IP.

- **Module/Entity Name** is the name of the Verilog module or VHDL entity that contains the IP.
- **Vendor, Library, Name, and Version** are the unique identifier for the IP-XACT file. If the IP was imported directly from VHDL or Verilog, these fields are not shown.
- **File Path** is the location on disk of the IP-XACT or HDL file. This is not shown for IP that comes with PathWave FPGA.

Right-click a block on the Design Canvas to open the context menu.



- **Copy** creates a duplicate of the selected block.
- **Flip** swaps the ports, so that inputs are on the right and outputs are on the left.
- **Remove** deletes the block from the project.
- **Properties...** opens the configuration dialog box shown above.
- **Show in IP Catalog** selects the IP for this block in the IP Catalog.

Connecting Ports and Interfaces

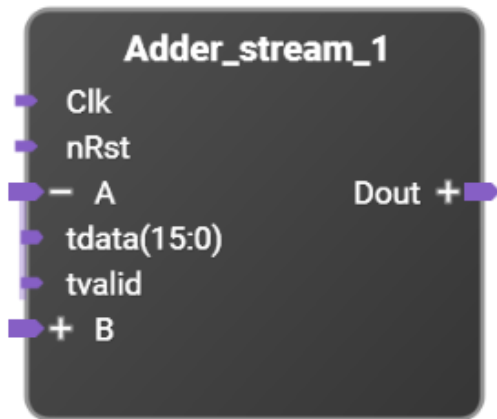
Blocks can be connected together by their ports and interfaces. An interface is defined to be a set of ports.



In the example above, this block has inputs to the left (input connectors point into the block), and outputs to the right side of the block (output connectors point out of the block).

This block has two ports (small connectors), and the other connectors are interfaces (larger connectors). The ports can represent one bit of data or a vector of bits. If the port represents a vector of bits, the size can be identified next to its name.

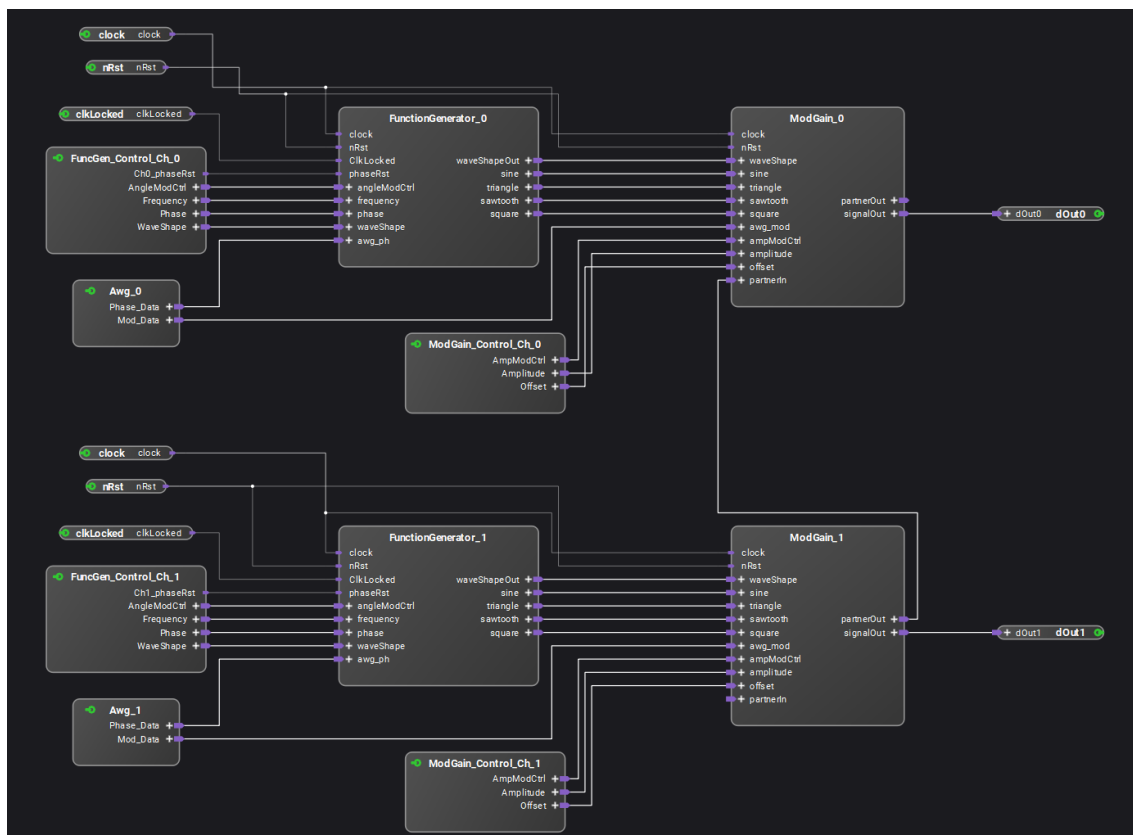
When clicking on the "+" sign of an interface, such as "A" in the above image, the internal ports of the interface appear shown below. Notice also that the "+" sign has changed to a "-" sign. Clicking on the "-" sign hides the ports again.



When the "A" interface is connected to the output of a compatible interface, all individual signals between the two interfaces are connected. If the design requires connecting an interface to an incompatible interface or individual ports on another block, the ports within the interface may be connected instead.

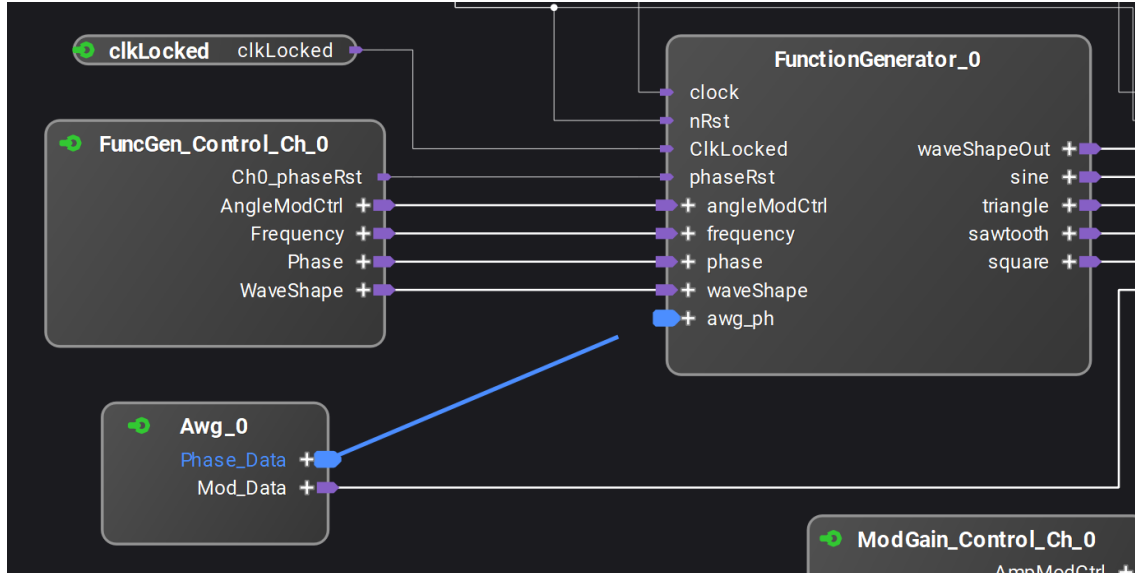
Connecting an Output Port to an Input Port

In the image below, connections are made by clicking on one port and then dragging the line from it to another suitable port. This can be done by dragging a line from an output port to an input port or by dragging a line from an input port to an output port. It may also be done by dragging a line from an input port to an existing compatible connection.

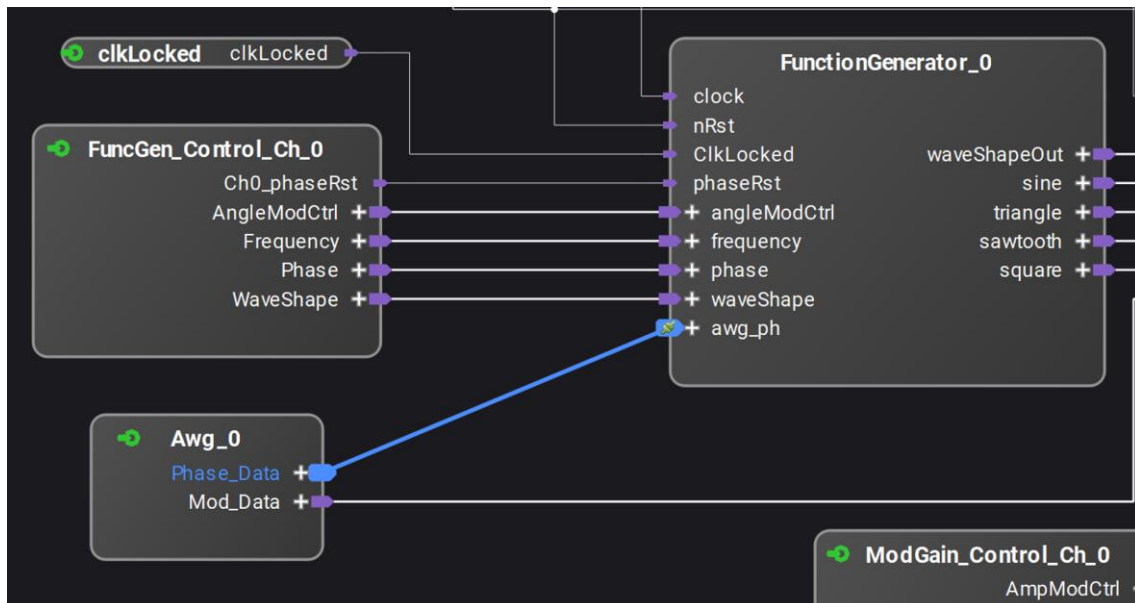


Connections can be created according to connection rules. For more information, refer [Connection Rules](#).

If a connection can be made from a connector, a new line appears from this connector to mouse and the mouse cursor changes to the axis icon as shown below. Furthermore, the possible target connectors are highlighted in blue for showing the different connection possibilities. See the input ports on the lower block "**Awg_0**" shown below.



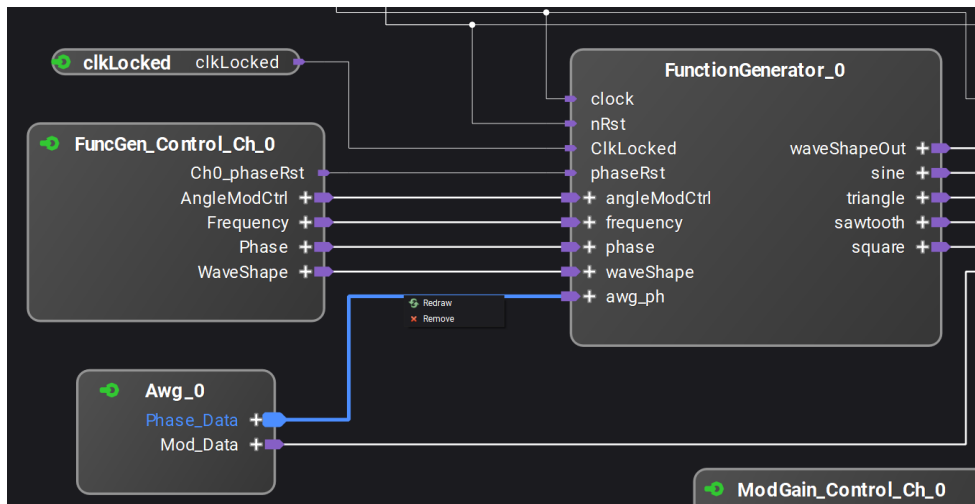
For finishing the connection, the end of the connection line is dragged by the mouse to a compatible target connector. In this case, the mouse icon changes to the green connection icon.



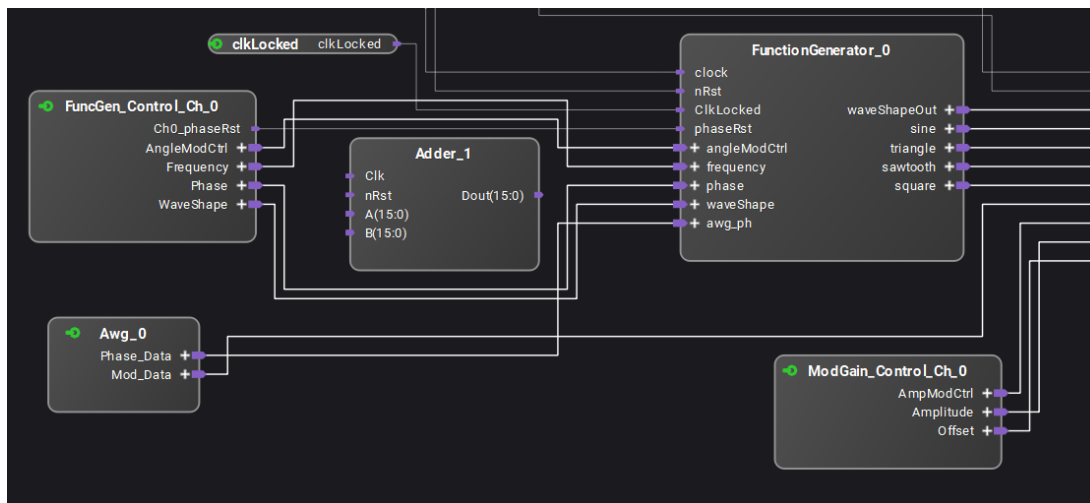
When the mouse button is released, the new connection is created.

Remove and Redraw operations

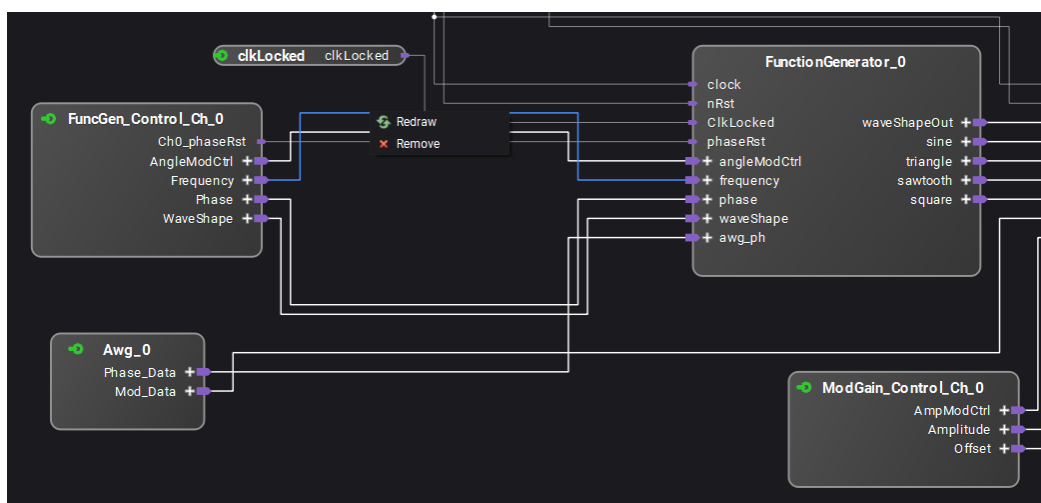
Right-click the line connecting the two ports to see two options: **Remove** and **Redraw**. Remove will delete the connecting line.



For example, add a block between the two ports. Notice the line connecting the ports is no longer straight.

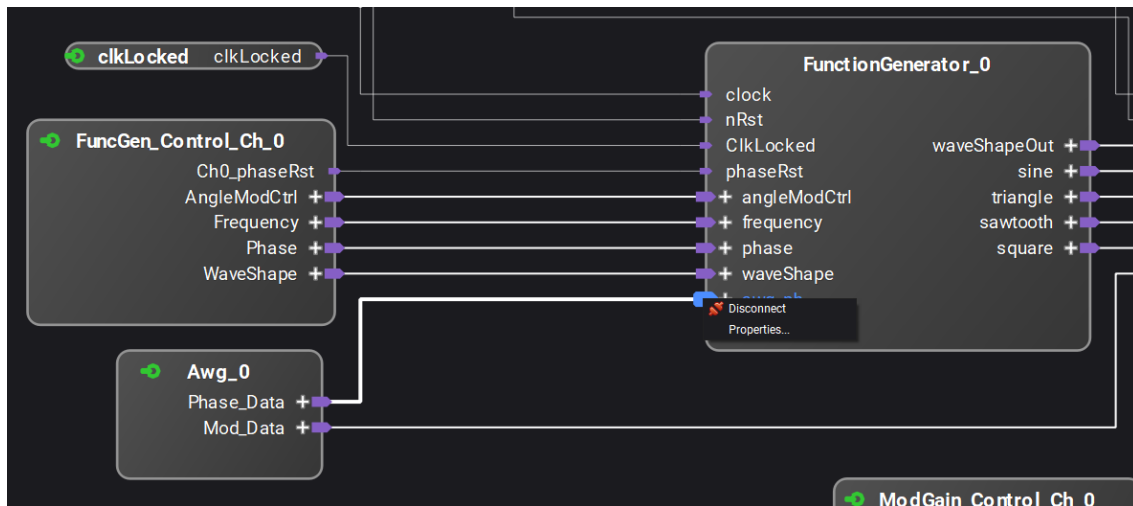


Delete the block that was just added and notice that the connecting line stays unchanged. Right-click the line and select **Redraw**. The line will be straight again.



Disconnecting a Connection

Once a connection is created, the connection can be disconnected by right-clicking on the connector, which displays the **Disconnect** option.



Connecting Input Ports to a Literal Constant

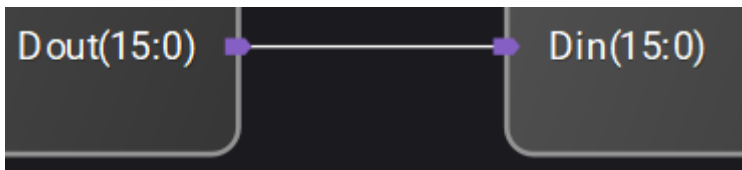
If you want to connect a input port to a constant numeric value, you should connect the port to a literal. Literals set 64-bit value constants at input ports. To insert a literal, right-click the port and select the 'Connect to literal' command. You can set the value to an integer, hexadecimal, or binary value:

- **Integer:** A integer number, negative numbers set a two's complement format. The range for valid inputs is from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807, or from $-(2^{63})$ to $2^{63} - 1$
- **Hexadecimal:** A hexadecimal number using the characters 0 - F can be entered, followed by an **h**; for example, **Ah**. The range for valid inputs is from 0h to FFFFFFFFFFFFFFFFh.
- **Binary:** Binary numbers can be added followed by a **b**, for example, **1010b**.

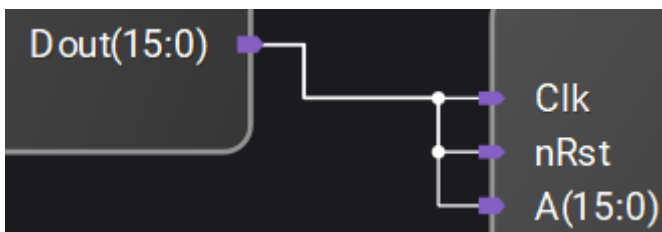
Connection Rules

Ports

There are input ports and output ports. The input ports can have only one connection to an output port. In this example, Din(15:0) has one connection.



The output ports can be connected to multiple input ports. In this example, Dout(15:0) output is connected to three inputs.



Port Size Mismatches

If a wider output port is connected to a narrower input port, then the LSBs of the output port are used to make the connection.

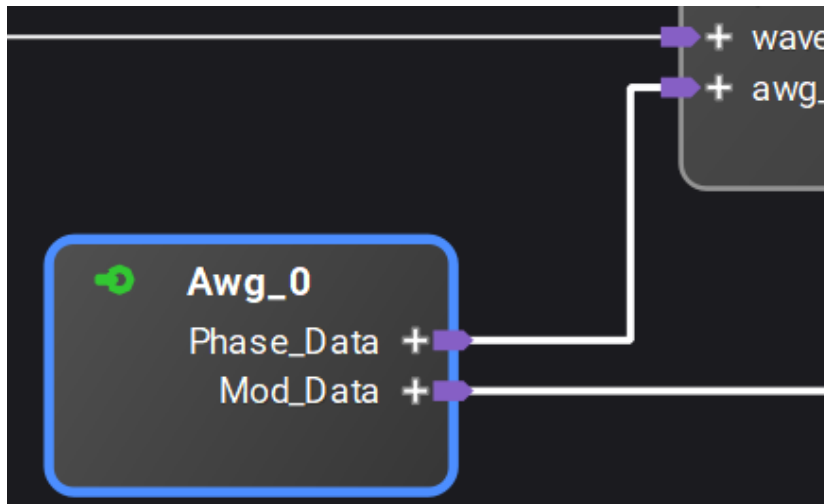
If a narrow output port is connected to a wider input port, the output port connects to the LSBs of the input port. The remaining bits of the input port are set to zero.

In general, if the smaller of the two ports has N bits, then bits N-1...0 of the output port are connected to bits N-1...0 of the input port. Any remaining output port bits are ignored, and any remaining input port bits are set to zero.

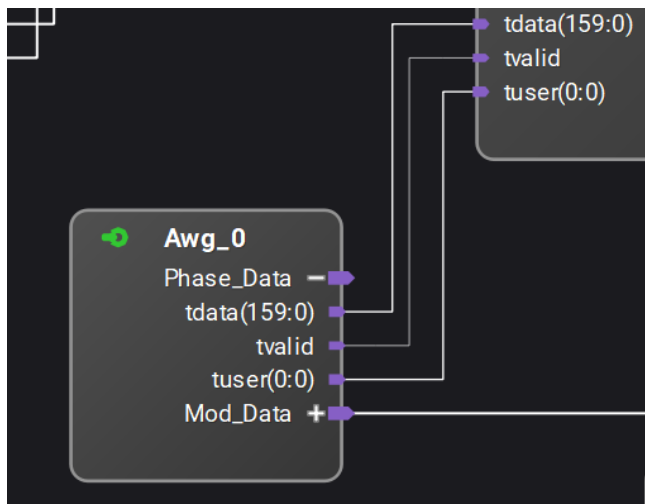
In the second example shown above, both clk and rst will be connected to Dout(0).

Interfaces

Interfaces with the same type can be connected together **as long as their data ports have the same width**. Therefore, interfaces of similar protocols can be put together with a single connection. By connecting one interface to another interface, as shown below, all the corresponding ports shown are connected. This removes the chore of having to connect each interface port as shown below.



Clicking on the "+" sign for either interface will expand the interface to show the underlying ports. When an interface is expanded, clicking the "-" sign will collapse the port back to showing just the interface name.



Connection between interface ports that have mismatched width, apart from data ports, is handled the same way as it is described in section [Port Size Mismatches](#).

Connecting Keysight interfaces to Xilinx interfaces

Keysight standard interfaces can be connected to Xilinx standard interfaces when appropriate mappings exist. i.e. a Keysight AXI4 can connect to a Xilinx AXI4. If no appropriate mapping is available, you cannot connect the interfaces.

Unconnected interface input ports

Input ports of an interface that are left without connection, either explicitly (by not connecting anything to those) or implicitly (in the case of an interface connection, where the respecting output port from the other interface is optional and not defined), will be initialized with the default value specified in the interface's specification. If a value other than the standard default value should be used for any of these ports, a literal with the desired value should be connected to that port.

Special Cases

In some cases it is not possible to define the default value as per spec definition inside PathWave FPGA. For example, the AXI4MM interface has some default values to depend on the width of the data bus.

In the following table you can find the default values that PathWave FPGA is using:

Interface Name	Port	Default value from spec.	Default value in PathWave
AXI4MM	awsiz	width of data bus in bytes as a power of 2, default assumes a bus width of 32-bits	2
AXI4MM	arsiz	width of data bus in bytes as a power of 2, default assumes a bus width of 32-bits	2

Another Special case for AXI4MM is the ID ports. If the ID port is present on a slave AXI4MM, the matching master port must have a width less than or equal to the size of the slave ID port.

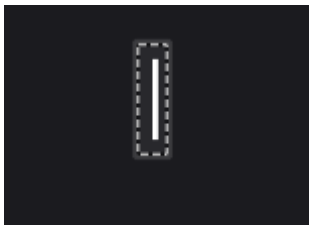
This rule is enforced so that no subtle bugs are introduced into your schematic logic.

If this does not match your expectations and the interface master does not include this port, you have to explicitly connect the unconnected input port to a literal with the desired default value.

Adding and Editing Comments

To add a comment:

1. Position the cursor within the project where the comment is to be inserted.
2. Right-click on a blank part of the canvas and select **Insert Comment...** .



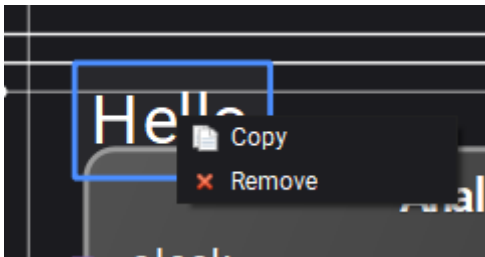
3. Add text to the comment text box.



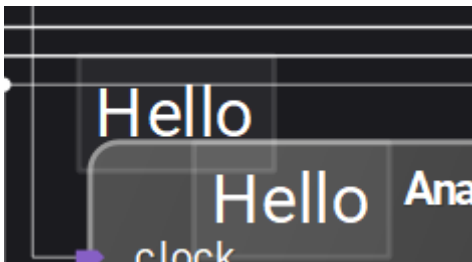
4. The comment can be moved by dragging it with the mouse. Notice the comment is in the foreground and appears above the project elements.



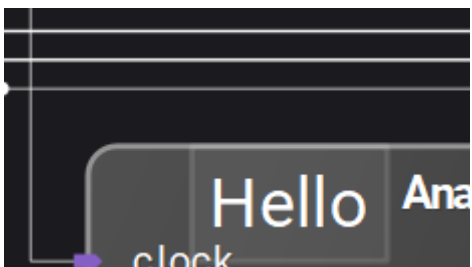
5. On right-clicking the comment, the option to copy or remove is provided.



6. Choose **Copy**, to create a duplicate comment.

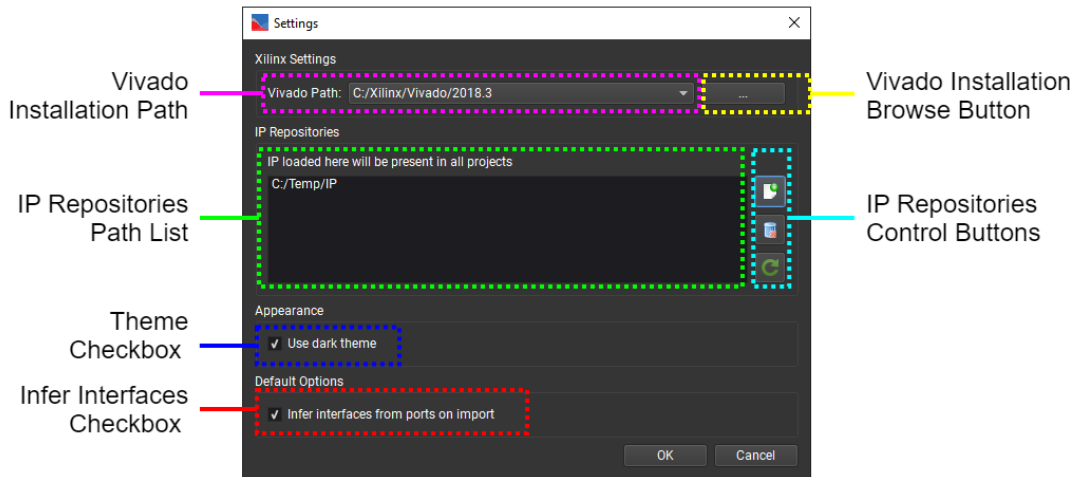


7. Choose **Remove**, to delete the comment.



Configuring PathWave FPGA

The Settings dialog provides options for configuring *PathWave FPGA*. You can specify the Vivado path, IP repositories, and the appearance of the interface. Select **Tools > Settings** to open the following dialog:



Vivado Installation Path


This drop-down list displays the installation path of the Xilinx Vivado version to be used by PathWave FPGA for the [bit file generation flow](#) as well as the [Xilinx IP Import](#). At start-up, PathWave FPGA populates the drop-down list with the Xilinx Vivado installations found on the local system. By default, the latest one is selected. The drop-down list may be used to select a different Vivado version. If the desired version is not located, the Browse Button can be used to locate a specific installation.

Vivado Installation Browse Button

Opens a browse dialog for the user to locate a Xilinx Vivado installation that was not found automatically.

IP Repositories Path List


Displays a list of directory paths, where PathWave FPGA will look for IP. To learn more information on how to create an IP repository, you can review the [IP Developers Guide](#).


The actual IP discovery process takes place either when the user clicks the  button explicitly, or when the list is updated and the settings dialog is accepted. If a project is open at the time of loading, the discovered IP will be loaded to the open project.


Currently, PathWave FPGA does not support having multiple IP with the same name. If more than one IP with the same name is encountered during a project load, PathWave FPGA will only load the first one and report an error for the others. To workaround this limitation, you can create a wrapper for your IP with name that does not conflict with any other in the project library.

The IP repositories specified here will be available to all projects. To specify IP repositories that will be available to a specific project, see [Project Settings Dialog](#).

IP Repositories Control Buttons

The  button opens a browse dialog for selecting an IP Repository location. If a location is selected, it is added to the IP Repositories Path List.

The  button removes the selected directories from the list.

The  button searches for IP inside the directories defined in the list. When IP repositories loading is completed, an informational message is displayed. In case of errors or warnings, the errors will be logged into a temporary file. The temporary file will exist until the closing of PathWave FPGA process. To regenerate the log file, repeat the loading procedure.

Theme Checkbox

To use the dark theme, check the *Use dark theme* check box.

Infer Interfaces Checkbox

When importing VHDL or Verilog User IP, interfaces can be deduced from the naming convention of the ports. Each time a new IP file is added, the user has the option to infer interfaces from the ports. The default choice is controlled by this checkbox.

Designing your FPGA Logic

- [Creating a New Sandbox Project](#)
- [Creating a New Submodule Project](#)
- [Project Settings Dialog](#)
- [Design Interfaces](#)
- [IP Catalog](#)
- [Naming Conventions](#)
- [Name Collisions](#)

Creating a New Sandbox Project

A sandbox project contains the customizable resources of the programmable FPGA of a PathWave FPGA hardware module. When selecting a target module, the project is opened with the factory settings of a standard module. The custom on-board solution is developed within this hardware project and is saved, compiled and loaded into the hardware module (the binary can be loaded into multiple identical modules).

Below are the steps to create a new sandbox project.

1. Select **File > New... > New Sandbox Project**.
2. Enter the project name.
3. Browse to select the project location.

NOTE

To place the project in a subdirectory by the same name, select the **Create project subdirectory** check box.

4. Click **Next**. If a project with the same name exists, a prompt to overwrite the project is displayed. Click **Yes** to overwrite the project.
5. Choose the **Board Support Package** for the target hardware module and click **Next**.

6. Choose a Project Template and click **Next**. A summary of the project details is displayed. Click **Finish**.
7. To save any changes you made to the project, click the **Save** icon or use the menu option.

NOTE	<p>Using the shortcut menu (right-click a block), you can perform the following operations:</p> <ul style="list-style-type: none"> • To duplicate a block, select Copy. • To flip a block horizontally, so inputs are on the right and outputs on the left, select Flip. • To redraw the connections to the block, select Redraw connections. • To remove the block, select Remove. • To view the description/properties, select Properties.
-------------	--

Sandbox Project Directory Structure

When a new project is created, a project folder with a corresponding project design file is created. This project folder will contain build output and any [Vivado XCI \(Xilinx Core Instance\)](#) IP that you have configured using PathWave FPGA. In the following example, the project created is named *myProject*. The directory structure is shown below:

- **myProject** - Project folder
 - **myProject.kfdk** - Project design file
 - **myProject.build** - Folder containing intermediate build output
 - **myProject.data** - Folder containing final build output and Vivado XCI IP
 - **bin** - Folder with the final build output
 - **myProject_<timestamp>** - Folder containing build output
 - **bitgen.log** - Vivado build log file
 - **myProject.k7z** - Program archive that can be downloaded into your FPGA
 - **myProject.spb** - Program FPGA bit file that is an older format, to supported existing instrument software for M3102A, M3202A, M3302A and associated instruments. Newer Keysight hardware will not produce this file output.
 - **VivadoIP** - Folder to contain output for Vivado XCI IP that was configured using PathWave FPGA
 - **<imported Vivado XCI>** - Folder for each Vivado XCI IP configured using PathWave FPGA
 - **submodules** - Folder to contain [submodule projects](#). The directory structure that is created is an [IP Repository](#) of the submodules defined in the project
 - **mySubmodule** - Submodule with default name
 - **mySubmodule.data** - Folder containing Vivado XCI IP

Source Control

This section will describe best practices for using a version system for tracking changes in your PathWave FPGA project.

PathWave FPGA takes some steps to work well with source control of its project files, such as using plain text and maintaining a consistent order of elements. If more than one person is

making changes to a PathWave FPGA project, the changes may need to be merged. However, automated merging of significant changes should not be considered reliable. Always check for problems by opening the project and running a synthesis build after a merge. When in doubt, manually "merge" the changes by repeating them using the PathWave FPGA GUI.

If your project includes software source code, follow the best practices as for any other software source control. The example projects that come with PathWave FPGA use CMake. In this case, CMake will place the Visual Studio solution or Unix makefiles and all build artifacts in a directory, usually with the word "build" in the name. This directory should be excluded, and most other files should be included.

Which files should be tracked

File path	Track?	
<i>projectName.kfdk</i>	Yes	The project design file should be tracked.
<i>projectName.kfdk.bak</i>	No	Backup project files are created when a project is upgraded to a newer version of PathWave FPGA, or when a project is retargeted to a different BSP. If you are tracking the project file with source control, you already have a backup and don't need these files.
<i>projectName.build/</i>	No	This directory contains intermediate build files which don't usually need to be stored.
<i>projectName.data/submodules/</i>	Yes/No	Each submodule is contained within a directory in submodules. Its directory structure mirrors that of the main project. There will never be a <i>submoduleName.build</i> directory, but there may be a <i>submoduleName.data</i> directory containing any Vivado IP used by the submodule. The same guidelines for the sandbox project directory apply for the submodule directory. In the provided .gitignore file, some patterns use ** so that they apply to both the sandbox and the submodule.
<i>projectName.data/bin/</i>	Maybe	Each successful sandbox build creates a directory in bin with the .k7z file, and sometimes other BSP-specific files. These files can be included or excluded depending on your requirements.
<i>projectName.data/VivadoIP/ip_user_files/</i>	No	This directory contains files generated by Vivado when configuring IP. These files are only created if you click Generate after configuring the IP. PathWave FPGA does not use these files because it always regenerates Vivado IP at the start of the build.

File path	Track?	
<code>projectName.data/VivadoIP/ipName/ipName.xml</code> <code>projectName.data/VivadoIP/ipName/ipName.xci</code>	Yes	<p>Each configured Vivado IP is contained in a directory inside VivadoIP. Two files in this directory are important: the <code>.xci</code> and the <code>.xml</code>. All other files should be excluded from version control.</p> <p>You may see critical warnings from Vivado during the sandbox build if the other files are missing, but the build will still complete correctly. To remove the critical warnings, launch the Vivado IP tool from within your project, select all of your IP, right click and select "Reset Output Products...". This will remove all extraneous files.</p>

Sample .gitignore file

The following file will make git ignore the files that shouldn't be tracked. Place this file in the same directory as the project file.

```
*.kfdk.bak*
*.build/
*.data/bin
**/*.data/VivadoIP/ip_user_files/
**/*.data/VivadoIP/**/*
!**.data/VivadoIP/**/*.*.xml
!**.data/VivadoIP/**/*.*.xci
```

Creating a New Submodule Project

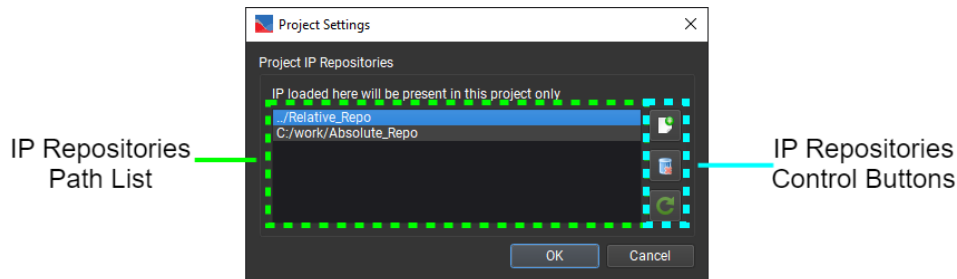
A submodule project allows you to organize your design hierarchically and reuse these designs in multiple projects.

Below are the steps to create a new submodule project.

1. Select **File > New... > New Submodule Project**, from the menu of an open sandbox project.
2. In the New Submodule Project dialog, enter the submodule project name and click **Next**.
3. Define the vendor, library, name and version (VLNV) and other properties of the submodule. This information can be modified later by selecting **Project > Properties...**
4. Click **Next**. A summary of the project details is displayed. Click **Prev** to make changes or **Finish** to save the new submodule project. See [Sandbox Project Directory Structure](#) for information about how submodule projects are saved.
5. A new instance of PathWave FPGA will be started where you can edit your new submodule.
6. In the Change Submodule Interfaces dialog, define the interfaces into and out of the submodule. See [Configuring Submodule Interfaces](#) for more information. The interfaces can be modified later by selecting **Project > Change Submodule Interfaces...**
7. Click **OK** to close the Change Submodule Interfaces dialog.
8. To save any changes you made to the project, click the **Save** icon or use the menu option.


Project Settings Dialog

The project settings dialog provides some options for configuring *your PathWave FPGA project* by adding IP repositories. Select **Project > Project Settings** to open the following dialog:




IP Repositories Path List

Displays a list of directory paths specific to the loaded project; PathWave FPGA will search for IP in these directories. To learn more creating an IP repository, see the [IP Developers Guide](#).


The actual IP discovery process takes place either when the  button is clicked, or when the list is modified and the OK button is clicked. If a project is open at the time of loading, the discovered IP will be loaded in the open project. After the IP discovery process, a dialog will report how many IP have been found, and any warnings or errors that occurred.

An IP repository path may be converted to a relative path by right-clicking on the path and selecting 'Use Relative Path' in the context menu. You may convert back to an absolute path by selecting 'Use Absolute Path'.

IP Repositories Control Buttons

The  button opens a browse dialog for selecting an IP Repository location. If a location is selected, it is added to the IP Repositories Path List.

The  button removes the selected directories from the list.

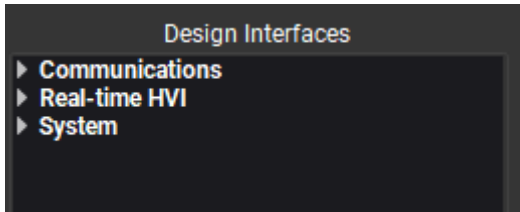
The  button searches for IP inside the directories defined in the list. When IP repositories loading is completed, an informational message is displayed. In case of errors or warnings, the errors will be logged into a temporary file. The temporary file will exist until PathWave FPGA is closed. To regenerate the log file, repeat the loading procedure.

NOTE

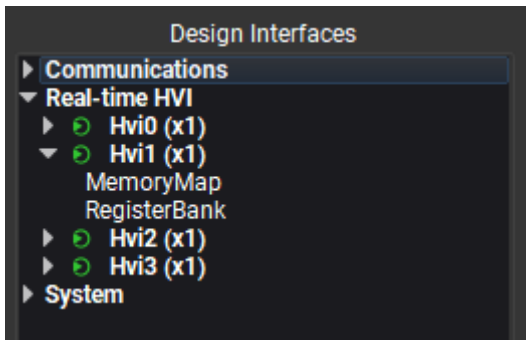
To specify IP repositories that will be available to all projects, specify the repositories in the PathWave FPGA Settings dialog (see [Configuring PathWave FPGA](#)).

Design Interfaces

To communicate between the design and what lies outside the design, i.e. the static region for sandbox designs, or some other design for submodules, you need to instantiate a design interface block from the [design interfaces pane](#). Each board support package provides a unique set of design interface blocks that are specific for the instrument. The design interface blocks are grouped based on the function of their connections to the "outside world". The interfaces of a design are collapsed, in order to show the different categories of design interfaces:



Apart from categorizing, some design interface blocks can be instantiated with different types of interfaces. For example, the interface "Hvi1" can be inserted to the schematic as a [MemoryMap](#) or connected directly to a [RegisterBank](#).



Finally, it is possible that an interface is comprised only by one port (e.g. a clock). In that case, the interface instance will only show the slot, like in the picture below:



Keysight Standard Interfaces

- [Introduction](#)
- [Interface Descriptions](#)
 - [Addressing scheme](#)
 - [Signal Types](#)
 - [Data Types](#)
 - [Data Packing/Extending](#)
 - [Polarity](#)
 - [Signal Interfaces](#)
 - [Example Usage](#)
 - [Discussion of Example](#)
 - [Associated Files](#)

Introduction

To facilitate connectivity between IP blocks and Design Interfaces, PathWave FPGA has standardized on a number of interfaces. IP blocks using these interfaces will be easier to interconnect and to connect to PathWave FPGA library blocks and Design Interfaces.

Interface Descriptions

The following is a brief description of the standard interfaces PathWave FPGA supports. Note that this is only a brief description of each interface and is not meant to be a complete description. Some interfaces (e.g. the AXI family) include optional signals that can be included or omitted in particular implementations depending on the design requirements. This allows the user to tailor the complexity and size of the interface while maintaining compatibility.

1. clock: A free running clock. Data is both sampled and changed on the rising edge of a clock.
2. nRst: An active low reset signal.
3. AXIMM: the industry standard, AXI4-Memory Mapped high performance bus architecture.
 - a. Includes address information. This is a byte-addressable interface, meaning that each address unit addresses 8-bits of data.
 - b. Supports data widths: 32, 64, 128, 256, 512, 1024 bits.
 - c. Supports burst (high performance) transfers.
 - d. Supports bi-directional flow control.
4. AXILite: the AXI4-Lite bus, a lightweight version of AXIMM for simpler interfaces that don't require the performance/features of full blown AXI4.
 - a. Limited data width: 32 (preferred) or 64 (if needed).
 - b. Only single transactions supported - no data bursting.
 - c. Supports bi-directional flow control.
5. AXIS: the AXI4-Streaming interface is for streaming arbitrarily long sequences of data.
 - a. Point-to-point streams - this interface does not include address data, though optional TID, and TDEST signals allow some routing (addressing) information.
 - b. Data width is any multiple of 8 bits. Unlike AXIMM and AXILite, AXIS can support, for example, 24 bit data. The standard allows 0 bit data (TDATA is optional). An AXIS interface without data just has the control signals.
 - c. Supports optional TUSER data signals. These are extra signals that are logically attached to data samples that could be used to include auxiliary data such as triggers or data marks or timing information.
 - d. Supports merging/packing multiple data items into wider stream.
 - e. Supports bi-directional flow control.
6. mem: a very light weight Keysight proprietary interface.
 - a. Can be bi-directional.
 - b. Includes addressing. This is a word-addressable interface, meaning that each address unit addresses 32-bits of data.
 - c. Does not include back-pressure - all transactions take place in one clock cycle and can not be held off.
 - d. Has deterministic timing.
 - e. Used for HVI register access. Please see the Keysight M3601 documentation for more information on HVI.
7. vector: a multi-bit vector of signals without any signaling protocol. This might be used to connect a control register to an IP block.
8. wire: a single bit signal. This might be used for a trigger signal.

Addressing scheme

By addressing scheme, we refer to the number of data bits each address unit is addressing.

For example, in a byte-addressing scheme, each address unit addresses 8-bits of data. That means that if we store a 32-bit data value `0xabcd0123`, in a little-endian memory structure, at address b'11110000, then the memory will look like this :

address	data (8-bit)
b'11101111	<other_data>
b'11110000	0x23
b'11110001	0x01
b'11110010	0xcd
b'11110011	0xab
b'11110100	<other_data>

On the other hand, for a word-addressing scheme, each address unit addresses 32-bits of data. That means that for the same example as before, the memory will look like this :

address	data (32-bit)
b'11101111	<other_data>
b'11110000	0xabcd0123
b'11110001	<other_data>

In the context of PathWave FPGA, there are currently three interfaces that are using addressing: AXI4MM, AXI4Lite and MEM. The first two (AXI4MM, AXI4Lite) use a byte-addressing scheme while the MEM interface uses a word-addressing scheme.

Signal Types

There are a number of different types of signals used in a typical design. These can roughly be categorized into control signals (typically used to setup, control, and monitor a measurement), and data flow signals (the data being processed - this could be a continuous stream of data or one or more blocks of data).

The following are the various types of signals that PathWave FPGA supports:

1. Control Bus Slaves. Typically these would be register control/status blocks where the driver could read and write status and control data.
2. Control Bus Master. This is for the case where the user IP wants to communicate with external devices via the PCIe (or other host control) bus, e.g. write to other modules to control multi-module measurements.
3. Continuous Streaming Data. This is an arbitrarily long stream of continuous data, e.g. from an ADC. Since the data may not be one sample per clock, flow control is required. Alongside the data, there may optionally be some amount of sideband data. This is auxiliary data that flows along with the main signal data. It could include triggers or marker info or be used to timestamp data.
4. Block Mode Stream Data. This would be an arbitrarily long stream of discontinuous blocks of data. Each block may represent the result of some measurement or calculation, e.g. the output of an FFT. To properly interpret this data, the boundaries of each block would need to be delineated.
5. Memory Read / Write Data. Typically the FPGA will have access to off chip memory. There needs to be a way for the user IP to read and write to this memory. This interface will need to include both address and data flow, and probably needs to support burst transfers for efficiency.
6. Supersampled Data. This is a variation of #3 and #4 above where more than one sample per clock needs to be transferred.

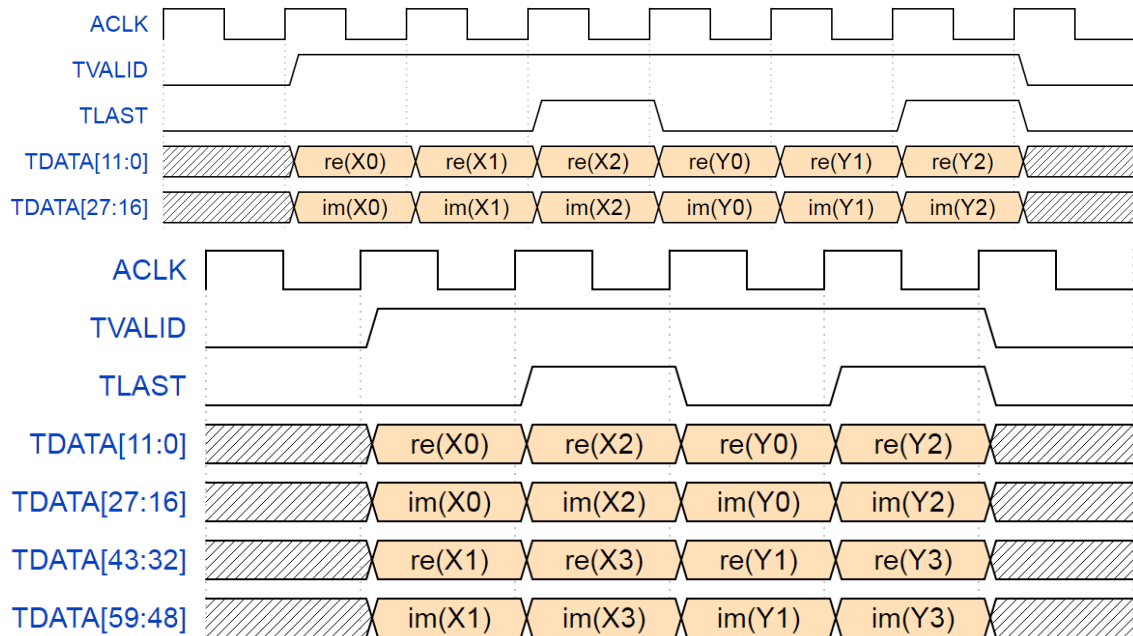
7. HVI. HVI needs an efficient, time deterministic mechanism to access control register.
8. Clock. One or more clocks. Signals change on and are sampled on the rising edge of clock.
9. Reset. One or more active low reset signals.

Data Types

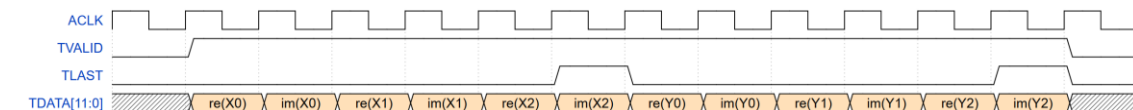
Most of the data that PathWave FPGA will be processing is likely to be fixed point (scaled ints) of varying bit widths. To facilitate interconnection of IP, limit the amount of data width conversion, and allow the use of standard interfaces, PathWave FPGA standardizes on data widths that are an integral number of bytes (i.e. multiples of 8 bits). Data that is natively a different size should be padded up to the next multiple of 8 bits by padding MSBs. Unsigned quantities are zero-extended, and signed quantities are sign extended. Thus a 12 bit unsigned number would place those 12 bits as the 12 LSBs of the interface with the 4 MSBs being zero. So if the data was $X[11:0]$, the interface used would be $TDATA[15:0] = \{4'b0000, X[11:0]\}$.

The preferred format for floating point numbers in PathWave FPGA will be IEEE-754 compliant. The two supported (preferred) sizes will be binary16 (16 bits with 11 bit fraction and 5 bit exponent) and binary32 (32 bits with 24 bit fraction and 8 bit exponent). Note that the number of fractional bits includes the implied leading "1" bit. The number of physical mantissa bits is one less than the number of fractional bits, and there is also sign bit. Physically, the binary32 format would have 1 sign bit, 8 exponent bits, and 23 mantissa bits.

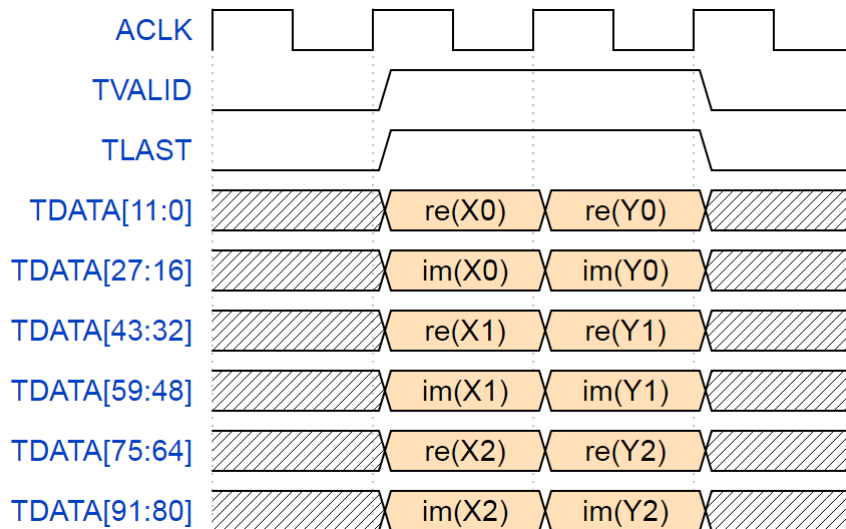
It is not uncommon to process complex data (that is, data consisting of a real and an imaginary component). If complex data is being sent over a single stream, the real and imaginary parts will be sent in parallel over a wider stream with the real part will go in the least significant word. For Serial data, the real part will come first (earlier in time).



Above are examples of parallel complex data (one sample per clock and two samples per clock). Below is an example of serial complex data.



For performance reasons (and the limited clock rate available in FPGAs), it is sometimes desired to transfer more than one sample per clock. This is called *supersampled* data. In this case, each sample (or component of the sample for complex data) is first extended to an integral number of bytes, and then these are packed together with the earlier in time samples occupying the lesser significant position:



Data Packing/Extending

When connecting two blocks with different data widths, there are two different ways of converting the signals. The AXI standard views data as a stream of bytes without explicit meaning. Going from a narrow to a wider interface will cause the bytes to be packed. For example, going from a 16 bit interface to a 32 bit interface will pack two 16 bit words into each 32 bit word. Likewise going from a wide to a narrow interface will retain all the data bytes with the output running at a higher rate than the input. This is desired behavior when interfacing to a memory, for example.

The other situation is when the underlying bit widths of the data changes, for example when interfacing a filter that uses 16 bit data to a filter using 24 bit data. When increasing the width (e.g. 16 bit source feeding a 24 bit sink) the data should be sign extended per PathWave FPGA's policy of right justifying fixed point data.

Polarity

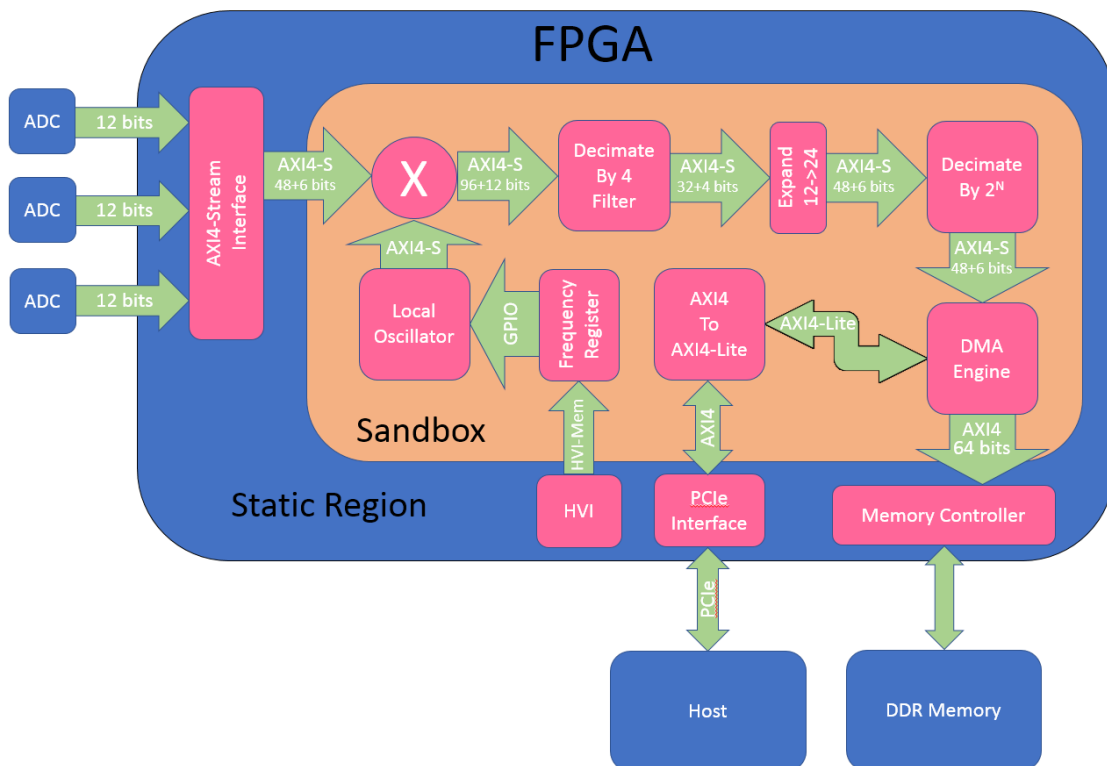
The control signals for the AXI buses are generally active high. The exception is the nRST signal which is active low. PathWave FPGA uses an active low nRST signal. The remaining control signals should be active high. Further, PathWave FPGA should sample signals and change signals on the rising edge of CLK.

Signal Interfaces

Signal Type	Interface	Discussion
Clock	clock	One or more free running clocks. Signals change on and are sampled on the rising edge of clock.
Reset	nRst	One or more active low reset signals.
Control Bus Slaves	AXIMM AXILite	Most Control Bus Slaves can probably use the simpler AXILite interface. A simple block of registers can easily decode an AXILite interface with minimal logic. If higher performance of burst access is desired, then the higher capabilities of the full AXIMM bus could be used.
Control Bus Masters	AXIMM AXILite	These interfaces are full featured enough to meet the needs of IP that needs to instigate access to addressable memory/devices.
Continuous Streaming Data	AXIS	This interface supports the flow control and auxiliary data needs of continuous data transfers.

Signal Type	Interface	Discussion
Block Mode Streaming Data	AXIS	This interface includes the TLAST signal that can be used to break the stream into arbitrary sized packets.
Memory Read/Write Data	AXIMM,AXILite, AXIS	Memory, particularly off-chip memory, is generally used for storing larger amounts of data which often require high throughput accesses. If the user IP needs random access to the memory, then AXI4 is probably the better fit. If the memory is going to be used as a source or sink of streaming data, using a DMA engine in the static region, then an AXI4-Streaming interface would be a better fit.
Supersampled Streaming Data	AXIS	As discussed above, if supersampled or complex data needs to be used, it will first be extended to an integral number of bytes and then packed into a wider AXIS interface.
Mem	mem	Some addressable interfaces, such as HVI, have distinct, deterministic timing performance requirements. For very simple designs, this provides an ultra-lightweight, addressable interface.

Example Usage



Discussion of Example

This simplified example shows how these interfaces might be utilized.

In the above example, ADCs generate three parallel 12 bit samples per clock. In the static region these samples are converted to an AXIS bus as follows. Each sample is converted from 12 to 16 bits by sign extension. The resulting six bytes are concatenated together to form a 48 bit wide streaming data bus. One bit per byte of User data is added (six bits total) to contain trigger information. Note that these are more bits than necessary, but for compliance with the specification recommendations the extra (unneeded) bits are included.

The three real samples per clock are mixed with the output of a local oscillator to form three complex samples (96 bits total). The user data (still one bit per byte) is now 12 bits wide. Note that even though the interface into and out of the mixer is 16 bit data, since the user knows the data is only 12 bits wide, the internal logic of the multipliers in the mixer need only operate on 12 bits of data (ignoring the 4 extension bits).

After decimating by four, the data rate has been reduced to one complex sample per clock (actually 3/4 sample per clock - thus handshaking is needed) with the real and imaginary halves each using 16 bits. For increased dynamic range, the Decimate by 2^N block operates on 24 bit data rather than 12 bit. An expander widens the bus to 24 bit data (time two because it is complex). Note that the AXIS bus need not be a power of 2. It only has to be an integer number of bytes.

The output of the Decimate by 2^N block flows into a DMA Engine. This is designed to FIFO up the data and burst data via an AXIMM bus to the memory controller in the static region that will interface to the external DDR memory.

The Host controls the DMA Engine via the PCIe interface. The static region contains the PCIe interface and passes an AXIMM bus into the Sandbox. Since the registers controlling the DMA Engine are simple, there is no need for the DMA Engine to implement a full blown AXIMM interface. Instead, the AXIMM bus from the PCIe interface is converted to the simpler AXILite bus which feeds the registers in the DMA Engine.

For allowing synchronous measurements with other modules, the Frequency Register is controlled via time deterministic PC-Mem bus. The output of the Frequency Register is a plain Vector without control signals or handshake. This output controls the frequency of the Local Oscillator the output of which feeds the mixer.

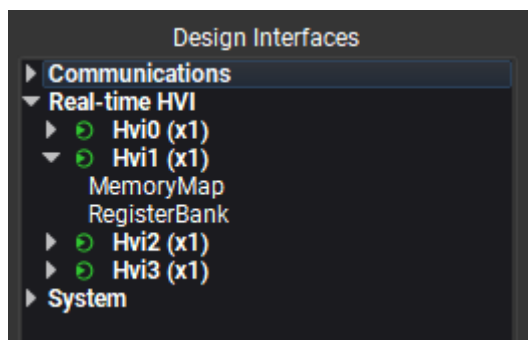
Associated Files

[AXI Reference Guide](#)

Adding a Memory Map

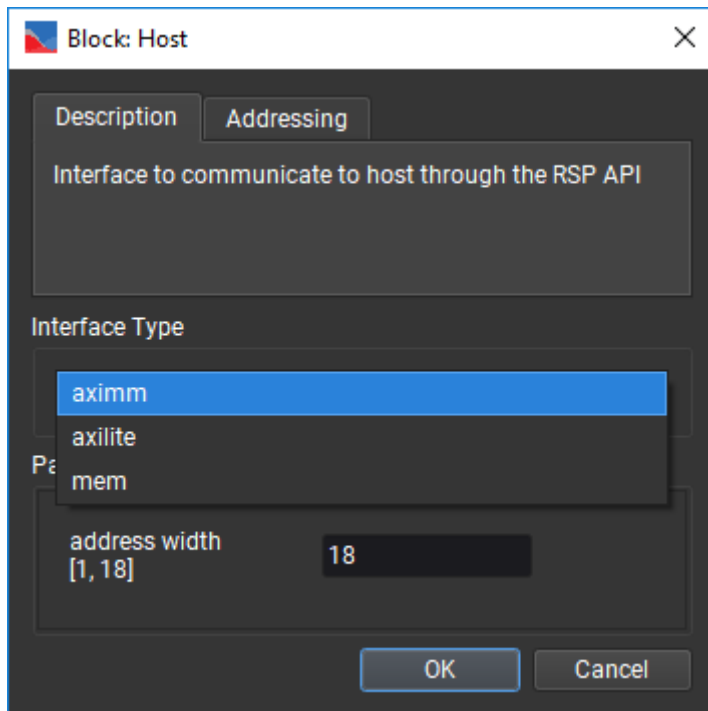
Some addressable design interfaces can be instantiated into the design using a different interface type from the one of the interface of origin. This is to simplify user's design by eliminating the use of an explicit converter, when such conversion is required. At build time, PathWave FPGA recognizes this type of interface instances and automatically generates the conversion logic between the design interface and the interface instance.

The design interfaces that support this function can be identified by the existence of the option *MemoryMap* underneath the interface name, as can be seen in the following image for interface *Hvi1*:



When double-clicking on the *MemoryMap* option, the new interface instance block dialog will appear. In the dialog, the **Interface Type** section provides a list of available conversions for this interface. The first item in the list is always the type of the design interface. If there is only one available option (i.e. only the design's interface type), then this one is automatically picked and the Interface Type section is not shown.

In the following image, the available options for the *Host* interface of M3XXXA modules are shown:



Adding a Register Bank

PathWave FPGA is dedicated to helping customers get their designs ready and tested fast; to facilitate this, PathWave FPGA created Register Banks.

Register Banks are a type of block that can be placed inside the PathWave FPGA schematic. When a register bank is placed in the schematic, PathWave FPGA will generate behind-the-scenes logic to connect the signals that are displayed on the schematic to a memory mapped bus that the customer can access from the Host. By moving this address logic creation inside PathWave FPGA, the user does not have to worry about address overlaps, or decoding blocks. This allows customers to focus their attention on the important parts of their design, and not have to worry about boilerplate components.

How to Create and Update a Register Bank

Below are the steps for creating a Register Bank, and then updating a register bank.

Launching the Register Bank Dialog

1. With a project open, in the [Design Interfaces pane](#), expand **Communications** then expand the interface to which the Register Bank will connect. For the M3102A and M3202A, this will be called **Host**. Under this interface there will be a selection called **RegisterBank**.
2. Either double click on **RegisterBank** or drag **RegisterBank** onto the design canvas to open the Register Bank Dialog.

Creating a Register Bank Using the Register Bank Dialog

With the Register Bank Dialog open you are able to start designing a Register Bank. Register Banks consist of a configurable group of registers with a contiguous address space.

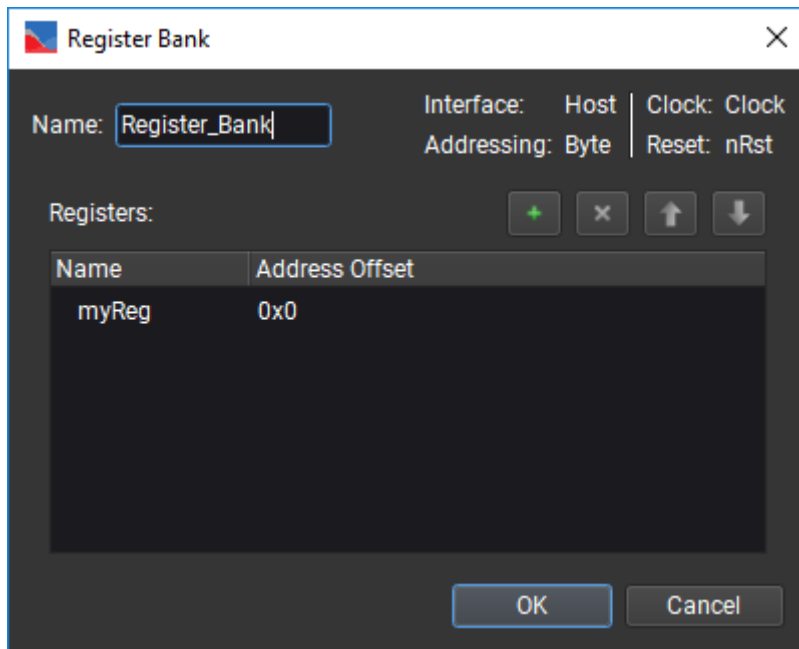


Figure 1: Register Bank Dialog when opened into a new project.

Below are main features of the Register Bank Dialog

1. Register Bank Name - This is the name that will be displayed on the block when it is placed in the schematic.
 - a. The Register Bank Name must be unique, and valid HDL syntax (see [Naming Conventions](#)).
2. Register Bank Information - The top right of the dialog displays what Clock and Reset are connected to and which Interface is being used.
 - a. Addressing - This is the unit used by the Address Offset column.
3. Registers - You can view and edit registers that are contained within the Register Bank here.
 - a. Name column - Double left click on an entry to edit a register name. A register name must be unique within the register bank, and have valid HDL syntax (see [Naming Conventions](#)).
 - i. If an issue is detected, the text will turn red and display a tool tip stating the reason for the failure.
 - b. Address Offset column - This column displays the address offset of a register. These are displayed for informational use only and cannot be directly edited.
 - c. Adding Registers - Click the "+" button on the dialog to add a register to the bottom of the list.
 - d. Removing Registers - Select a register (or multiple registers by selecting one, then holding shift and clicking another) and click the "x" button on the dialog or press the delete key.
 - e. Reordering Registers - A register or multiple registers can be moved by selecting them and either dragging them or using the up and down arrow buttons on the dialog. This changes the address offset field of the moved register and updates offsets of other registers affected by the move.
4. OK/Cancel - Click OK to create a Register Bank that can be placed on the schematic, or Cancel to close the dialog with no other actions taken.

- a. If the dialog detects any issues with the Register Bank, it will disable the "OK" button and display the text "Issue Detected". Please look for the red text to see why the Register Bank is invalid.

Placing the Register Bank in the Schematic

Now that we are done editing the Register Bank, it is time to place the block onto the schematic. To place the block onto the schematic, hit the "OK" button. The block will now be hovered below your cursor. At the location you want to place the block, left click. Below is an example block that was created with default values.

As the block instance name, the 'Name' defined in the dialog earlier is used, followed by the associated interface name inside square brackets.

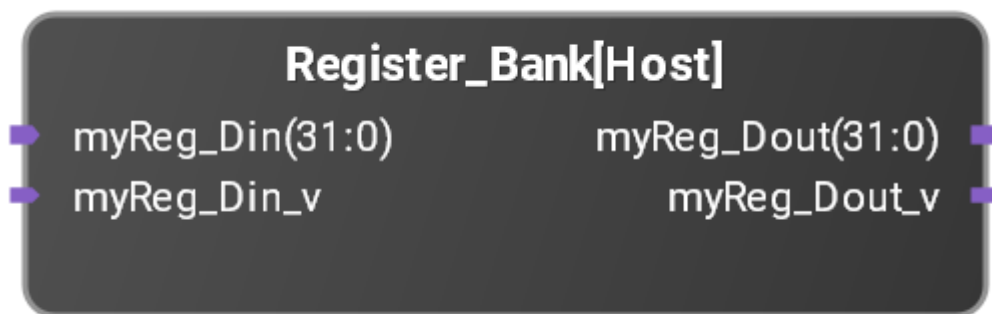


Figure 2: Register Bank block when placed onto the schematic.

Once in the schematic, Register Banks are treated the same as any other block. You are able to move, copy, flip ports, and remove. To use them in your design, just connect the signals displayed on the block to the logic you wish to interact with from the host. PathWave FPGA will handle all of the routing logic for Simulation and Building. You are able to recognize the individual registers in a Register Bank by looking at the names of the signals. The more registers you add to the Register Bank, the more signals will be available. Below is an example of a register block with two registers added to it.

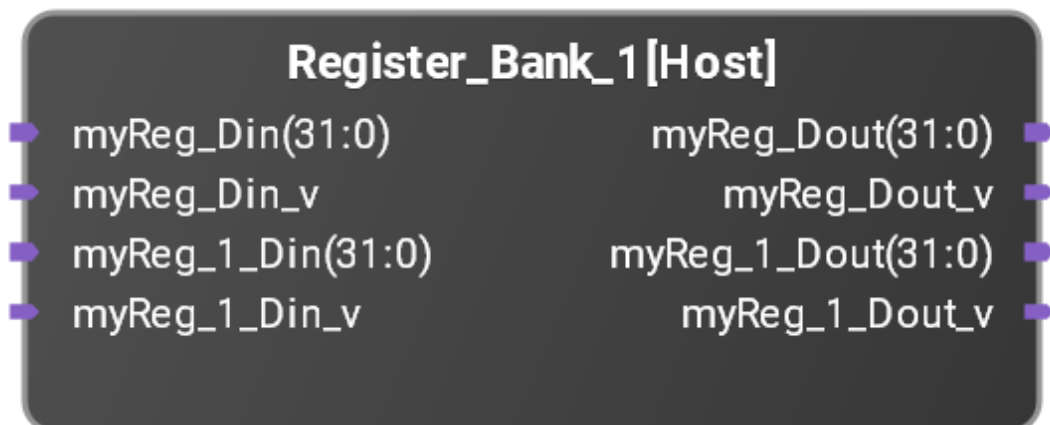


Figure 3: Register Bank block that has two RW registers in it.

Updating Register Banks

A unique feature of Register Banks, is their ability to be modified after they are placed on the schematic. To update the Register Bank we have in Figure 2 to the Register Bank we have in

figure 3 we will open the Register Bank Dialog up from the block. There are two ways of opening this dialog.

1. Double click on the Register Bank that you wish to update.
2. Right click on the Register Bank you wish to update, and select "Properties..."

The Register Bank Dialog will open up and display the information that describes the Register Bank you will update.

To add in the second register to our Register Bank, click "Add", then click "OK". Your Register Bank will now have the signals associated with the second register.

If you wish to return your register to the state it was in before the update, simply click the "Undo" Icon in the Icon bar, or use "Ctrl + z".

Register Bank Operation

A Register Bank is a collection of N 32-bit registers. For each register within the register bank, there is an internal data register. The data in this internal register may be updated either by a host write access or from the Din port if Din_v is asserted for that register. If both Din_v is asserted and the host writes to the register, then the host write will take precedence (though if Din_v remains asserted, then the data in the internal register that the host wrote will up replace by the Din value on the next clock cycle). Thus if Din_v is tied high, then the register becomes essentially a read only register.

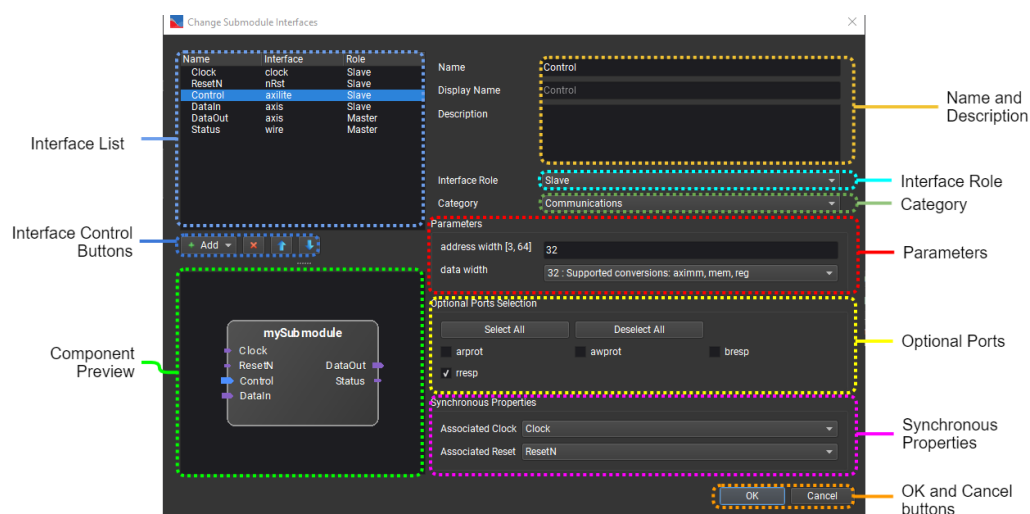
The Dout port is driven by the value of the internal register. A host read access will return the value of the internal register.

The Dout_v signal is asserted high for one clock period when new data is written. This is any time a host write occurs or when Din_v is asserted.

When Din_v is asserted, the internal register is updated from Din (unless the host is also writing to the register at the same time). This value may then be read by a host read access. This value will also be sent out the Dout port one clock later.

Configuring Submodule Interfaces

PathWave FPGA submodules contain interfaces to connect to blocks in the parent design. When a submodule project is created, the Change Submodule Interfaces dialog will open automatically. To open it again, select **Project > Change Submodule Interfaces...** or click the **Change Submodule Interfaces** button in the **Design Interfaces** section of the main window. This menu option and button will only appear when editing a submodule project.



Interface List


This table lists the interfaces in the submodule, with their name, interface type, and interface role. When you select an interface in this table, it will be the target of any changes made with


the other controls in the dialog. Interfaces can also be reordered by dragging them to their desired position within this table.



Component Preview

This shows the submodule as it will appear when added to another design. Slave interfaces are placed on the left, and Master interfaces are placed on the right. The interface that is selected in the table above is colored blue.

Interface Control Buttons

When you click the  **Add** button, you can select an interface from a list. This will add a new interface of that type.

The  **Remove** button will delete the selected interface.

The  **Up** and  **Down** buttons will move the selected interface in the table and the Component Preview.

Name and Description

The **Name** field changes the name of the interface.

The **Display Name** is what will appear in *PathWave FPGA*.

The text entered in the **Description** field is shown when adding instances of this interface to the submodule. It is also shown in the **Properties** dialog for the interface when the submodule is used in another design.

Interface Role

The **Interface Role** controls whether the interface will be a **Master**/output or **Slave**/input. **Master** and **Slave** are defined in terms of using the submodule in another design, from the outside looking in.

Category

The **Category** controls where the interface will appear in the **Design Interfaces** section of the main window.

Parameters

Some interfaces have one or more parameters, which control the width of some of the ports in the interface. In the example diagram, the AXI Lite interface has two parameters. Address Width must be between 1 and 64 bits. Data Width has two options, 32 and 64 bits. The parameter values are verified to be within the limits when you click the **OK** button. If they are not within the limits, they must be corrected. If a parameter controls the width of an optional port and that port is disabled, the parameter field will be disabled (grayed-out).

Optional Ports

Some interfaces have one or more optional ports. The check-box for each port determines whether that port will be present in the interface. The **Select All** and **Deselect All** buttons will enable or disable all optional ports.

Synchronous Properties

Some interfaces must be associated with a clock and reset. If there are any synchronous interfaces in the submodule, there must be at least one clock and one reset. If there is more than one clock or reset, then the **Associated Clock** or **Associated Reset** menu allows you to choose the associated clock or reset for each interface.

OK and Cancel Buttons

The **OK** button will apply the changes to the submodule interfaces. If there are any parameter errors or missing associated clock/resets, you will need to correct them before the changes can be applied.

The **Cancel** button will discard the changes to the submodule interfaces.

Changes to the Sandbox

After pressing Ok on the dialog, if there were no errors, the sandbox is automatically updated with the new changes.

Removing an Interface

If an interface is removed, then all Design Interfaces blocks with that interface are removed.

Changing an Interface

If any modifications are made (except changing Interface Role), then those changes are made reflected in all Design Interfaces blocks with that interface. This may result in connections being lost if they were connected to an optional port which was removed.

Changing the interface role results in the Design Interfaces blocks with the interface being removed.

Replacing an Interface

If you remove an interface and replace it with a **compatible** interface with an **identical name**, then all Design Interface blocks that had the old interface are replaced with blocks that have the new interface.

If you remove an interface and replace it with an **incompatible** interface with an **identical name**, then all Design Interface blocks that had the old interface are removed as if the interface was removed.

Currently the only interface types compatible with each other are axilite and aximm. They are also considered compatible if the original interface type is the same as the new one (e.g. axilite to axilite).

For example, you could replace an aximm named 'host' with an axilite called 'host' and it will substitute the appropriate Design Interface blocks. But you could not replace an aximm interface named 'host' with a mem interface named 'host'.

Adding an Interface

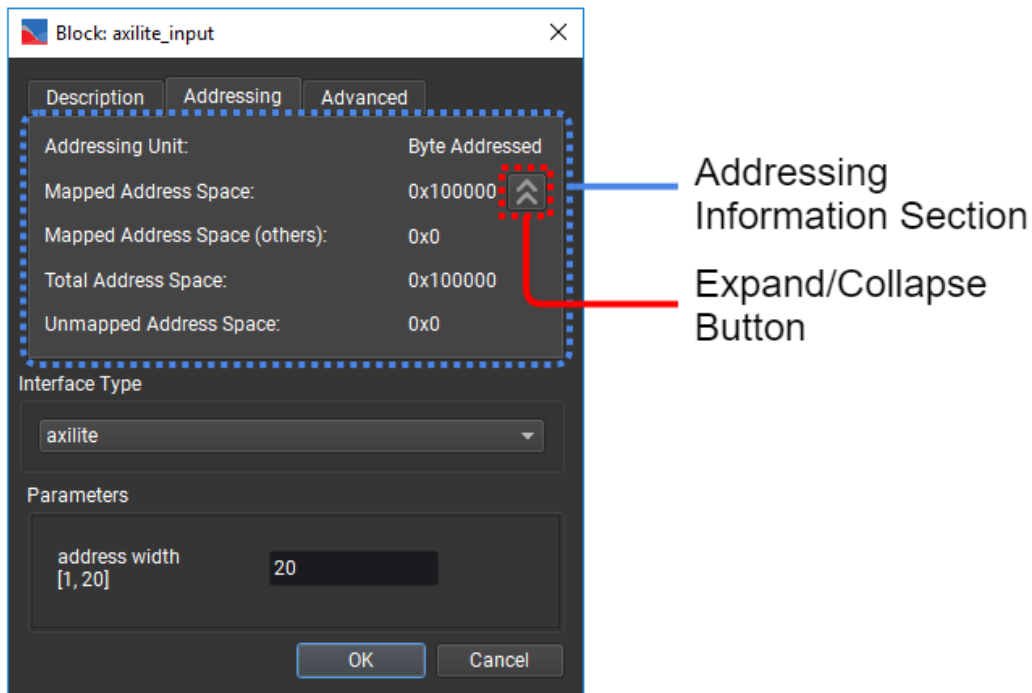
The interface was just added, so no blocks with the interface will be in the Submodule.

Deciding the Address Width of an Interface







If an addressable slave interface is available for a design, the user is allowed to configure the address width for the instances of this interface.

Selecting the address width value is straightforward when only one instance of an interface is instantiated. Deciding the correct value when there are multiple instances of an interface, and Register Banks, can become a complicated job. If we include to the above scenario the usage of interface instances with different addressing schemes, the probability of exceeding the available address space is high.

PathWave FPGA assists by calculating, on-the-fly, the address space mapping of the instances of an interface and provides this information to the user in the instance's block dialog, as shown below.



Addressing Information Section

- **Addressing Unit:** Displays the data bits addressed by each addressable unit which depends on the [addressing scheme](#) of the selected interface instance. For the case of 8-bits and 32-bits addressable data bits, *Byte Addressed* and *Word Addressed* are used respectively.
- **Mapped Address Space:** Displays the address space, in *Addressing Unit*, that is mapped by the current interface instance.
 - If there is an error in the calculation, the value turns to red and an error symbol, , is displayed at the left side of the value. Hovering over the  symbol provides more information about the error in a tool-tip.
 - For important information regarding the calculation, an information symbol, , is displayed at the left side of the value. Hovering over the  symbol provides more important information in a tool-tip.
- **Mapped Address Space (others):** Displays the address space, in *Addressing Unit*, that is mapped by other interface instances, and register banks, in the design.
 - If other instances exist, an information symbol, , is displayed at the left side of the value. Hovering over the  symbol provides a list of all the other instances and registers along with their mapped address space as a tool-tip.
- **Total Address Space:** Displays the address space, in *Addressing Unit*, that is available by the interface.
- **Unmapped Address Space:** Displays the address space, in *Addressing Unit*, that is left unmapped for the interface.

Expand/Collapse Button

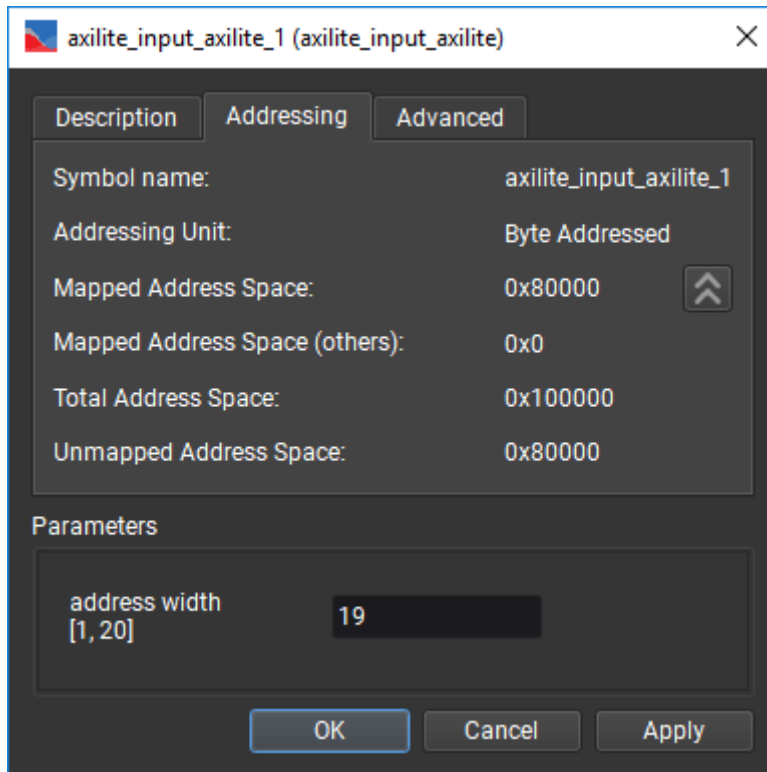
This button is used to show/hide some of the addressing information. When collapsed, the only information visible to the user is the *Addressing Unit* and the *Mapped Address Space*.

Exceeding the available address space

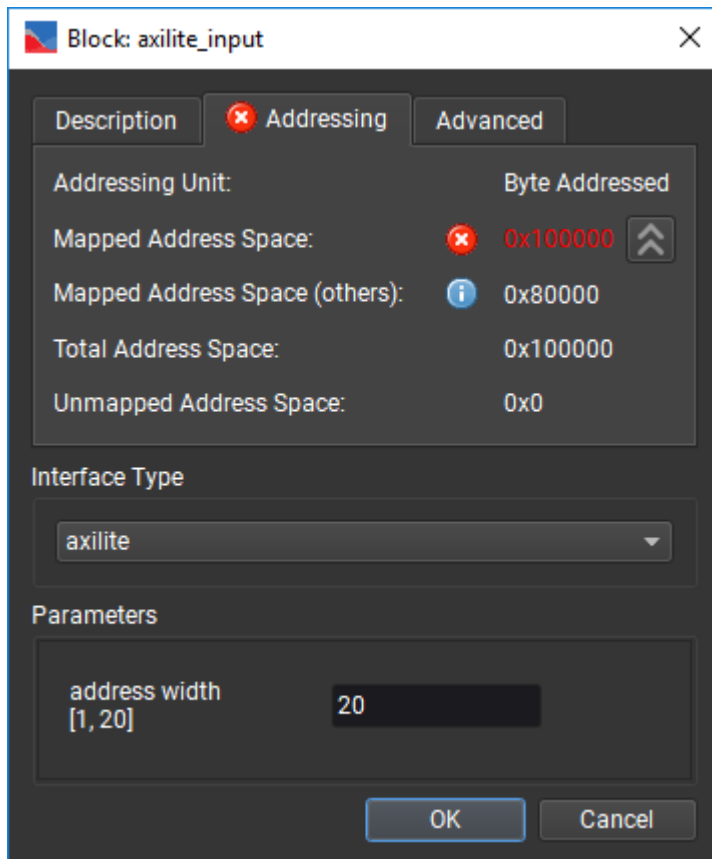
In case of multiple interface instances (and/or registers) for a design interface, it is possible that more than the available address space is required by the user's design. In that case, the calculator of the address space will identify the issue and display error to the user through the block dialogs of the interface instances or register banks.

For example, let's take the case of a byte-addressed axilite interface with 20-bits of address width. This will give a total of `0x100000` bytes available address space to be mapped.

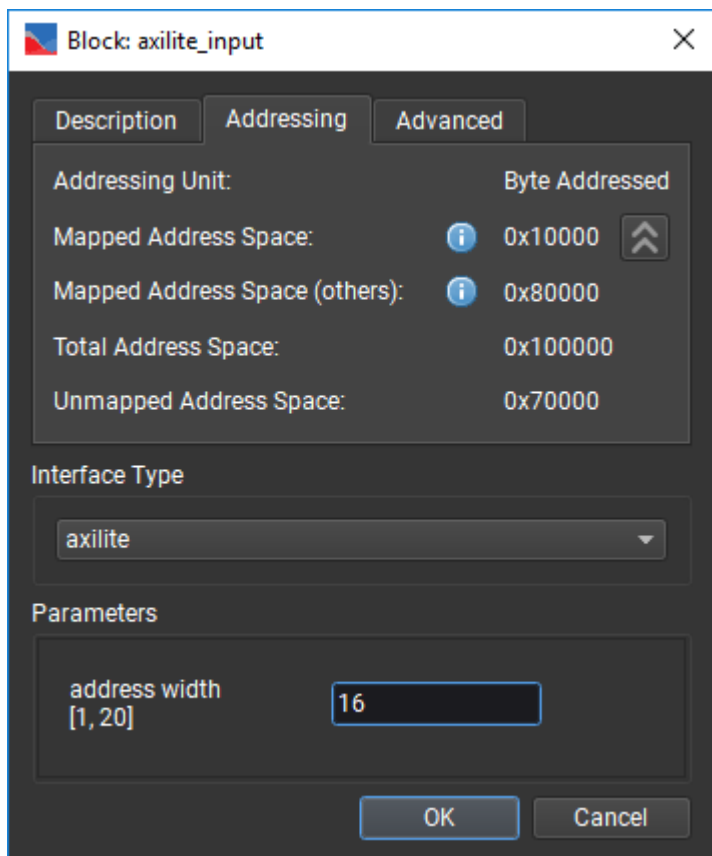
If we create one instance of this interface using an address width of 19-bits, the mapped, and unmapped, address space will become `0x80000`.




If we now create a second instance of this interface selecting to use 20-bits, the calculator will detect the overflow and report error, as shown in the following picture:

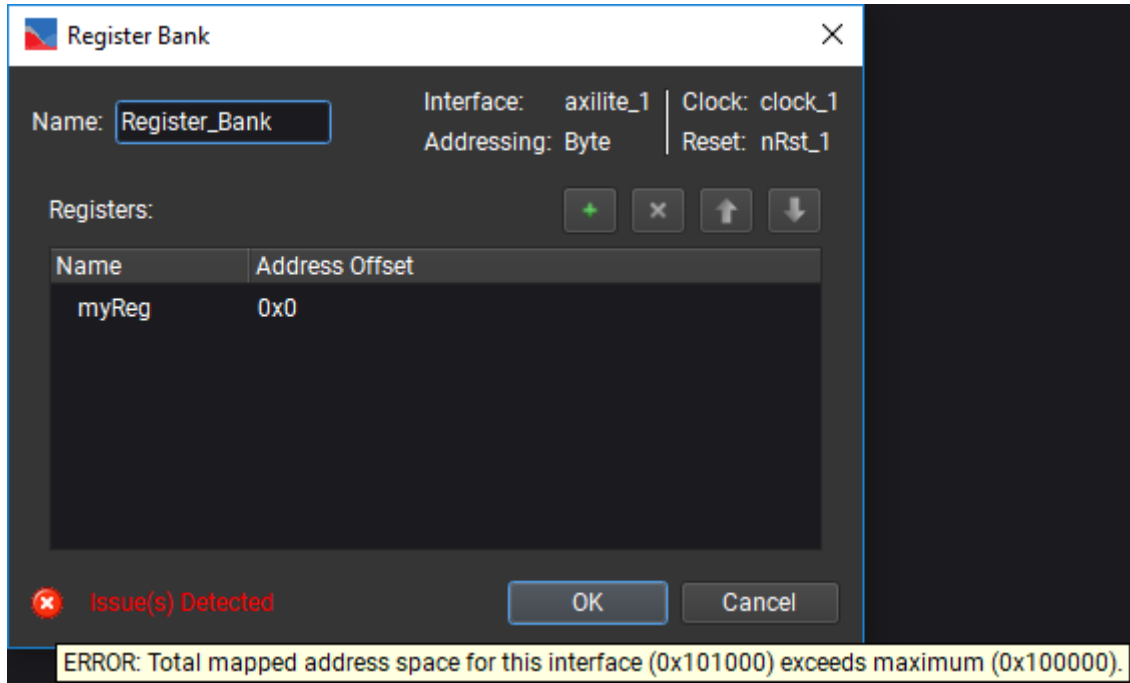


By adjusting the address width value of the new instance, the user can find the value that satisfies the space limits:



Showing address space calculation errors in Register Banks

Register Banks are related to a design's interface and are taken into account for the calculation of the address space of an interface. If creating a Register Bank, or increasing the number of registers in an existing one, leads to required space overflow, an error message is displayed at the lower left corner of the Register Bank dialog. By hovering over the  symbol, the exact issue is described.



Registering Sandbox Interfaces

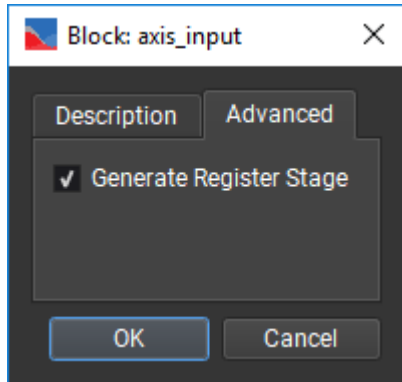
Registering a sandbox interface is the process of adding a register stage for the interface, just after it has crossed the sandbox boundary (and therefore, has entered the sandbox design).

The purpose of this procedure is to control the **timing closure** of the design. By placing a register stage at the boundary crossing from the static region to the sandbox, the path from the origin of the interface signals to their destination is made shorter. This makes meeting the timing requirements of the design easier. On the downside, this extra registration stage **increases the latency** in the path.

PathWave FPGA allows the user to control the registration of the sandbox interfaces when register stages are supported by the BSP. When register stages are supported by the BSP, a check box will appear in the properties dialog of a Design Interface block. This checkbox allows the user to choose whether or not to place the register stage in the design. When register stages are *not* supported by the BSP, this checkbox will not be shown.

Modifying the default value of the 'Generate Register Stage' checkbox is an advanced feature. If not done properly, it can lead to timing violations or invalid operation of the design. Always read the BSP documentation before applying any modifications to the default values.

An example of the properties dialog of a Design Interface block that supports register stages is shown below:



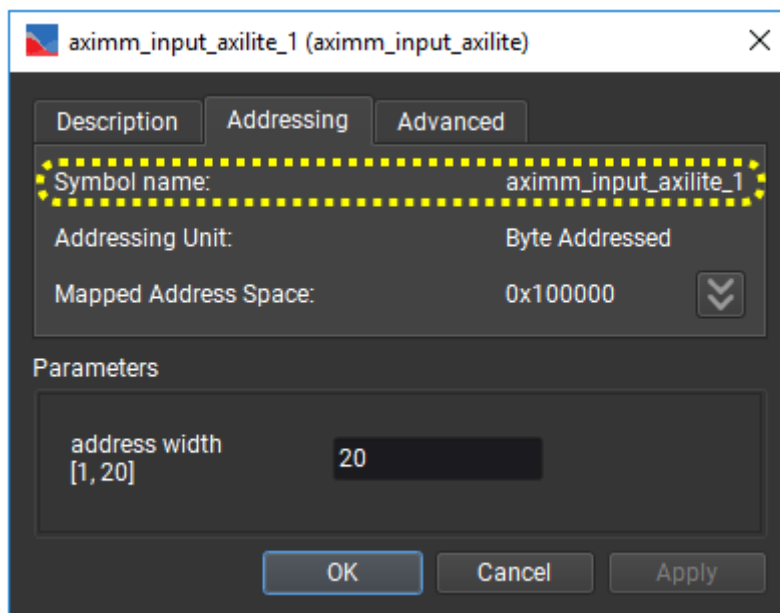
For more information about closing timing for sandbox designs, consult the Xilinx [Vivado Design Suite User Guide - Partial Reconfiguration](#) document, particularly the "Reconfigurable Partition Interfaces" section.

Symbol Names

PathWave FPGA generates symbol names for the interfaces of a design (sandbox or submodule). The purpose of these names is to be used at run-time for easier access to properties of the respective interface. Currently, they are used to get the memory address of addressable interfaces in an instrument driver application.

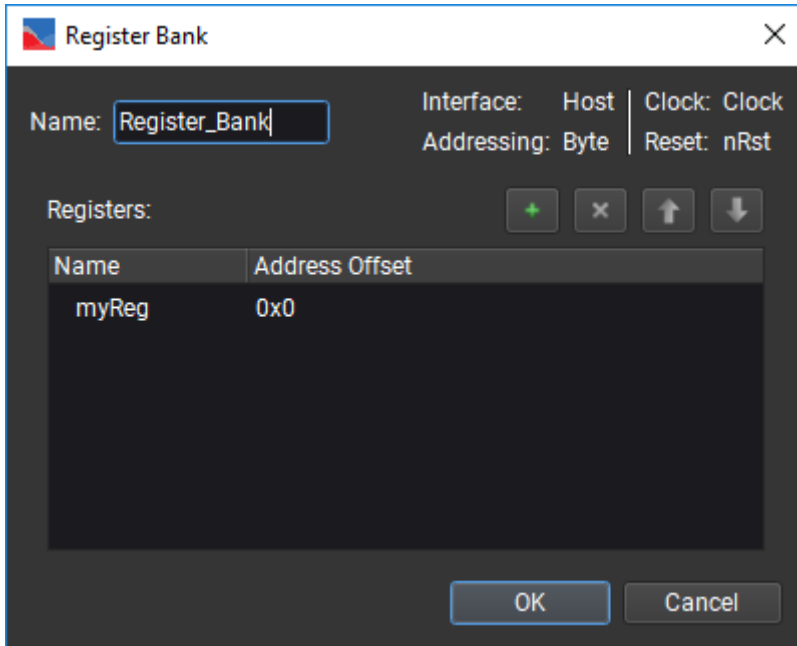
Symbol Name of a Memory Mapped Interface

The symbol name of a memory mapped interface instance is the same as the display name of that instance, so long as the display name follows the [Naming Conventions](#) rules (This is always true at the creation of a new instance). In case the display name of an instance is updated to a non-compliant one, the last valid symbol name is preserved.



Symbol name of a Register

The symbol name of a register is constructed as the concatenation with an underscore (_) of the *Register Bank* name and the *Register* name. In the *Register Bank* shown in the picture below, the symbol name for *register myReg* will be **Register_Bank_myReg**.



Symbol names for submodule interfaces

The symbol name for an interface inside a submodule design is constructed as the concatenation with a period (.) of the name(s) of the parent submodule(s) and the symbol name of the interface.

For example, imagine a sandbox design has an instance of the submodule *topSubmodule* named *topSubmoduleInstance*. Inside this submodule is an instance of *bottomSubmodule* named *bottomSubmoduleInstance*. Inside *bottomSubmodule* is an interface instance named *submoduleInterface*, creating a hierarchy that looks like this:

- Sandbox design
 - *topSubmoduleInstance*
 - *bottomSubmoduleInstance*
 - *submoduleInterface*

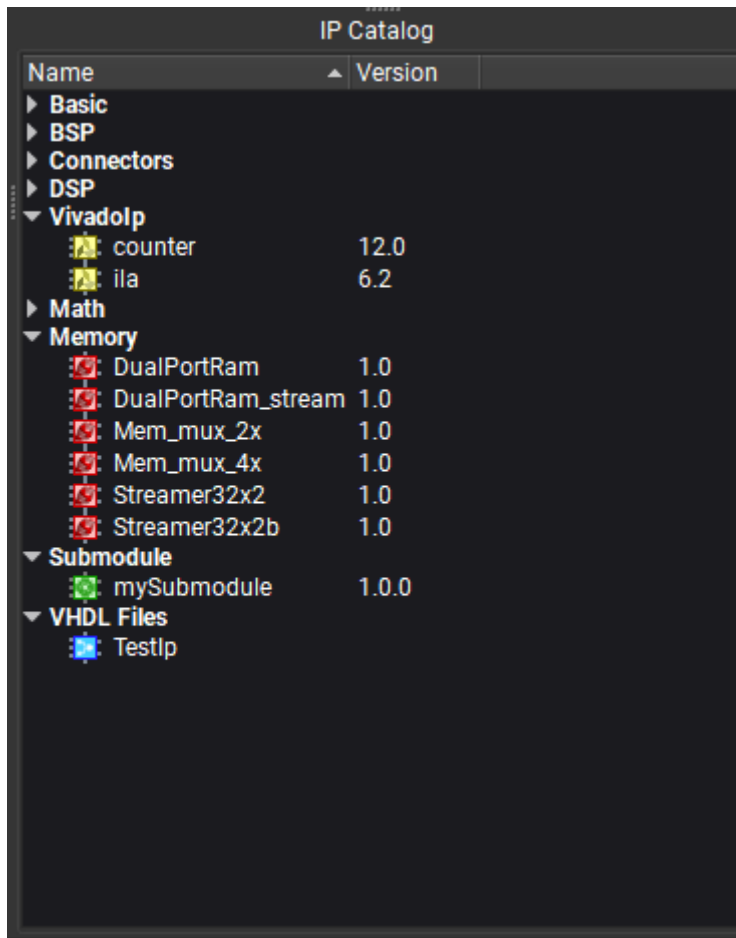
The symbol name for the interface instance *submoduleInterface* would be ***topSubmoduleInstance.bottomSubmoduleInstance.submoduleInterface***.

Similar to memory mapped interfaces, the symbol name will be the display name as long as it follows the [Naming Conventions](#) rules. Otherwise, it will retain its last valid name.

IP Catalog

PathWave FPGA comes with a catalog of IP components for you to use inside your design. PathWave FPGA also provides several methods to add your own IP or IP repositories.

The *IP Catalog* pane shows the the IP available for use in your project. You can customize how the IP is grouped into a hierarchy and which columns of information are visible. The color of the icon depends on the import method (as a standalone IP or as part of a repository) or the IP type (e.g. Vivado IP, Submodule IP).

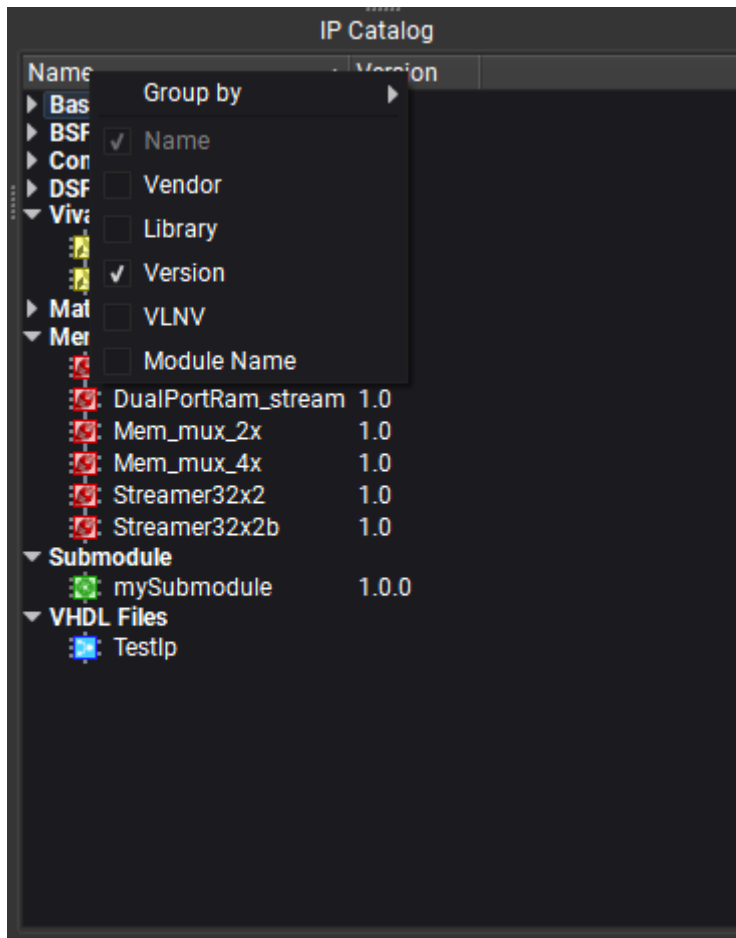


- [PathWave FPGA IP Repository](#)
- [IP Repositories](#)
- [Imported User IP](#)
- [Vivado XCI \(Xilinx Core Instance\)](#)
- [PathWave FPGA Submodule](#)

Customizing the IP Catalog window

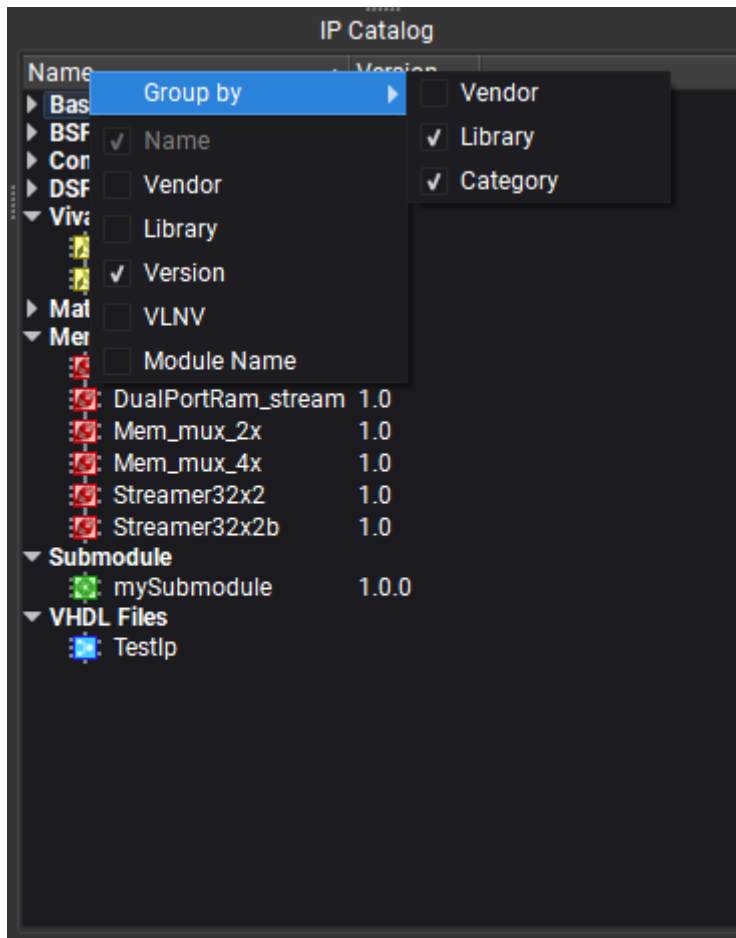
Customizing which columns are displayed

To customize the visible columns in the *IP Catalog* window, right-click on the header of the table. Click on an option to show or hide that column. By default, the *Name* and *Version* are displayed. The additional columns are *Vendor*, *Library*, *VLNV*, and *Module Name*. the *Name* header is always visible.



Customizing how the IP is grouped

To customize the way the IP are grouped in the *IP Catalog*, right-click on the header of the *IP Catalog* table, hover over the *Group by* option, and click an option to enable or disable it.




By default, the IP are grouped by *Library* and *Category*. Additionally, you can group the IP by *Vendor*.

If all options are deselected, a list of all available IP will be displayed without any kind of grouping:


Name	Version
Adder	1.0
Adder_stream	1.0
Adder_streamFC	1.0
AnalogTrigger	1.0
AnalogTrigger_x1	1.0
Axi4liteToMem	1.0
Axi4ToMem	1.0
axis_broadcaster_2x	1.0
axis_broadcaster_4x	1.0
Combine1toN	1.0
Combiner	1.0
Comparison	1.0
Complex2Real	1.0
Concat	1.0
Concat_stream	1.0
Concat_streamFC	1.0
counter	12.0
Cross_clk_domains	1.0
DecimateBy5	1.0
DecimateBy5Complex	1.0
Decombiner	1.0
Delay	1.0
Delay_stream	1.0
DualPortRam	1.0
DualPortRam_stream	1.0
ila	6.2

IP icon color coding

All IP are displayed with a "chip" icon at the left of their name. This icon can take one of the following colors based on the import method or type of the IP:

Red : This is used for IP that are coming from an IP repository, either the [PathWave FPGA default one](#), the [BSP specific one](#) or a [user imported one](#).

Green : This is used for [PathWave FPGA submodule IP](#) that are loaded into the project.

Yellow : This is used for [Vivado IP](#).

Blue : This is used for user IP imported as an [external block](#).

PathWave FPGA IP Repository

PathWave FPGA includes some IP blocks that a user can incorporate into their FPGA design. The IP blocks are categorized into different libraries so that similar blocks are grouped together. Below is a description of the IP blocks included in PathWave FPGA.

Some of the IP blocks are designed so that they can optionally process multiple samples in the same clock. This is called *supersampling*. For blocks that support this, there is a parameter called *supersample* that denotes the number of parallel samples. For example, a 32 bit adder with supersample=1 would add two 32 bit numbers. A 32 bit adder with supersample=2 would add two pairs of 16 bit numbers. This can be useful when processing data at a higher sample rate than the clock rate of the FPGA.

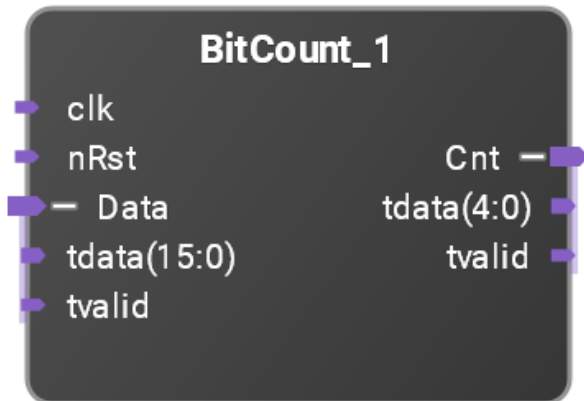
- [Basic IP blocks](#)
 - [BitCount](#)

- [Combiner](#)
- [Concat](#)
- [Concat_stream](#)
- [Concat_streamFC](#)
- [Counter](#)
- [Cross_clk_domains](#)
- [Decombiner](#)
- [Delay](#)
- [Delay_stream](#)
- [Latch](#)
- [LatchClr](#)
- [Mux2, Mux4, Mux8](#)
- [Read_mux](#)
- [Reg_xN](#)
- [Sample_delay](#)
- [Sample_delayFC](#)
- [sign_extension](#)
- [sign_extension_stream](#)
- [sign_extension_streamFC](#)
- [slice](#)
- [slice_stream](#)
- [slice_streamFC](#)
- [Connectors](#)
 - [Axi4liteToMem](#)
 - [Axi4Tomem](#)
 - [AXIStream_Broadcaster](#)
- [Math](#)
 - [Adder](#)
 - [Adder_stream](#)
 - [Adder_streamFC](#)
 - [Comparison](#)
 - [Integrator](#)
 - [Integrator_stream](#)
 - [Integrator_streamFC](#)
 - [Logic_NOT](#)
 - [Logicgate](#)
 - [Multiplier](#)
 - [Multiplier_stream](#)
 - [Multiplier_streamFC](#)
 - [Saturator](#)

- [Saturator_stream](#)
- [Saturator_streamFC](#)
- [Shift](#)
- [Shift_stream](#)
- [Shift_streamFC](#)
- [DSP](#)
 - [Combine1toN](#)
 - [Complex2Real / Real2Complex](#)
 - [ConvertBitWidth / ConvertBitWidth_stream](#)
 - [DecimateBy5](#)
 - [DecimateBy5 Complex](#)
 - [FreqCnt](#)
 - [InterpolateBy5](#)
 - [InterpolateBy5 Complex](#)
 - [Lo](#)
 - [Lo5_dc](#)
 - [Lo5_uc](#)
 - [Power2Decimator](#)
 - [Power2Interpolator](#)
 - [PRBS - Pseudo Random Bit Sequence](#)
 - [Reorder / Reorder_stream](#)
 - [ReshapeM1 / ReshapeP1](#)
 - [ReshapeDown / ReshapeUp](#)
 - [ssDecim2](#)
 - [ssInterp2 / ssInterp2fc](#)
 - [Trigger / TriggerM3x](#)
- [Memory](#)
 - [DualPortRam](#)
 - [DualPortRam_stream](#)
 - [Mem_mux_2x](#)
 - [Mem_mux_4x](#)
 - [Streamer32x2 and Streamer32x2b](#)

Basic IP blocks

BitCount



Count the number of "1" bits in the input data word. The input and output interfaces are AXI-streaming.

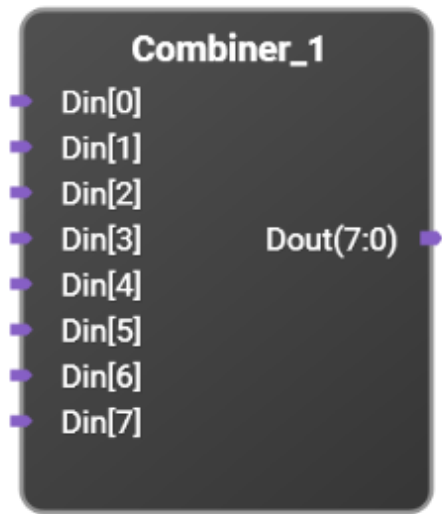
This block will sum the number of "1" bits in groups of either 4 (Pipeline=1) or 8 (Pipeline=0) input bits. Then either 4 or 8 (depending on Pipeline) of these sums are then summed. This continues until all the bits are summed. If the number of input bits is N, then the latency of this block will be $\text{ceil}(\log_4(N))$ if Pipeline=1, or $\text{ceil}(\log_8(N))$ if Pipeline=0. Setting Pipeline=0 can potentially reduce the latency through the block while setting Pipeline=1 can potentially allow the block to run at a higher clock speed. For example, if the input is N=40 bits wide, then the latency if Pipeline=1 is $\text{ceil}(\log_4(40)) = \text{ceil}(2.66) = 3$ clocks. If Pipeline=0, then the latency is $\text{ceil}(\log_8(40)) = \text{ceil}(1.77) = 2$ clocks.

Parameters

Data size: Sets the number of bits in the input data word. Variable from 1 to 1024. Default is 16.

Pipeline: When set, this adds extra pipelining as described above.

Combiner

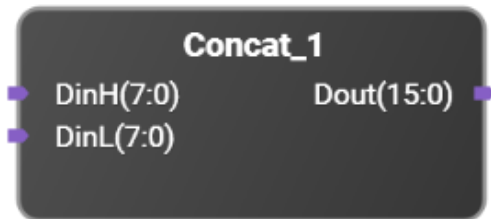


Combines N single-bit inputs into a single N-bit output vector.

Parameters

Din width: Sets the number of single bit inputs. Variable from 1 to 1024. Default is 8.

Concat



Concatenates two input signals into one single signal. DinH is the most significant half of Dout, and DinL is the least significant half of Dout.

This module does not introduce extra delay.

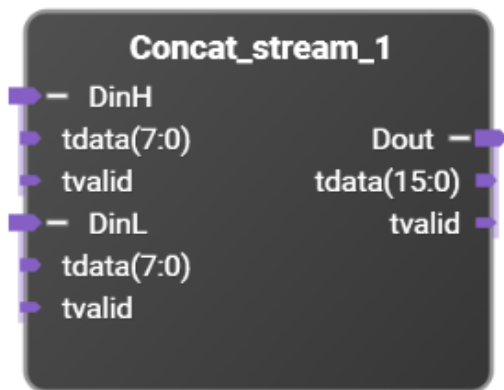
Parameters

DinH width: Sets the data width of DinH. Variable from 1 to 1024. Default is 8.

DinL width: Sets the data width of DinL. Variable from 1 to 1024. Default is 8.

supersample: Sets the supersample amount. Variable from 1 to 64. Default is 1.

Concat_stream



Streaming version of the concat block.

Concatenates two input signals into one single signal. DinH is the most significant half of Dout, and DinL is the least significant half of Dout

This module does not introduce extra delay.

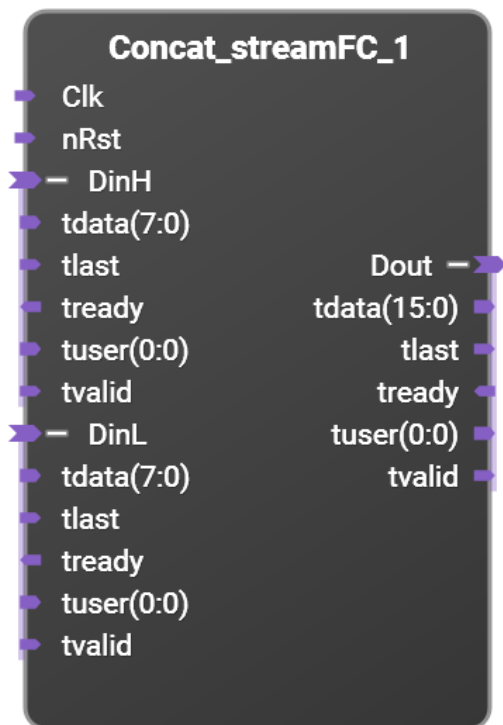
Note that both streaming inputs must assert and deassert tvalid at the same time.

Parameters

DinH width: Sets the data width of DinH. Variable from 1 to 1024. Default is 8.

DinL width: Sets the data width of DinL. Variable from 1 to 1024. Default is 8.

supersample: Sets the supersample amount. Variable from 1 to 64. Default is 1.

Concat_streamFC

Streaming flow controlled version of the concat block.

Concatenates two input signals into one single signal. DinH is the most significant half of Dout, and DinL is the least significant half of Dout

This module introduces minimum 2 clock delay.

Parameters

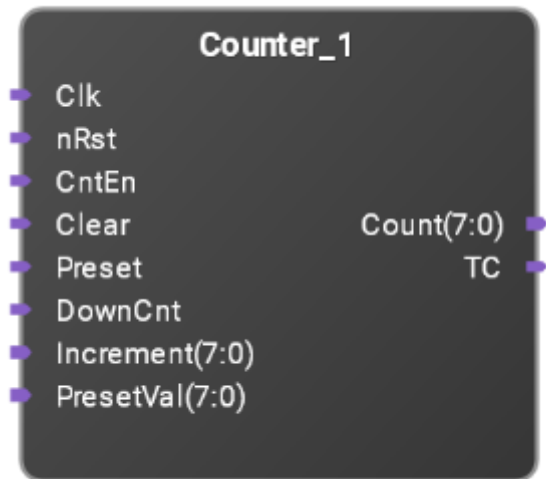
DinH width: Sets the data width of DinH. Variable from 1 to 1024. Default is 8.

DinL width: Sets the data width of DinL. Variable from 1 to 1024. Default is 8.

tuser width: Sets the data width of tuser. Variable from 1 to 8. Default is 1.

supersample: Sets the supersample amount. Variable from 1 to 64. Default is 1.

Counter



Up/down presetable counter. The counter supports clearing, presetting, counting up or down, and rolling over or stopping upon reaching Terminal Count (the last value before rolling over).

The Modulus parameter determines how far the counter will count. By default, it is set to 0 which is interpreted as 2^{Bitwidth} , or a normal binary counter. To make a decade counter, for example, set Modulus to 10.

The amount that the counter is changed for every count can be set. By default, this value is 1. If a different counter increment amount is desired, for example a counter that counts by 3s, then the Incr parameter can be changed. This parameter defaults to 1, and a non-zero value denotes a fixed counter increment. In this case, the Increment port is ignored. If Incr is zero, then the counter's increment value is determined by the Increment port. This allows changing the increment value in real time. This feature can be used to make a rudimentary accumulator.

DownCnt determines which direction the counter will count. If DownCnt=0, it is an up counter so $\text{Count} \leq \text{Count} + \text{Incr}$. If DownCnt=1, it is a down counter so $\text{Count} \leq \text{Count} - \text{Incr}$.

TC (Terminal Count) is asserted when Count is about to roll over. For an up counter, TC is asserted when $\text{Count} \geq \text{Modulus} - \text{Incr}$. For a down counter, TC is asserted when $\text{Count} < \text{Incr}$.

Rollover determines if the counter will roll over after TC or stop. If Rollover=1, then after the Terminal Count the counter will rollover. For example, if Modulus=10, Incr=4, DownCnt=0, and Count=8, then the new Count value would be 2 ($\text{Count} + \text{Incr} - \text{Modulus}$ or $8 + 4 - 10$). If Rollover=0, then the counter will stop counting at the Terminal Count value. So for a down counter with Incr=1, the counter will count down to zero and then stop.

Asserting Clear will set the counter to Reset Value (a parameter).

Asserting Preset will set the counter to PresetVal (a port).

CntEn enables the counter. If asserted, the counter will count up or down depending on DownCnt.

Note that neither Clear nor Preset need CntEn to be asserted in order to function.

When multiple control signals are asserted (e.g. Clear and Preset) at the same time, the order of precedence is:

1. (highest) nRst
2. Clear
3. Preset
4. (lowest) CntEn

Parameters

Bitwidth: determines number of bits in counter.

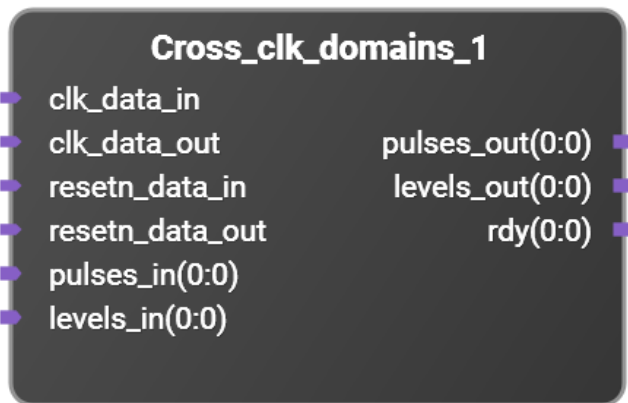
Modulus: determines how far the counter will count before rolling over or stopping with a default value of 1. Modulus=0 means 2^{Bitwidth} .

Incr: sets the fixed increment for the counter with a default of 1. If Incr=0, then the Increment port is used for a variable increment value. If Incr>0, the Increment port is ignored.

Rollover: determines if the counter rolls over (Rollover=1) or stops (Rollover=0) at the Terminal Count.

Reset Value: sets the value the counter will take when Clear is asserted.

Cross_clk_domains



Logic to handle the crossing of signal levels and pulses to and from arbitrarily related clock domains.

Logic high pulses on the input clock domain are synchronously transferred to logic high pulses on the output clock domain. An output logic pulse will always have a pulse width of one 'clk_data_out' cycle, regardless of the pulse width of the input logic pulse.

Logic levels on the input clock domain are synchronously transferred to logic levels on the output clock domain.

The transfer delay of signals from the input clock domain to the output clock domain depends upon the frequency and phase relationship between the two clock domains. Input signal levels are assumed to be relatively static compared with the clock frequencies. Input signal pulses cannot be repeated until each pulse has fully propagated through the block. The 'rdy' output signal should be used to determine when the block is ready to transfer an input pulse to the output, especially if input signal pulses may otherwise occur in rapid succession. When a bit in the 'pulses_in' input port is asserted high, the corresponding 'rdy' bit will be asserted low. When the 'rdy' bit is again asserted high, the 'pulses_in' input may again be asserted high. The 'rdy' output signal is synchronous with the 'clk_data_in' clock.

Regardless of the input and output clock frequencies, if a level input and pulse input are asserted simultaneously, the corresponding level output will be asserted either simultaneous with or before the pulse output is asserted.

Note that positive transitions are detected in the 'pulses_in' input to determine that a pulse input has occurred. Consequently, if a 'pulses_in' input is asserted high and remains high, only one pulse will be output.

'resetsn_data_in' is an active low reset signal, synchronized to the 'clk_data_in' clock.

'resetsn_data_out' is an active low reset signal, synchronized to the 'clk_data_out' clock.

Note that this block is available only for sandboxes which include more than one clock.

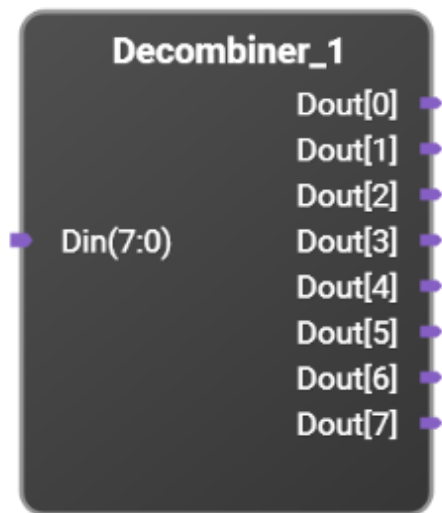
Parameters

`pulses_width`: Sets the data width of `pulses_in`, `pulses_out` and `rdy`

`levels_width`: Sets the data width of `levels_in` and `levels_out`

`levels_reset_value`: Sets the reset value of `levels_out`

Decombiner

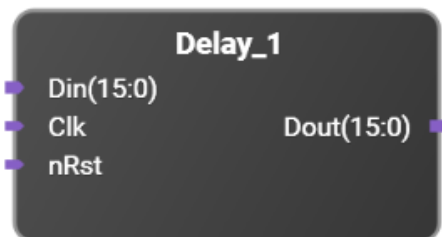


Converts a single N-bit input vector into N single-bit output signals.

Parameters

Din width: Sets the Din data width. Variable from 1 to 1024. Default is 8.

Delay



Delays input N cycles.

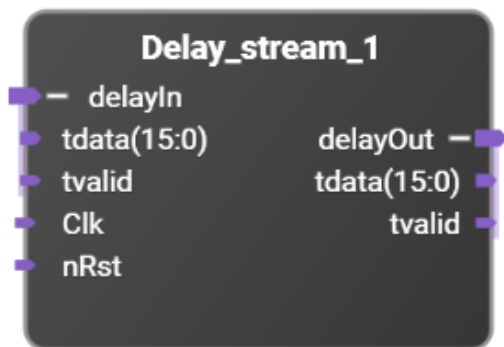
Parameters

bus width: Sets the bus width of Din and Dout. Variable from 1 to 1024. Default is 16.

latency: Sets the latency through the delay block. Variable from 1 to 1024. Default is 1.

Reset Value: Sets the value of the delay registers when nRst is asserted low. It should be the same size is Din. Default is 0.

Delay_stream



Streaming version of the delay block.

Delays input N cycles.

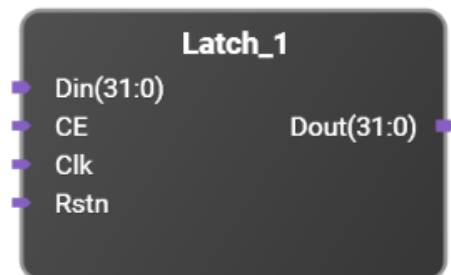
Parameters

bus width: Sets the bus width of Din and Dout. Variable from 1 to 1024. Default is 16.

latency: Sets the latency through the delay block. Variable from 1 to 1024. Default is 1.

Reset Value: Sets the value of the delay registers when nRst is asserted low. It should be the same size as tdata. Default is 0.

Latch



32 bit latch with write enable.

Parameters

Bus width: Sets the register bus width. Variable from 1 to 1024. Default is 32.

LatchClr



Latch with clock enable and synchronous clear.

If $nRst = 0$, then Dout is set to the initialization value (typically 0).

If $nRst = 1$ and $CE = 0$, Dout remains unchanged.

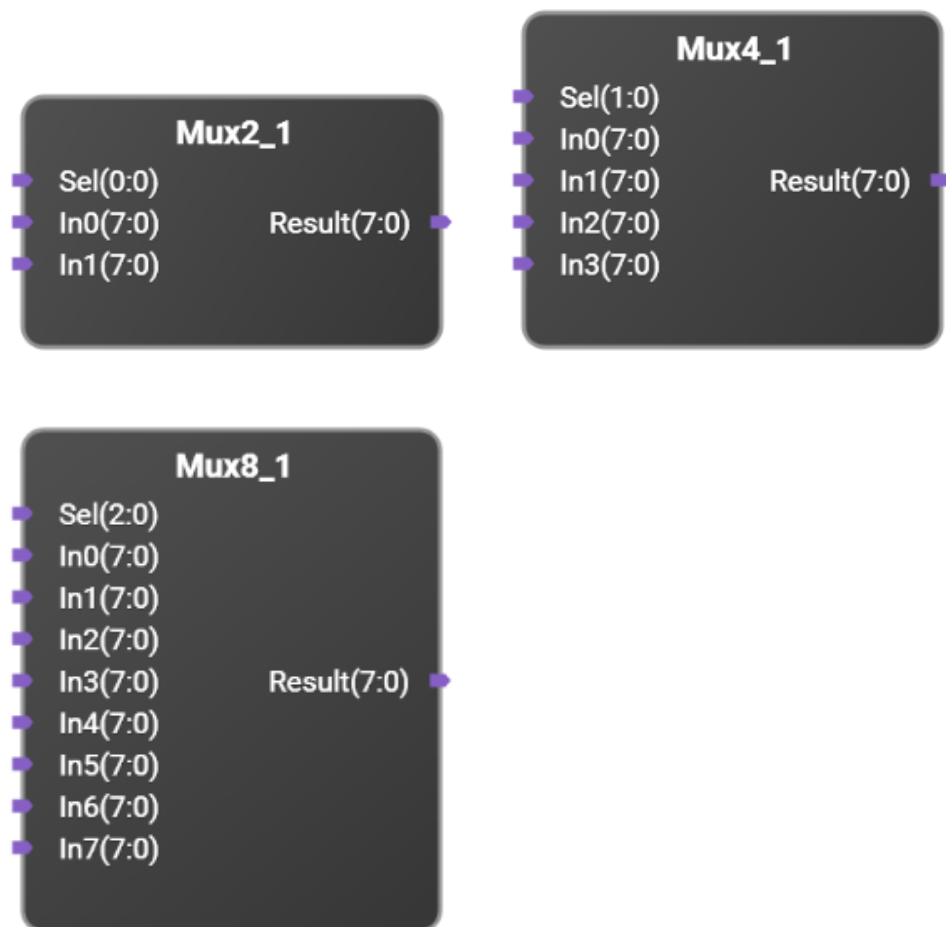
If $nRst = 1$, $CE = 1$, and $Clr = 1$, Dout is set to the initialization value on the rising edge of clk.

If $nRst = 1$, $CE = 1$, and $Clr = 0$, Dout is set to Din on the rising edge of clk.

Parameters

Bus width: Sets the register bus width. Variable from 1 to 1024. Default is 32.

Init: Sets the value that the latch resets/clears to. Default is 0.

Mux2, Mux4, Mux8

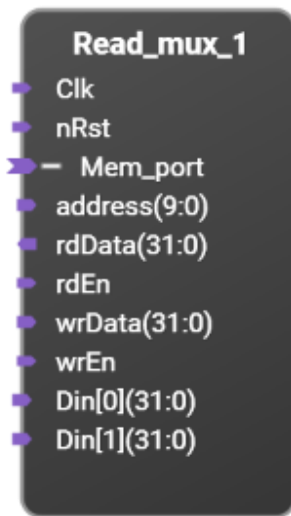
These are 2 to 1, 4 to 1, and 8 to 1 multiplexers. The value of the Sel input determines which of the various In ports connect to Result.

These are combinatorial.

Parameter

width: Sets the bus width of the In ports and Result.

Read_mux



Read data from multiple sources.

Address port is used to select one of N, 32 bit data sources. If the address index is larger than the number of input data sources, this block will return zeros.

Parameters

Number of inputs: Sets the number of 32 bit data sources. Default is 2.

Reg_xN



Captures N, 32 bit data inputs and drives to outputs. The internal data register may be updated through a write access on the 'mem' port indexed by the address value. The internal data register may also be updated to the Din value by asserting the corresponding Din_v signal[n]. When both updates are attempted at the same time, the mem write value will take precedence. The values of the internal data registers are driven out the Dout[n] ports.

Note that the Reg_xN block uses the Mem interface which uses word addressing, not byte addressing.

Mem read access will return the value of the indexed internal data register.

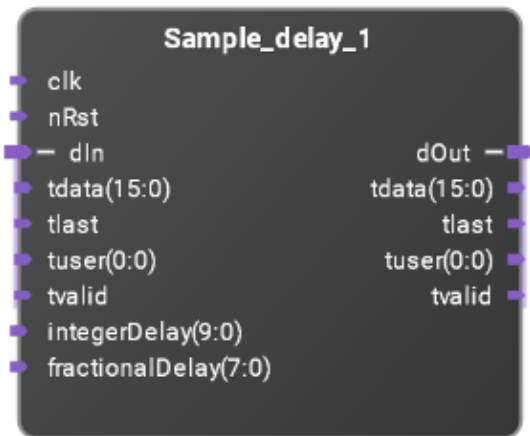
The Dout_v[n] signal is asserted high for one clock period when new data is written. This is any time a mem write occurs or when Din_v[n] is asserted.

Parameters

Number of Registers: Variable from 1 to 1024. Default is 2.

Address width: Variable from 1 to 32. Default is 32.

Sample_delay



The Sample_delay IP block delays the input data stream by a programmable number of samples. The delay does not have to be a multiple of the superample value.

The integerDelay input port sets the delay amount to a multiple of the supersample rate and the fractionalDelay input sets the delay amount to a fraction of the supersample rate. The maximum integer number of supersample delays is determined by the depth of the internal block ram and is set by the block ram address width parameter when the Sample_delay block is instantiated. The maximum number of fractional sample delays is determined by the supersample rate. The valid range for fractionalDelay is 0 to (Supersample Rate - 1). For a supersample rate of 5x, the value of fractionalDelay can be between 0 and 4. The total delay in samples is (Supersample Rate * integerDelay + fractionalDelay). For example, with a supersample rate of 5x (5 parallel samples per clock cycle) to get a delay of 7 samples set the integerDelay to 1 and the fractionalDelay to 2.

Parameters

Data width: data width (bits) of each sample. The dIn and dOut tdata ports are (Supersample Rate * Data Width) bits wide. Default is 16.

Memory Address Width: Address width (bits) of the internal block ram. Default is 10.

Supersample Rate: Number of parallel supersampled samples. Default is 1.

Tuser width: Number of Tuser bits for each sample. The Tuser ports are (Supersample Rate * Tuser size) bits wide. Default is 1.

Sample_delayFC



The Sample_delayFC IP block is the same as the Sample_delay IP block, but has reverse flow control. The Sample_delayFC IP block delays the input data stream by a programmable number of samples. The delay does not have to be a multiple of the superample value.

The integerDelay input port sets the delay amount to a multiple of the supersample rate and the fractionalDelay input sets the delay amount to a fraction of the supersample rate. The maximum integer number of supersample delays is determined by the depth of the internal block ram and is set by the block ram address width parameter when the Sample_delay block is instantiated. The maximum number of fractional sample delays is determined by the supersample rate. The valid range for fractionalDelay is 0 to (Supersample Rate - 1). For a supersample rate of 5x, the value of fractionalDelay can be between 0 and 4. The total delay in samples is (Supersample Rate * integerDelay + fractionalDelay). For example, with a supersample rate of 5x (5 parallel samples per clock cycle) to get a delay of 7 samples set the integerDelay to 1 and the fractionalDelay to 2.

Parameters

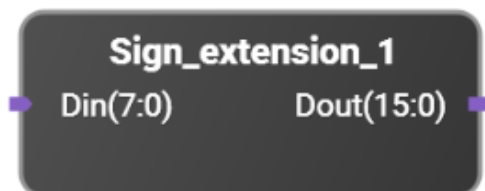
Data width: data width (bits) of each sample. The dIn and dOut tdata ports are (Supersample Rate * Data Width) bits wide. Default is 16.

Memory Address Width: Address width (bits) of the internal block ram. Default is 10.

Supersample Rate: Number of parallel supersampled samples. Default is 1.

Tuser width: Number of Tuser bits for each sample. The Tuser ports are (Supersample Rate * Tuser size) bits wide. Default is 1.

sign_extension



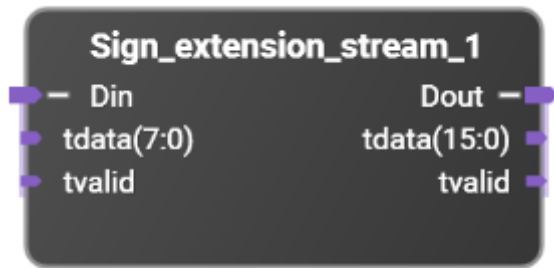
Sign extends the input vector.

Parameters

Din width: Sets the Din bus width. Variable from 1 to 1024. Default is 8.

Dout width: Sets the Dout bus width. Variable from 1 to 1024. Default is 16.

supersample: Sets the supersample amount. Variable from 1 to 64. Default is 1.

sign_extension_stream

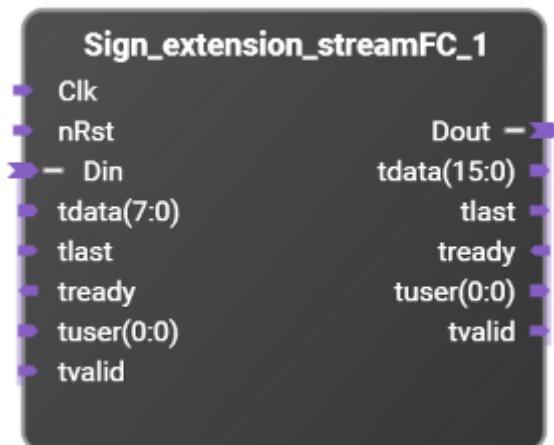
Sign extends the input data stream.

Parameters

Din width: Sets the Din bus width. Variable from 1 to 1024. Default is 8.

Dout width: Sets the Dout bus width. Variable from 1 to 1024. Default is 16.

supersample: Sets the supersample amount. Variable from 1 to 64. Default is 1.

sign_extension_streamFC

Sign extends the input data stream using full flow control.

This block adds a minimum delay of 2 cycles.

Parameters

Din width: Sets the Din bus width. Variable from 1 to 1024. Default is 8.

Dout width: Sets the Dout bus width. Variable from 1 to 1024. Default is 16.

Tuser width: Sets the tuser bus width. Variable from 1 to 8. Default is 1.

supersample: Sets the supersample amount. Variable from 1 to 64. Default is 1.

slice

Selects certain number of bits from a vector signal input.

This does not introduce extra delay.

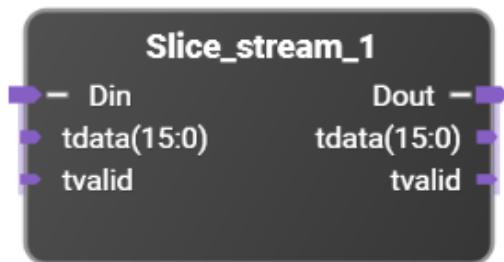
Parameters

Din width: Sets the bus width of Din. Variable from 1 to 1024. Default is 16.

offset: Sets the starting LSB position to extract bits from Din [Dout(bus_out_width:0) = Din(bus_in_width:offset_lower_bit)]. Default is 0.

Dout width: Sets the bus width of Dout. Variable from 1 to 1024. Default is 16.

supersample: Sets the supersample amount. Variable from 1 to 64. Default is 1.

slice_stream

Streaming version of the slice block.

Selects certain number of bits from a vector signal input.

This does not introduce extra delay.

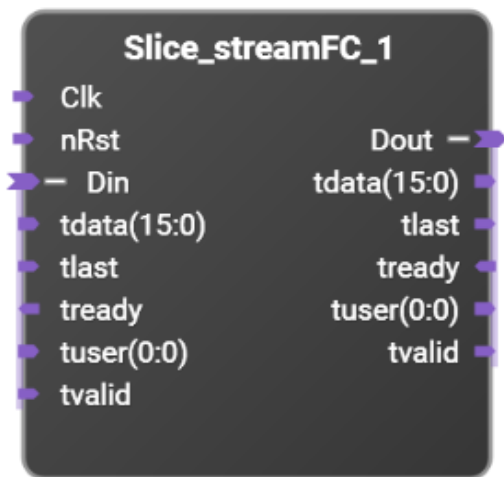
Parameters

Din width: Sets the bus width of Din. Variable from 1 to 1024. Default is 16.

offset: Sets the starting LSB position to extract bits from Din [Dout(bus_out_width:0) = Din(bus_in_width:offset_lower_bit)]. Default is 0.

Dout width: Sets the bus width of Dout. Variable from 1 to 1024. Default is 16.

supersample: Sets the supersample amount. Variable from 1 to 64. Default is 1.

slice_streamFC

Streaming version of the slice block supporting full flow control.

Selects certain number of bits from a vector signal input.

This block adds a minimum delay of 2 cycles.

Parameters

Din width: Sets the bus width of Din. Variable from 1 to 1024. Default is 16.

Offset: Sets the starting LSB position to extract bits from Din [Dout(bus_out_width:0) = Din(bus_in_width:offset_lower_bit)]. Default is 0.

Dout width: Sets the bus width of Dout. Variable from 1 to 1024. Default is 16.

Tuser width: Sets the bus width of tuser. Variable from 1 to 8. Default is 1.

Supersample: Sets the supersample amount. Variable from 1 to 64. Default is 1.

Connectors

Axi4liteToMem



Converts Axi4Lite slave interface to PC_Mem master interface.

Parameters

Axi address width: Sets the AXI interface address width. Default is 8.

Since the Mem interface uses word addressing while the Axi4Lite interface uses byte addressing, the size of the Mem interface address bus is two bits smaller.

Axi4Tomem

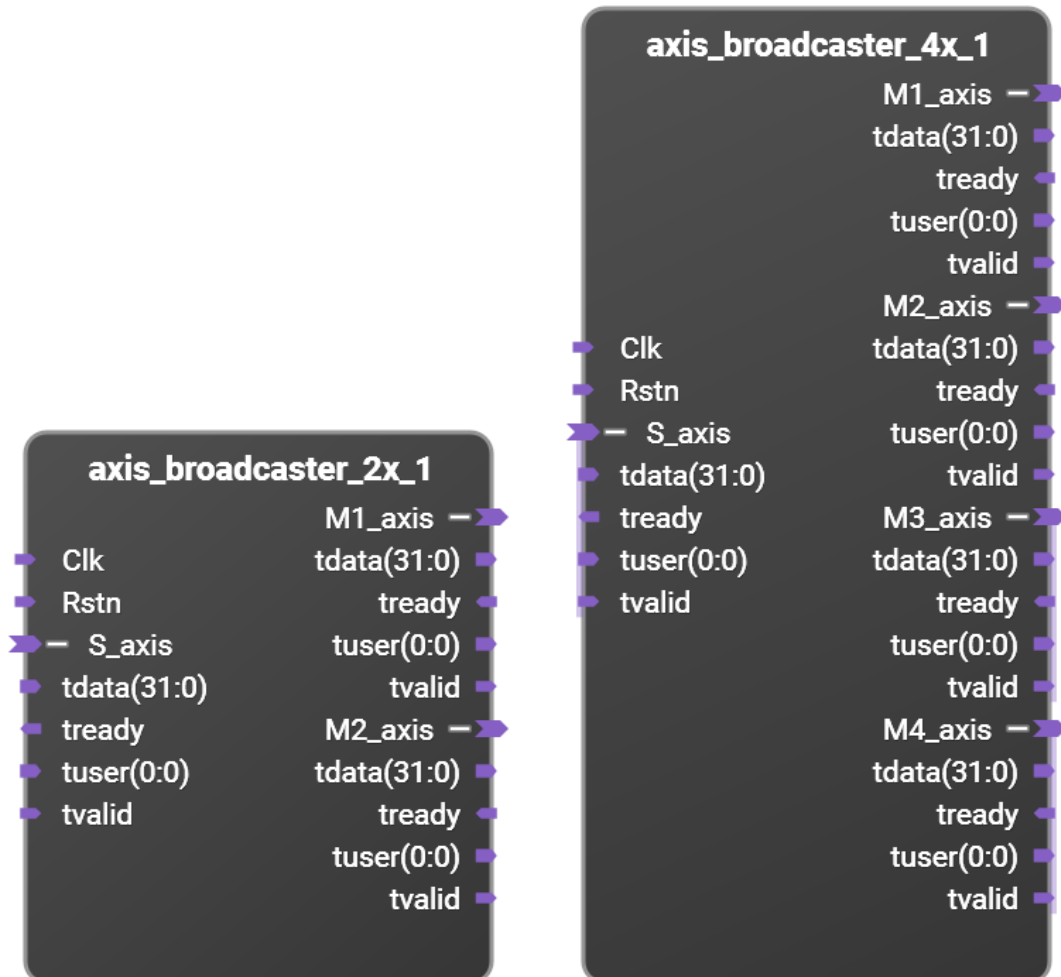
Converts Axi4MM slave interface to PC_Mem master interface.

Parameters

Axi address width: Sets the AXI interface address width. Default is 8.

Since the Mem interface uses word addressing while the Axi4 interface uses byte addressing, the size of the Mem interface address bus is two bits smaller.

AXIStream_Broadcaster



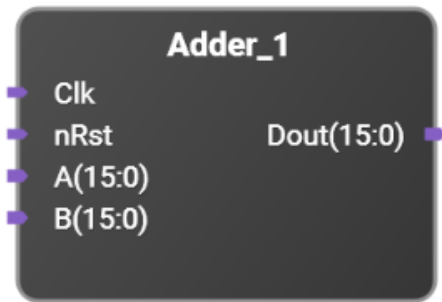
Broadcasts AXI4 streaming data from one AXI4 master to multiple AXI4slaves.

Parameters

Tdata bitwidth, default is 32. Tuser bitwidth, default is 1.

Math

Adder



Signed adder.

Inputs are expected to have the same length.

Overflow and underflow check is done when saturate is enabled.

Output width is increased by 1 when full precision is enabled.

Subtraction changes operation from A+B to A-B.

This module adds a delay of 1 cycle.

When latch input is enabled, 1 extra cycle of delay is added.

Parameters

input width: Sets the bus width of the A and B inputs. Default is 16.

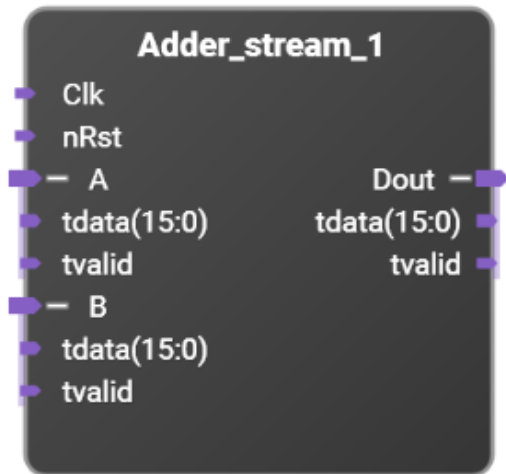
Adder implementation: Selects saturate or full precision adder modes. Default is Saturate.

latch input: When enabled the data on the A and B inputs is latched. Default is no latch.

subtract: When enabled the adder operation is changed from A+B to A-B. Default is add.

supersample: Sets the supersample amount. Variable from 1 to 64. Default is 1.

Adder_stream



Signed adder with streaming interface.

Inputs are expected to have the same length.

Overflow and underflow check is done when saturate is enabled.

Output width is increased by 1 when full precision is enabled.

Subtraction changes operation from A+B to A-B.

This module adds a delay of 1 cycle.

When latch input is enabled, 1 extra cycle of delay is added.

Parameters

input width: Sets the bus width of the A and B inputs. Default is 16.

Adder implementation: Selects saturate or full precision adder modes. Default is Saturate.

subtract: When enabled the adder operation is changed from A+B to A-B. Default is add.

supersample: Sets the supersample amount. Variable from 1 to 64. Default is 1.

Adder_streamFC

Signed adder with streaming interface with full flow control support.

Inputs are expected to have the same length.

Overflow and underflow check is done when saturate is enabled.

Output width is increased by 1 when full precision is enabled.

Subtraction changes operation from A+B to A-B.

This module adds a delay of 4 cycles.

Parameters

Input Width: Sets the bus width of the A and B inputs. Default is 16.

User Width: Sets the bus width of the tuser input. Variable between 1 and 8. Default is 1.

Adder Implementation: Selects saturate or full precision adder modes. Default is Saturate.

Subtract: When enabled the adder operation is changed from A+B to A-B. Default is add.

Supersample: Sets the supersample amount. Variable from 1 to 64. Default is 1.

Comparison



Comparisons between inputs A and B.

Output is set to one when the comparison set by the operation parameter is true.

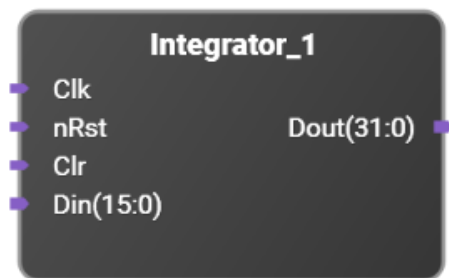
Parameters

operation: Select between A==B, A!=B, A>B, A<B, A>=B, and A<=B. Default is A==B.

data size: Sets the bus width of the A and B inputs. Default is 16.

sign: Select when the data on the A and B inputs is signed. Default is unsigned.

Integrator



Data integrator.

When selecting signed input, sign extension is automatically applied.

The internal accumulator can be reset by the nRst or Clr inputs.

When supersample > 1, all the input samples are summed into the same internal accumulator.

This module adds a delay of 1 cycle by default.

When latch input is enabled, an extra cycle of delay is added.

Parameters

input_width: Sets the bus width of the input samples. Variable from 1 to 1024. Default is 16.

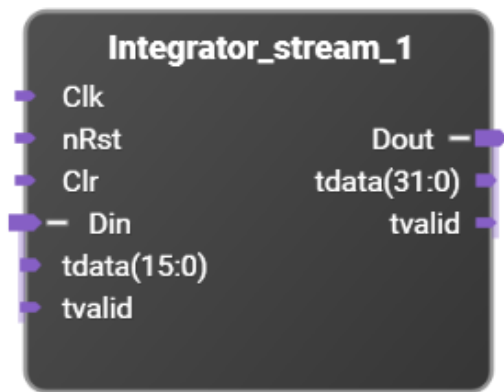
output_width: Sets the bus width of the internal accumulator and the output. Variable from 1 to 1024. Default is 32. The output_width must be greater than or equal to input_width.

input_signed: When enabled, the input samples represent signed values and will be sign extended prior to accumulation. Default is unsigned.

latch input: When enabled, the input data is latched prior to accumulation. This adds a cycle of delay. Default is no latch.

supersample: Sets the supersample amount. All the input samples are summed into the same internal accumulator. Variable from 1 to 64. Default is 1.

Integrator_stream



Data integrator with streaming interface.

When selecting signed input, sign extension is automatically applied.

The input samples are accumulated only when the Din tvalid signal is asserted.

The internal accumulator can be reset by the nRst or Clr inputs.

When supersample > 1, all the input samples are summed into the same internal accumulator.

This module adds a delay of 1 cycle by default.

When latch input is enabled, an extra cycle of delay is added.

Parameters

input_width: Sets the bus width of the input samples. Variable from 1 to 1024. Default is 16.

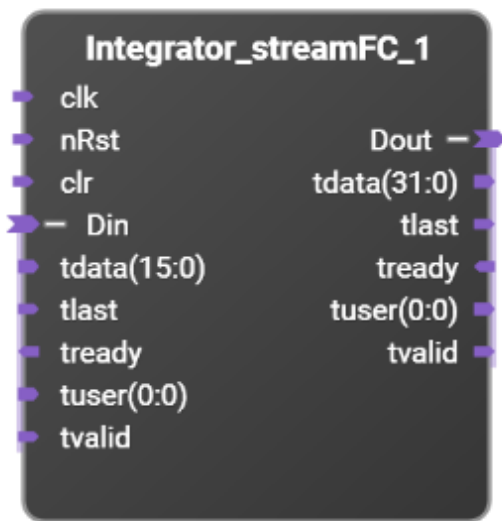
output_width: Sets the bus width of the internal accumulator and the output. Variable from 1 to 1024. Default is 32. The output_width must be greater than or equal to input_width.

input_signed: When enabled, the input samples represent signed values and will be sign extended prior to accumulation. Default is unsigned.

latch input: When enabled, the input data is latched prior to accumulation. This adds a cycle of delay. Default is no latch.

supersample: Sets the supersample amount. All the input samples are summed into the same internal accumulator. Variable from 1 to 64. Default is 1.

Integrator_streamFC



Data integrator with streaming interface with full flow control support.

When selecting signed input, sign extension is automatically applied.

The internal accumulator can be reset by the nRst or Clr inputs.

The Clr input will only clear the internal accumulator but allow input samples to pass through while asserted.

When supersample > 1, all the input samples are summed into the same internal accumulator.

This module adds a delay of 3 cycles by default.

When latch input is enabled, an extra cycle of delay is added.

Parameters

input_width: Sets the bus width of the input samples. Variable from 1 to 1024. Default is 16.

output_width: Sets the bus width of the internal accumulator and the output. Variable from 1 to 1024. Default is 32. The output_width must be greater than or equal to input_width.

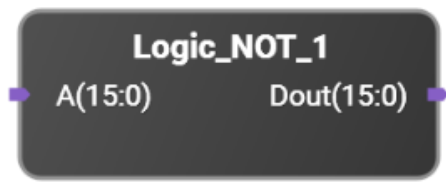
tuser_width: Sets the bus width of the tuser input. Variable between 1 and 8. Default is 1.

input_signed: When enabled, the input samples represent signed values and will be sign extended prior to accumulation. Default is unsigned.

latch input: When enabled, the input data is latched prior to accumulation. This adds a cycle of delay. Default is no latch.

supersample: Sets the supersample amount. All the input samples are summed into the same internal accumulator. Variable from 1 to 64. Default is 1.

Logic_NOT

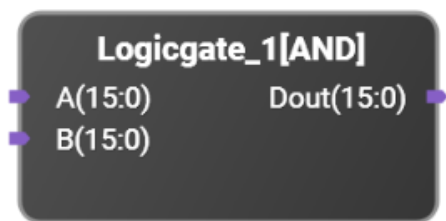


Logic NOT operation.

Parameters

data size: Sets the bus width of the A and Dout ports. Variable from 1 to 1024. Default is 16.

Logicgate



Output is the logical operation between inputs A and B.

The operation parameter determines which logical operation is performed from AND, OR, XOR, NAND, NOR, and XNOR.

Parameters

data size: Sets the bus width of the A, B, and Dout ports. Variable from 1 to 1024. Default is 16.

operation: Selects one of the logic operations listed above. Default is AND.

Multiplier



Multiplier (DSP core).

Input lengths and signedness are configurable.

When both inputs are signed, the multiplication product length is the sum of both input lengths minus 1. Otherwise, the product length is the sum of both input lengths.

When the Dout length is less than the product length, Dout will consist of the upper (most significant) bits of the product. The maximum Dout length is the product length.

When both inputs are signed, the product $-FullScale \times FullScale$ can not be represented in the output. Instead, $+FullScale$ is output, which is one less than the real product.

This block adds a delay of 1 cycle.

Latch input increases the total delay by an additional clock cycle.

Pipeline increases the total delay by an additional clock cycle.

Parameters

A width: Sets the bus width of the A input. Variable between 1 and 1024. Default is 16.

A signed: Select when the A input data is signed. Default is unsigned.

B width: Sets the bus width of the B input. Variable between 1 and 1024. Default is 16.

B signed: Select when the B input data is signed. Default is unsigned.

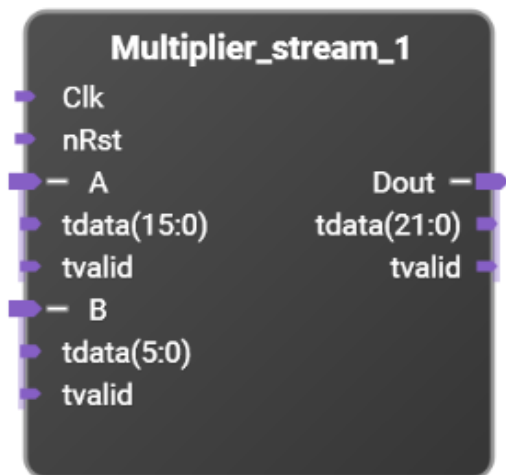
Dout width: Sets the bus width of the Dout port. Variable between 1 and 1024. Default is 16.

Latch input: Input data is latched when selected. Default is no latch.

supersample: Sets the supersample amount. Variable between 1 and 64. Default is 1.

pipeline: When selected a pipelined multiplier is used. Default is no pipelining.

Multiplier_stream



Multiplier (DSP core) with streaming interface.

Input lengths and signedness are configurable.

When both inputs are signed, the multiplication product length is the sum of both input lengths minus 1. Otherwise, the product length is the sum of both input lengths.

When the Dout length is less than the product length, Dout will consist of the upper (most significant) bits of the product. The maximum Dout length is the product length.

When both inputs are signed, the product $-FullScale \times FullScale$ can not be represented in the output. Instead, $+FullScale$ is output, which is one less than the real product.

This block adds a minimum delay of 1 cycle.

Pipeline increases the total delay by an additional clock cycle.

Parameters

A width: Sets the bus width of the A input. Variable between 1 and 1024. Default is 16.

A signed: Select when the A input data is signed. Default is unsigned.

B width: Sets the bus width of the B input. Variable between 1 and 1024. Default is 16.

B signed: Select when the B input data is signed. Default is unsigned.

Dout width: Sets the bus width of the Dout port. Variable between 1 and 1024. Default is 16.

pipeline: When selected a pipelined multiplier is used. Default is no pipelining.

supersample: Sets the supersample amount. Variable between 1 and 64. Default is 1.

Multiplier_streamFC



Multiplier (DSP core) with streaming interface and full flow control support.

Input lengths and signedness are configurable.

When both inputs are signed, the multiplication product length is the sum of both input lengths minus 1. Otherwise, the product length is the sum of both input lengths.

When the Dout length is less than the product length, Dout will consist of the upper (most significant) bits of the product. The maximum Dout length is the product length.

When both inputs are signed, the product -FullScale times -FullScale can not be represented in the output. Instead, +FullScale is output, which is one less than the real product.

This block adds a minimum delay of 4 cycles.

Pipeline increases the total delay by an additional clock cycle.

Parameters

A width: Sets the bus width of the A input. Variable between 1 and 1024. Default is 16.

A signed: Select when the A input data is signed. Default is unsigned.

B width: Sets the bus width of the B input. Variable between 1 and 1024. Default is 16.

B signed: Select when the B input data is signed. Default is unsigned.

Tuser width: Sets the bus width of the tuser input. Variable between 1 and 8. Default is 1.

Dout width: Sets the bus width of the Dout port. Variable between 1 and 1024. Default is 16.

pipeline: When selected a pipelined multiplier is used. Default is no pipelining.

supersample: Sets the supersample amount. Variable between 1 and 64. Default is 1.

Saturator



Output data is set to a saturation value (set by Thld port) whenever input data is equal or greater than that value.

For signed data, output data is set to a saturation value (-Thld) whenever input data is less than that value.

Saturation value can not be greater than the maximum possible value of the output vector.

Parameters

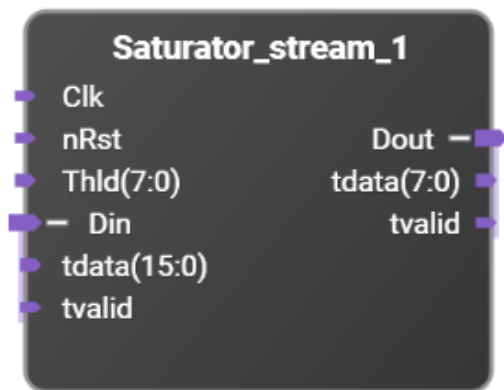
Din signed: Select when the data on the Din input is signed. Default is signed.

Din width: Sets the Din bus width. Variable between 1 and 1024. Default is 16.

Dout width: Sets the Dout bus width. Variable between 1 and 1024. Default is 8. The Dout width must be less than or equal to Din width.

supersample: Sets the supersample amount. Variable between 1 and 64. Default is 1.

Saturator_stream



Data saturator with streaming interface.

Output data is set to a saturation value (set by Thld port) whenever input data is equal or greater than that value.

For signed data, output data is set to a saturation value (-Thld) whenever input data is less than that value.

Saturation value can not be greater than the maximum possible value of the output vector.

Parameters

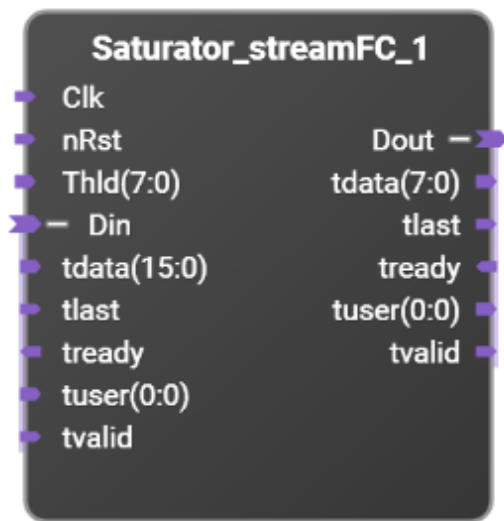
Din signed: Select when the data on the Din input is signed. Default is signed.

Din width: Sets the Din bus width. Variable between 1 and 1024. Default is 16.

Dout width: Sets the Dout bus width. Variable between 1 and 1024. Default is 8. The Dout width must be less than or equal to Din width.

supersample: Sets the supersample amount. Variable between 1 and 64. Default is 1.

Saturator_streamFC



Data saturator with streaming interface with full flow control support.

Output data is set to a saturation value (set by Thld port) whenever input data is equal or greater than that value.

For signed data, output data is set to a saturation value (-Thld) whenever input data is less than that value.

Saturation value can not be greater than the maximum possible value of the output vector.

This block adds a minimum delay of 3 cycles.

Parameters

Din signed: Select when the data on the Din input is signed. Default is signed.

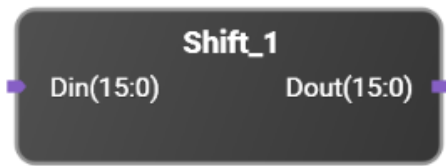
Din width: Sets the Din bus width. Variable between 1 and 1024. Default is 16.

Tuser width: Sets the tuser bus width. Variable between 1 and 8. Default is 1.

Dout width: Sets the Dout bus width. Variable between 1 and 1024. Default is 8. The Dout width must be less than or equal to Din width.

supersample: Sets the supersample amount. Variable between 1 and 64. Default is 1.

Shift



Signal shifter with configurable input size, direction and number of shifts.

This block does not introduce extra delay.

Zeros are introduced on the shifted side.

Parameters

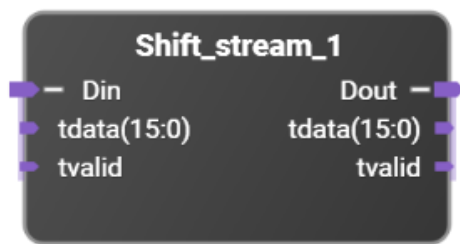
bus width: Sets the data width of the Din and Dout ports. Variable between 1 and 1024. Default is 16.

shift direction: Sets the direction to shift the Din data. Possible options are Left shift or Right shift. Default is Left shift.

shift amount: Sets the number of bits to shift. Default is 0.

supersample: Sets the supersample amount. Variable between 1 and 64. Default is 1.

Shift_stream



Signal shifter with configurable input size, direction and number of shifts using streaming interfaces.

This block does not introduce extra delay.

Zeros are introduced on the shifted side.

Parameters

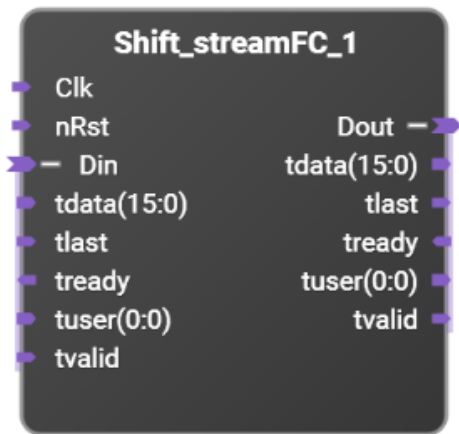
bus width: Sets the data width of the Din and Dout ports. Variable between 1 and 1024. Default is 16.

shift direction: Sets the direction to shift the Din data. Possible options are Left shift or Right shift. Default is Left shift.

shift amount: Sets the number of bits to shift. Default is 0.

supersample: Sets the supersample amount. Variable between 1 and 64. Default is 1.

Shift_streamFC



Signal shifter with configurable input size, direction and number of shifts using streaming interfaces with full flow control.

This block adds a minimum delay of 2 cycles.

Zeros are introduced on the shifted side.

Parameters

bus width: Sets the data width of the Din and Dout ports. Variable between 1 and 1024. Default is 16.

tuser width: Sets the tuser bus width. Variable between 1 and 8. Default is 1.

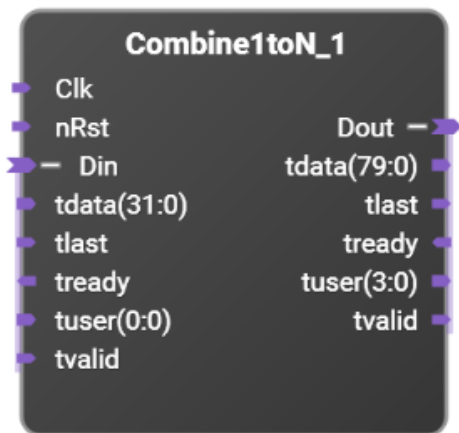
shift direction: Sets the direction to shift the Din data. Possible options are Left shift or Right shift. Default is Left shift.

shift amount: Sets the number of bits to shift. Default is 0.

supersample: Sets the supersample amount. Variable between 1 and 64. Default is 1.

DSP

Combine1toN



Combines N AXI-streaming samples into one AXI-streaming sample that is N times wider. The input is not supersampled while the output is supersampled by N.

Parameters

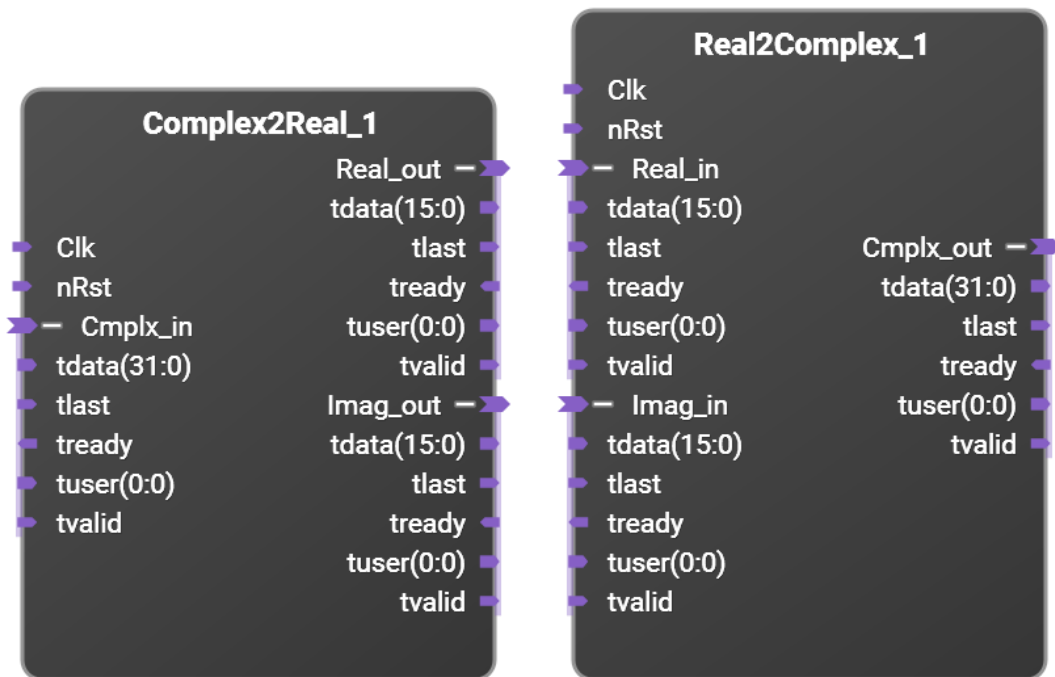
Tdata size: This sets the data width of Din_tdata. Dout_tdata will be N or N+1/2 times this value in width.

Tuser size: This sets the data width of Din_tuser. Dout_tuser will be N or N+2 times this value in width.

N: This sets how many input samples are combined into one output sample.

Add 1/2 to N: When selected, combine N+1/2 samples into the output rather than N samples.

Complex2Real / Real2Complex



Converts between one complex stream of data using interleaved real and imaginary parts and two separate streams, one for real and one for imaginary parts. These can be used to split off the real and imaginary streams into different destinations or to combine two real streams into one complex stream.

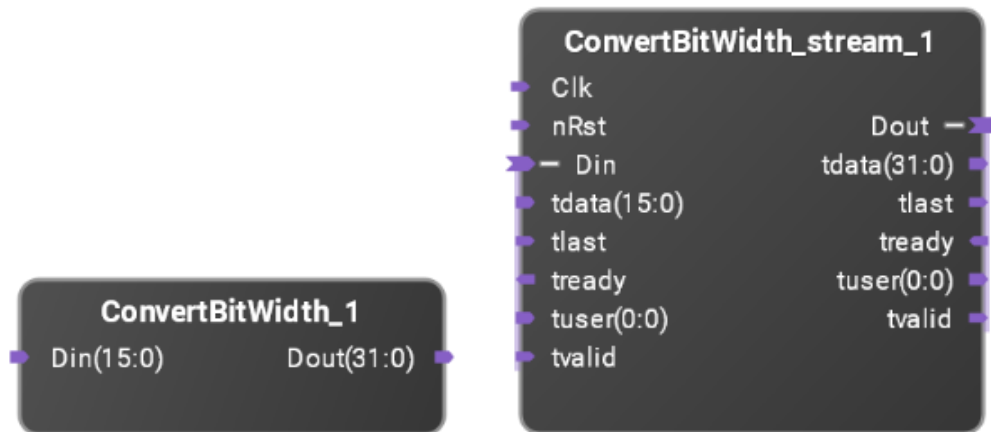
Parameters

Tdata size: This sets the data width of the real and imaginary parts of each sample.

Tuser size: This sets the tuser bits per sample.

supersample: This sets the number of samples per clock in the input and output streams.

ConvertBitWidth / ConvertBitWidth_stream



Convert sample data between different bit widths with or without rounding/clamping.

The justification of the data needs to be specified. If the data is left justified, then the MSBs of the input and output samples will be the same (subject to rounding). This is typically the case when the data is a fixed point value representing values between ± 1 . If the data is right justified, then the LSBs of the input and output will be the same (subject to clamping). This is typically the case when the data represents integers.

When the sample size increases (the output data width > input data width), then for left justified data the data is zero padded on the LSB side to extend the length, and for right justified data, the MSBs are either zero padded or sign extended, depending on whether the data is signed or not), on the MSB side to extend the length.

When the sample size decreases (the output data width < input data width), then there is the possibility of data loss. These IP blocks can do this safely by rounding and/or clamping as necessary. This adds some amount of logic latency but prevents data rollover and any truncation bias. If the user knows that the input data will fit in the output data size without problem, the IP can be set to not round or clamp. In this case the IP adds zero delay to the signal as it then becomes just wires. However in this case left justified data will be truncated possibly introducing a bias and right justified data may roll over.

The rounding algorithm used is convergent rounding. Numbers with a fractional part less than $1/2$ will round down. Numbers with a fractional part greater than $1/2$ will round up. Numbers with a fractional part of exactly $1/2$ will round towards the even integer. This rounding minimizes rounding noise while being unbiased for large enough signals.

The ConvertBitWidth IP block is purely combinatorial. If registering is needed for timing, then it should be added externally to this block. The ConvertBitWidth_stream IP block supports full flow controlled AXI-streaming interfaces. This block optionally always adds a register stage which can assist with timing closure. If this is not included, then the data path between the input and output is combinatorial. If the register stage is included, then this block will add latency.

Parameters

Supersample: This sets the number of samples per clock in the input and output streams.

Input Data Width: This sets the number of data bits in each input sample. The Din size is Supersample*Input Data Width bits for real data, and 2*Supersample*Input Data Width bits for complex data.

Output Data Width: This sets the number of data bits in each output sample. The Dout size is Supersample*Output Data Width bits for real data, and 2*Supersample*Output Data Width bits for complex data.

Justify: Indicates if the data is left justified (the MSBs are the more important) or right justified (the LSBs are the more important). The two choices are "Left (Keep MSBs)" and "Right (Keep LSBs)".

Round/Clamp: Indicates if the bit width conversion should use rounding/clamping or not. The two choices are "Round/Clamp" and "No Round/Clamp".

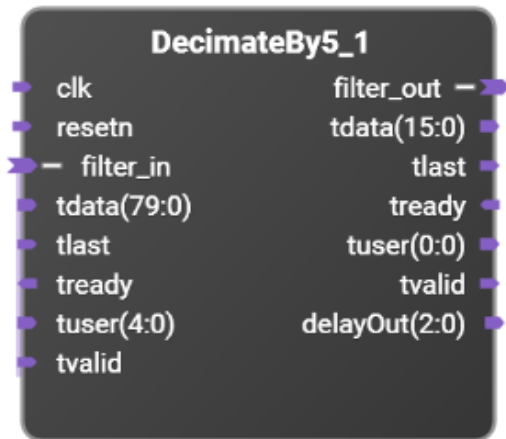
Signed Data: Indicates if the data is signed or unsigned.

Complex Data: Indicates if the data is complex or real only.

Register: When selected, this adds a register stage to the output, else the IP block is purely combinatorial (only on ConvertBitWidth_stream).

Tuser size: This sets the number of tuser bits per sample in the axi-streaming interface (only on ConvertBitWidth_stream).

DecimateBy5



Decimate 5x supersampled streaming input by a factor of 5. Decimation is achieved using a polyphase FIR filter. This block offers a choice of two filter bandwidths. The default 60% filter prevents any aliasing but only has a passband that is 60% of the Nyquist rate. The 80% filter's passband extends to 80% of Nyquist. With the 80% filter, the passband is alias protected though there may be aliasing in the transition band.

`tlast` may be used when an input sample stream includes packetized data.

`filter_in_tuser` may be used to tag a particular sample of the input stream. There are *User Data Width* bits for each of the five input samples.

`filter_out_tuser` will be asserted to indicate the corresponding sample of the output stream.

If *User Data Width* is greater than one, then the `tuser` input will have $5 * \text{User Data Width}$ bits and the `tuser` output will have *User Data Width* bits.

Parameters

Data Width: This sets the input and output sample widths. Note the input `tdata` width is $5 \times \text{Data Width}$ bits

User Data Width: This sets the number of TUSER data bits per sample. The use of these TUSER bits are used defined.

Bandwidth Select: This selects which filter to use with the allowable choices being either 60 (default) or 80.

Pipeline: (Rev 1.1 and later) When checked, this adds two extra pipeline stages in the calculations to facilitate timing.

DecimateBy5 Complex



Decimate a complex 5x supersampled streaming input by a factor of 5. Decimation is achieved using a polyphase FIR filter. The real and imaginary parts of each sample are interleaved with the real part occupying the less significant (lower bit number) word. The lower order *Data Width* bits of the output are real output data and the upper *Data Width* bits of the output are imaginary output data. This block offers a choice of two filter bandwidths. The default 60% filter prevents any aliasing but only has a passband that is 60% of the Nyquist rate. The 80% filter's passband extends to 80% of Nyquist. With the 80% filter, the passband is alias protected though there may be aliasing in the transition band.

`tlast` may be used when an input sample stream includes packetized data.

`filter_in_tuser` may be used to tag a particular sample of the input stream. There are *User Data Width* bits for each of the five input samples.

`filter_out_tuser` will be asserted to indicate the corresponding sample of the output stream. If *User Data Width* is greater than one, then the `tuser` input will have $5 * \text{User Data Width}$ bits and the `tuser` output will have *User Data Width* bits.

Parameters

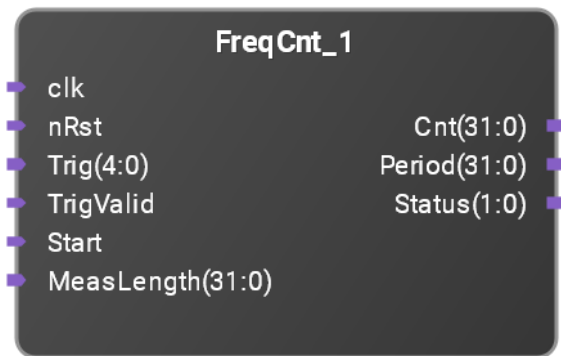
Data Width: This sets the input and output sample widths. Note the `filter_in_tdata` width is ten times *Data Width* bits, and the `filter_out_tdata` width is twice *Data Width* bits.

User Data Width: This sets the number of TUSER data bits per sample. The use of these TUSER bits are used defined.

Bandwidth Select: This selects which filter to use with the allowable choices being either 60 (default) or 80.

Pipeline: (Rev 1.1 and later) When checked, this adds two extra pipeline stages in the calculations to facilitate timing.

FreqCnt



FreqCnt is a frequency counter block. It measures an integral number of periods during the measurement interval and reports the number of trigger events (the number of signal periods) and the time between the first and last trigger event during the measurement interval. The input is the Trig port which consists of one bit per supersampled data. These trigger signals might come from the Trigger IP block. FreqCnt supports supersampled data.

MeasLength sets the length of the measurement interval in clock cycles (which is the sample rate divided by the supersample factor). Note that FreqCnt only counts clocks when TrigValid is asserted. For a given value of MeasLength, if the Trig values only come in every other clock, the elapsed time for the measurement will be twice as long as it would be if Trig values come in every clock.

Start begins a new measurement. Note that Start can be tied high in which case a new measurement will start immediately after the previous measurement. The Cnt and Period outputs are latched at the end of a measurement so they may be read while a new measurement is in progress. Depending on the *SyncStart* parameter, the measurement interval will either begin when Start is asserted (*SyncStart* = 0) or on the first trigger event after Start is asserted (*SyncStart* = 1).

Cnt outputs the number of whole signal periods during the measurement interval. This is one less than the total number of trigger events (each indicated by a Trig bit being 1).

Period outputs the time between the first and last trigger event measured in samples. The frequency of the input signal can then be calculated $f = f_s * Cnt / Period$ where f_s is the sample rate.

Status indicates the internal state of the measurement. Status[0] = 1 indicates that a measurement is in progress. Note that if Start is tied high, a measurement will almost always be in progress since the next measurement will start immediately after the previous one finishes. Status[1] = 1 indicates that the first trigger event has been observed. If *SyncStart* = 1 and Status[1:0] = 01, it means that the FreqCnt is awaiting the first trigger event which will then start the actual measurement interval.

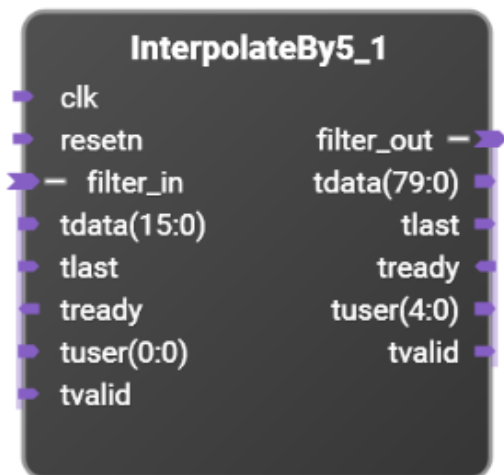
Parameters

Supersample: This sets the supersample value and determines how many parallel samples are processed at the same time. This also sets the bit width of the Trig port.

SyncStart: If set to one, the measurement interval will start on the first trigger event after Start has been asserted. If set to zero, the measurement interval starts immediately after Start has been asserted.

CntLen: This sets the bit width of the internal counters in the FreqCnt block. This also sets the size of the MeasLength, Cnt, and Period ports. It defaults to 32 bits.

InterpolateBy5



Interpolate an input stream by a factor of 5. Interpolation is achieved using an oversampled FIR filter. This block offers a choice of two filter bandwidths. The default 60% filter prevents any aliasing but only has a passband that is 60% of the Nyquist rate. The 80% filter's passband extends to 80% of Nyquist. With the 80% filter, the passband is alias protected though there may be aliasing in the transition band.

tlast may be used when an input sample stream includes packetized data.

filter_in_tuser may be used to tag a particular sample of the input stream.

filter_out_tuser will be asserted to indicate the corresponding sample of the output stream. There are *User Data Width* bits for each of the five input samples.

If *User Data Width* is greater than one, then the tuser input will have *User Data Width* bits and the tuser output will have $5 * \text{User Data Width}$ bits.

Parameters

Data Width: Sets the bus width of filter_in_tdata. Default is 16.

User Data Width: This sets the number of TUSER data bits per sample. The use of these TUSER bits are used defined.

Bandwidth Select: This selects which filter to use with the allowable choices being either 60 (default) or 80.

InterpolateBy5 Complex



Interpolate a complex input stream by a factor of 5. Interpolation is achieved using an oversampled FIR filter. This block offers a choice of two filter bandwidths. The default 60% filter prevents any aliasing but only has a passband that is 60% of the Nyquist rate. The 80% filter's passband extends to 80% of Nyquist. With the 80% filter, the passband is alias protected though there may be aliasing in the transition band.

tlast may be used when an input sample stream includes packetized data.

filter_in_tuser may be used to tag a particular sample of the input stream.

filter_out_tuser will be asserted to indicate the corresponding sample of the output stream.

There are *User Data Width* bits for each of the five input samples.

If *User Data Width* is greater than one, then the tuser input will have *User Data Width* bits and the tuser output will have $5 * \text{User Data Width}$ bits.

Parameters

Data Width: Sets the data width for each of the real and imaginary samples. Default is 16 (32 total bits for I and Q data). The filter_in_tdata will be twice this size and the filter_out_tdata will be ten times this size.

User Data Width: This sets the number of TUSER data bits per sample. The use of these TUSER bits are used defined.

Bandwidth Select: This selects which filter to use with the allowable choices being either 60 (default) or 80.

Lo



Parameterized Local Oscillator/Mixer. By default it generates a local oscillator signal and mixes (multiplies) this LO signal with the X input to generate the Y output. Optionally the block can output just the local oscillator signal without mixing it with the input.

The LO block handles supersampled or non-supersampled data, and either the input and/or the output can be real or complex. Additionally, the output mixer may be bypassed so that the output Y interface outputs a full scale local oscillator signal (the sine and cosine values). In this case, the input X interface is ignored.

A and B control the local oscillator's frequency. Let S be the amount of supersampling, and let T be the smallest power of 2 greater than or equal to S (so that $S \leq T < 2S$).

The LO frequency is given by $f = f_s * (A+B/5^{10})/((S/T)^{2^{25}})$. Note that f_s represents the signal's sample rate, not the FPGA clock rate. For a sample rate, f_s , of 1 Gs/s, this results in an even decimal frequency resolution of 0.1 Hz. Frequencies can be positive or negative. Valid input ranges for A and B are such that $-1/2 \leq f/f_s \leq 1/2$. Values of A and B that are outside this range will give incorrect results.

When asserted, phRst will reset the phase of the phase accumulator to zero and flush data in the LO's pipelines without resetting the programmed frequency. For phase continuous frequency changes, leave phRst negated. Note that in this case due to pipeline stages, the results of the frequency change will not be visible at the output for several samples. To eliminate this delay in seeing the effects of frequency changes, assert phRst on or after the new frequency is set. This may easily be done by driving phRst with the same signal as setFreq. Note that phRst should not be tied high as that will prevent operation of the LO.

Parameters

Tdata size: Sets the data width for each sample (real data) or for each of the real and imaginary parts of each sample (complex data).

Tuser size: Sets the number of tuser bits per sample.

Complex Input: If set, then the input data is complex. If cleared, then the input data is real only.

Complex Output: If set, then the output data is complex. If cleared, then only the real part of the output data is generated.

Supersample: This sets the supersample value and determines how many parallel samples are processed at the same time.

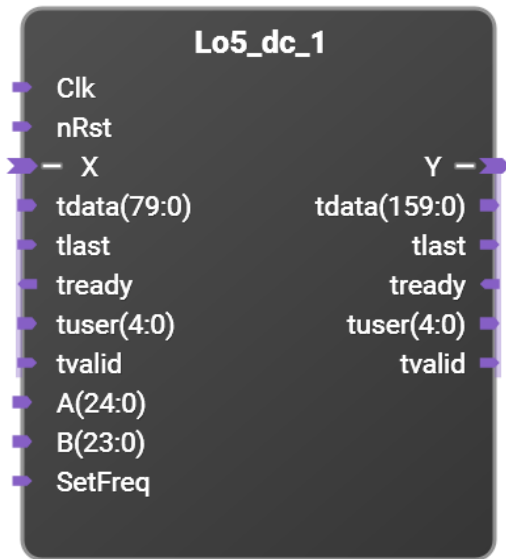
Shift Direction: If Shift Direction = 0, the input is multiplied by $e^{j\omega t}$ (shift frequencies up).
 If Shift Direction = 1, the input is multiplied by $e^{-j\omega t}$ (shift frequencies down).

Dither: Enables phase dithering to help convert spurious signals into more noise like signals (default is Dither=1 or enabled).

Trig Only: Enables output of the LO's trig functions and bypasses the mixer. If enabled, the X interface is ignored.

Pipeline: (Version 1.1 and later) When checked, this adds an extra pipeline stages in the phase accumulator and sine/cosine calculation to facilitate timing (default is Pipeline=1 or enabled).

Lo5_dc



Note: This block is deprecated and not recommended for new designs. New designs should use the Lo block instead.

Down converting Local Oscillator for use in digitizers with 5X supersampled ADCs. Input is 5X supersampled real data while the output is a 5X supersampled data stream representing complex output data.

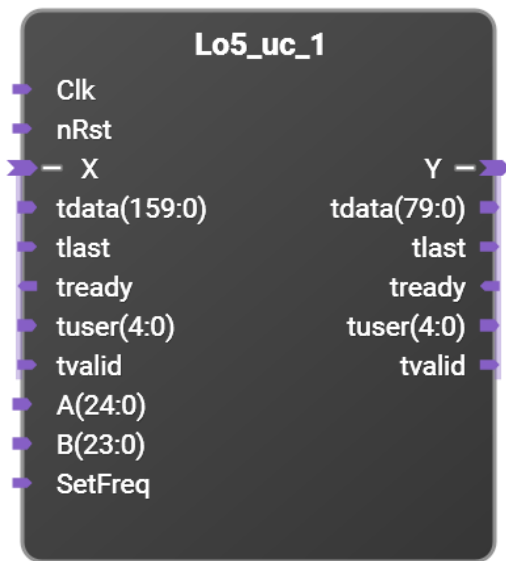
A and B control the local oscillator's frequency.

The LO frequency is given by $f = f_s * (A+B/5^{10})/(5*2^{22})$. For a sample rate, f_s , of 1 Gs/s, this results in an even decimal frequency resolution of 0.01 Hz. Note that for this block f/f_s is limited to the range +/- 0.4.

Parameters

Tdata size: This sets the data width of the samples. Since the data is 5X supersampled, the input tdata width is five times this value and the output tdata width is ten times this value.

Tuser size: This sets the tuser bits per sample.

Lo5_uc

Note: This block is deprecated and not recommended for new designs. New designs should use the **Lo** block instead.

Up converting Local Oscillator for use in sources with 5X supersampled DACs. Input is a 5X supersampled data stream representing complex input data. Output is one 5X supersampled real data stream.

A and B control the local oscillator's frequency.

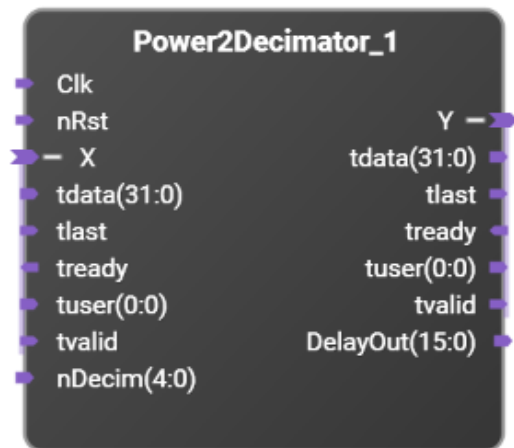
The LO frequency is given by $f = f_s * (A+B/5^{10})/(5*2^{22})$. For a sample rate, f_s , of 1 Gs/s, this results in an even decimal frequency resolution of 0.01 Hz. Note that for this block f/f_s is limited to the range +/- 0.4.

Parameters

Tdata size: This sets the data width of the samples. Since the data is 5X supersampled, the input tdata width is five times this value and the output tdata width is ten times this value.

Tuser size: This sets the tuser bits per sample.

Power2Decimator



This is a power of two decimation filter that operates on real or complex data. It accepts real or complex data at up to one sample per clock. It filters and decimates the data by 2^N , where $N=0\dots16$. It offers a choice of two filter bandwidths. The default 60% filter prevents any aliasing but only has a passband that is 60% of the Nyquist rate. The 80% filter is a halfband filter with flatter passband that extends to 80% of Nyquist. With the 80% filter, the passband is alias protected though there may be aliasing in the transition band.

Parameters

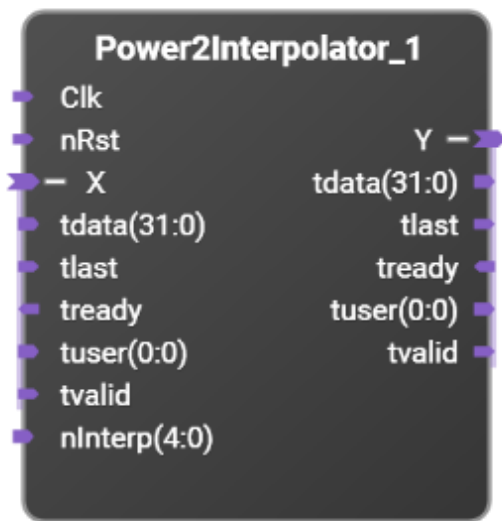
Tdata size: This sets the data width of the samples. For a real filter, this is the size of the tdata buses. For a complex filter, the widths of the tdata busses are twice this value.

Tuser size: This sets the tuser bits per sample.

Bandwidth Select: This selects which filter to use with the allowable choices being either 60 (default) or 80.

Complex: If this is checked, then the filter operates on complex data. If unchecked, then the filter operates on real only data.

Power2Interpolator



This is a power of two interpolation filter that operates on real or complex data. It accepts data and interpolates and filters the data by 2^N , where $N=0\ldots 16$, generating up to one real or complex output sample per clock. It offers a choice of two filter bandwidths. The default 60% filter prevents any aliasing but only has a passband that is 60% of the Nyquist rate. The 80% filter is a halfband filter with flatter passband that extends to 80% of Nyquist. With the 80% filter, the passband is alias protected though there may be aliasing in the transition band.

Parameters

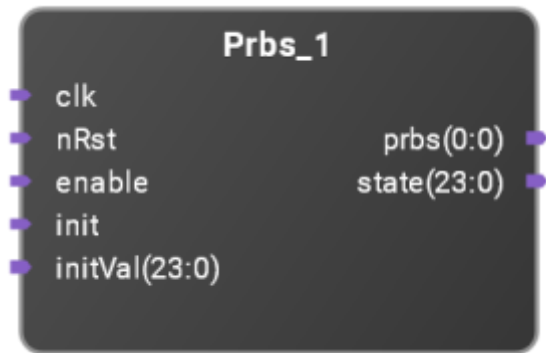
Tdata size: This sets the data width of the samples. For a real filter, this is the size of the tdata buses. For a complex filter, the widths of the tdata busses are twice this value.

Tuser size: This sets the tuser bits per sample.

Bandwidth Select: This selects which filter to use with the allowable choices being either 60 or 80.

Complex: If this is checked, then the filter operates on complex data. If unchecked, then the filter operates on real only data.

PRBS - Pseudo Random Bit Sequence



This is a Pseudo Random Bit Sequence generator. It can generate both Galois and Fibonacci PRBS sequences using a customizable polynomial. It can generate one or multiple new bits every clock.

The size of the internal state register is determined by the polynomial chosen. The number of bits in the state register is $\log_2(\text{polynomial})$. The number of new PRBS bits to output can be independently chosen.

When nRst is asserted, the internal state register is assigned the resetVal value. Otherwise if init is asserted, the internal state register is set to initVal. Otherwise if enable is asserted the state register is advanced multiple bits as specified by the Advance parameter.

The parameters for the polynomial and reset value are broken into two separate parameters each, one consisting of the MSBs (bits 63:32) and one consisting of the LSBs (bits 31:0).

Parameters

Polynomial (MSBs): This is bits 63:32 of the polynomial to be used.

Polynomial (LSBs): This is bits 31:0 of the polynomial to be used.

Nbits: This is the number of new PRBS bits to generate each clock cycle. Nbits=1 generates one new bit each clock, whereas Nbits=8 will advance the state register 8 iterations to generate 8 new bits each clock.

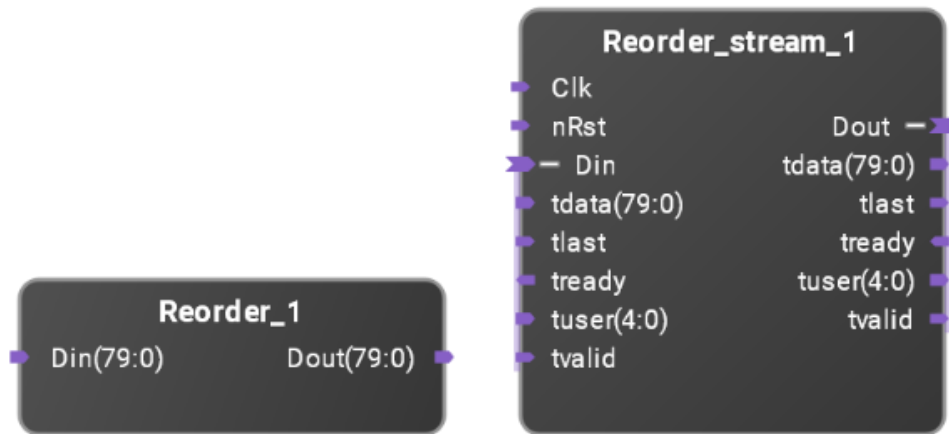
Type: This determines the type of LFSR to use. Type=0 is a Galois LFSR. Type=1 is a Fibonacci LFSR.

Reset value (MSBs): This is bits 63:32 of the Reset value to be used when nRst is asserted.

Reset value (LSBs): This is bits 31:0 of the Reset value to be used when nRst is asserted.

XNOR: When checked, this uses XNORs instead of XORs in the feedback path.

Reorder / Reorder_stream



These blocks will reorder the samples in supersampled data. These blocks can reserve the sample ordering within the supersampled word as well as optionally swap real/imaginary parts of each sample.

Reversing the samples would change {x5,x4, x3, x2, x1} to {x1,x2,x3,x4,x5}.

Reversing complex data would change {i3,r3,i2,r2,i1,r1} to {i1,r1,i2,r2,i3,r3}. Note that the complex samples are reordered, but the ordering of real/imag is preserved.

Reversing and swapping complex data would change {i3,r3,i2,r2,i1,r1} to {r1,i1,r2,i2,r3,i3}. Note that the complex samples are reordered, and the ordering of real/imag is swapped.

Just swapping complex data would change {i3,r3,i2,r2,i1,r1} to {r3,i3,r2,i2,r1,i1}. Note that the order of complex samples is preserved, and the ordering of real/imag is swapped.

Both of these blocks are purely re-routing. There is no logic or delay/latency added. The **Reorder_stream** block does not use the **Clk** or **nRst** ports. However, these ports are included for compliance with the AXI-streaming specification.

Parameters

Supersample: This sets the number of samples per clock in the input and output streams.

Data Width: This sets the number of data bits in each sample. The **Din/Dout** sizes are **Supersample*Data Width** bits for real data, and **2*Supersample*Data Width** bits for complex data.

Reverse: This will reverse the order of samples in the supersampled word.

Complex Data: Indicates if the data is complex or real only.

Swap: This will swap the real and imaginary parts of complex data. This is ignored for real data.

Register: When selected, this adds a register stage to the output, else the IP block is purely combinatorial (only on **ConvertBitWidth_stream**).

Tuser size: This sets the number of tuser bits per sample in the axi-streaming interface (only on **ConvertBitWidth_stream**).

ReshapeM1 / ReshapeP1



These blocks will reshape supersampled streaming data streams to convert between an odd number of samples per clock and an even number of samples per clock. For reshapeM1, dataOut will have one less (Minus 1) samples per clock than dataIn. For reshapeP1, dataOut will have one more (Plus 1) samples per clock than dataIn. Typically reshapeP1 is used between multiple instances of ssDecim2 since ssDecim2 requires the input supersample value to be even. Typically reshapeM1 is used between multiple instances of ssInterp2 since ssInterp2 always has an even output supersample value. Note that the reshapeP1 block only supports forward flow control (TVALID) and does not support reverse flow control (TREADY). The reshapeM1 block does support reverse flow control (TREADY).

Parameters

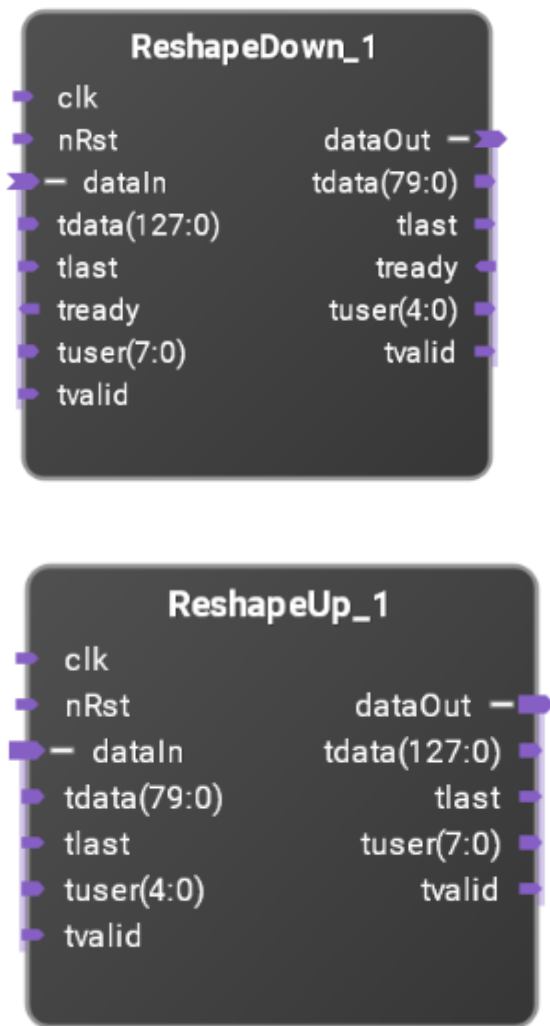
Supersample In: This sets the input supersample value (how many parallel samples per clock on the dataIn interface). For the reshapeM1 block the output supersample value is one less than this. For the reshapeP1 block, the output supersample value is one more than this.

Data size: This sets the data width of each sample. For real data, the width of the dataIn/tdata port will be (Supersample In)*(Data size) while for complex data, the width of the dataIn/tdata port will be 2*(Supersample In)*(Data size).

Tuser size: This sets the number of tuser bits per sample. The width of the dataIn/tuser port will be (Supersample In)*(Tuser size).

Complex: If checked, this indicates that the data is complex. If unchecked, then the data is real only.

ReshapeDown / ReshapeUp



These blocks will reshape supersampled streaming data streams to convert between different numbers of samples per clock. With reshapeDown, dataOut will have fewer samples per clock than dataIn. For reshapeUp, dataOut will have more samples per clock than dataIn. Note that the reshapeUp block only supports forward flow control (TVALID) and does not support reverse flow control (TREADY). The reshapeDown block does support reverse flow control (TREADY).

Parameters

Supersample In: This sets the input supersample value (how many parallel samples per clock on the dataIn interface).

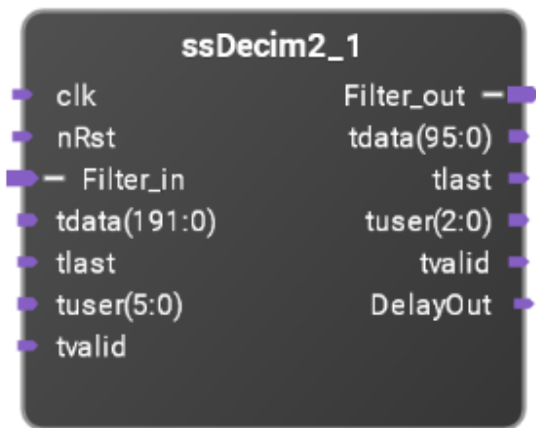
Supersample Out: This sets the output supersample value (how many parallel samplers per clock on the dataOut interface). For reshapeDown, Supersample Out must be less than Supersample In. For reshapeUp, Supersample Out must be greater than Supersample In.

Data size: This sets the data width of each sample. For real data, the width of the dataIn/tdata port will be (Supersample In)*(Data size) while for complex data, the width of the dataIn/tdata port will be 2*(Supersample In)*(Data size).

Tuser size: This sets the number of tuser bits per sample. The width of the dataIn/tuser port will be (Supersample In)*(Tuser size).

Complex: If checked, this indicates that the data is complex. If unchecked, then the data is real only.

ssDecim2



This is a supersampled half-band decimate by two block. It can operate on real or complex supersampled data. The input supersample value must be an even number. If desired, the reshapeP1 block can be used to convert from an odd supersample value to the required even supersample value. The output supersample value is half of the input supersample value.

The characteristics of the filter used in the decimator can be selected from a family of choices. The stopband attenuation can be chosen as either 70dB or 80dB. The fractional passband size can be selected from 10%, 20%, 40%, or 80% of Nyquist. This allows trading off filter performance and resource utilization.

The step response of these filters is not monotonic. The step response will show some amount of ringing. The DC gain through the filter can be set to one (if Shift Input is not checked) or one-half (if Shift Input is checked). To avoid clipping for an arbitrary input, the Shift Input parameter should be checked to set the DC gain to one-half. However, if the input signal is known to not cause clipping (e.g. if the ssDecim2 block is preceded by another ssDecim2 block) then the Shift Input parameter should not be checked to set the DC gain to one.

This block only supports forward flow control. It does not support reverse flow control (TREADY).

Parameters

Supersample: This sets the input supersample value (how many parallel samples per clock on the Filter_In interface). This must be an even number for this block. The output supersample value will be half this value.

Data size: This sets the data width of each sample. For real data, the width of the Filter_In/tdata port will be (Supersample)*(Data size) while for complex data, the width of the Filter_In/tdata port will be 2*(Supersample)*(Data size).

Tuser size: This sets the number of tuser bits per sample. The width of the Filter_In/tuser port will be (Supersample)*(Tuser size).

Complex: If checked, this indicates that the data is complex. If unchecked, then the data is real only.

Shift Input: If checked, the input data is shifted down one bit to set the DC gain of the filter to one-half. This can be useful to avoid output clipping. If not checked, the DC gain of the filter is unity.

Bandwidth Select: This chooses how wide the passband of the filter should be. Choices are 10%, 20%, 40%, and 80% of Nyquist.

Stop Band Rejection: This chooses the amount of stopband rejection in the filter. Choices are 70dB and 80dB.

Duplicate Resets: If checked, this will internally duplicate the reset signal. This incurs a one clock latency in the reset signal. This can be used in high speed designs with large supersample values to help timing. It defaults to off.

ssInterp2 / ssInterp2fc

These are supersampled half-band interpolate by two blocks. They can operate on real or complex supersampled data. The output supersample value is twice of the input supersample value.

The characteristics of the filter used in the interpolator can be selected from a family of choices. The stopband attenuation can be either 70dB or 80dB. The fractional passband size can be selected from 10%, 20%, 40%, or 80% of Nyquist. This allows trading off filter performance and resource utilization.

The step response of these filters is not monotonic. The step response will show some amount of ringing. The DC gain through the filter can be set to one (if Shift Input is not checked) or one-half (if Shift Input is checked). To avoid clipping for an arbitrary input, the Shift Input parameter should be checked to set the DC gain to one-half. However, if the input signal is known to not cause clipping (e.g. if the `ssInterp2` block is preceded by another `ssInterp2` block) then the Shift Input parameter should not be checked to set the DC gain to one.

The `ssInterp2` block only supports forward flow control. It does not support reverse flow control (TREADY). The `ssInterp2fc` block includes extra hardware in order to support reverse flow control (TREADY). This is required if this block will drive the `reshapeM1` block.

Parameters

Supersample: This sets the input supersample value (how many parallel samples per clock on the `Filter_In` interface). The output supersample value will be twice this value (and hence always even).

Data size: This sets the data width of each sample. For real data, the width of the `Filter_In/tdata` port will be $(\text{Supersample}) \times (\text{Data size})$ while for complex data, the width of the `Filter_In/tdata` port will be $2 \times (\text{Supersample}) \times (\text{Data size})$.

Tuser size: This sets the number of tuser bits per sample. The width of the `Filter_In/tuser` port will be $(\text{Supersample}) \times (\text{Tuser size})$.

Complex: If checked, this indicates that the data is complex. If unchecked, then the data is real only.

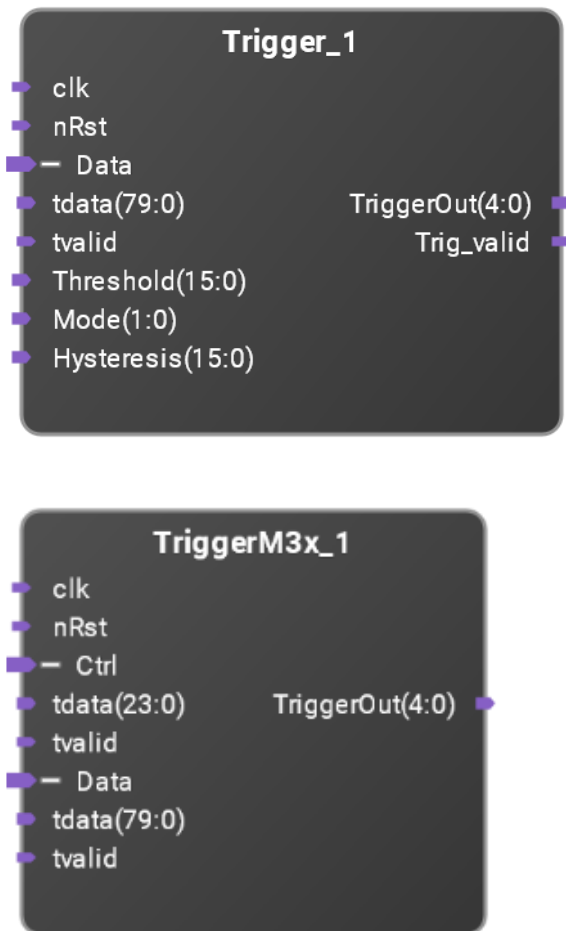
Shift Input: If checked, the input data is shifted down one bit to set the DC gain of the filter to one-half. This can be useful to avoid output clipping. If not checked, the DC gain of the filter is unity.

Bandwidth Select: This chooses how wide the passband of the filter should be. Choices are 10%, 20%, 40%, and 80% of Nyquist.

Stop Band Rejection: This chooses the amount of stopband rejection in the filter. Choices are 70dB and 80dB.

Duplicate Resets: If checked, this will internally duplicate the reset signal. This incurs a one clock latency in the reset signal. This can be used in high speed designs with large supersample values to help timing. It defaults to off.

Trigger / TriggerM3x



Trigger is an analog trigger block that supports hysteresis. It will generate a single clock wide TriggerOut pulse for each detected trigger event. Triggers can be off (Mode=0), rising edge (Mode=1), falling edge (Mode=2), or both (Mode=3). A rising edge trigger is detected when the input signal goes below Threshold-Hysteresis and then rises above Threshold. A falling edge trigger is detected when the input signal goes above Threshold+Hysteresis and then falls below Threshold. Both edge triggers will detect both of these event. The hysteresis amount is an unsigned value that can either be set at run-time or can be connected to a constant value.

The latency through the Trigger block from Data/tdata to TriggerOut is 2 clock cycles. Trig_valid is Data/tdata delayed by two clocks. Trig_valid indicates when TriggerOut may be driven. When Trig_valid is negated, TriggerOut will always be driven to zero.

TriggerM3x is the Trigger block inside a wrapper that makes the ports of TriggerM3x compatible with the ports of the AnalogTrigger blocks in the M3xxxx series of digitizers. The sample size is fixed at 16 bits, and the amount of hysteresis is fixed as a parameter to the block. TriggerM3x has a "supersample" parameter that should be set to 1 or to 5 depending on whether the digitizer module is supersampled or not.

Parameters

Supersample: This sets the input supersample value (how many parallel samples per clock). This also sets the bit width of the TriggerOut port.

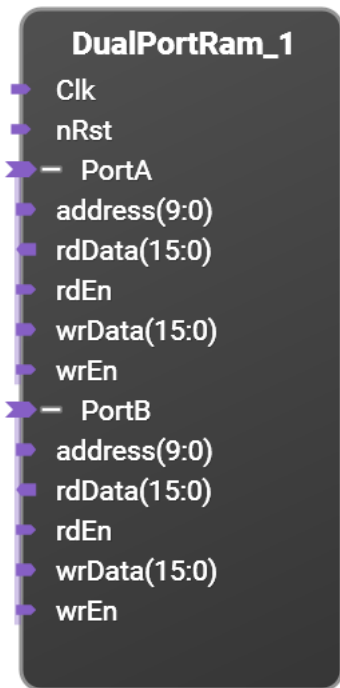
Datasize: This sets the data width of each sample. The width of the Data/tdata port will be (Supersample)*(Data size). [Trigger block only]

Signed: If checked, then the data and threshold are signed values. If not checked, then data and threshold are unsigned. [Trigger block only]

Hysteresis: This unsigned value sets the amount of hysteresis in the trigger. [TriggerM3x block only]

Memory

DualPortRam



Dual port Block Ram up to 1024 bits x 65536 positions using PC MEM interfaces.

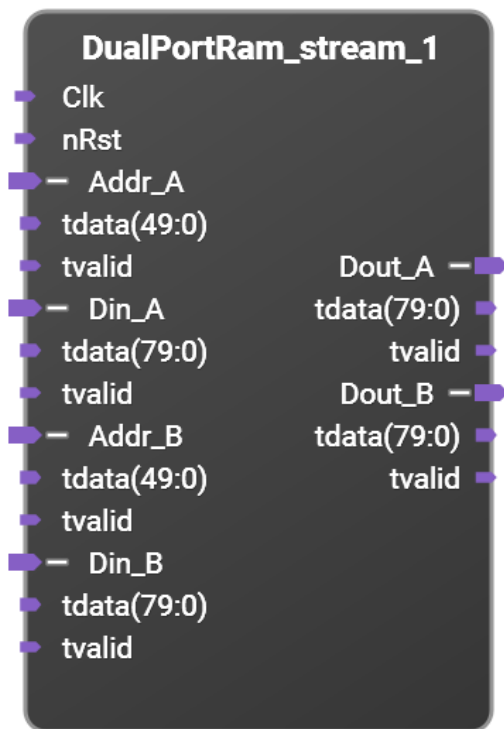
Read latency is 1 cycle.

Parameters

Data width: Sets the PortA and PortB data widths. Variable between 1 and 1024. Default is 16.

Address width: Sets the PortA and PortB address widths. Variable between 1 and 16. Default is 10.

DualPortRam_stream



Dual port Block Ram up to 1024 bits x 65536 positions using AXI Streaming interfaces.

Note that the tvalid for Addr and Din inputs must be asserted high and low at the same time for interfaces A or B.

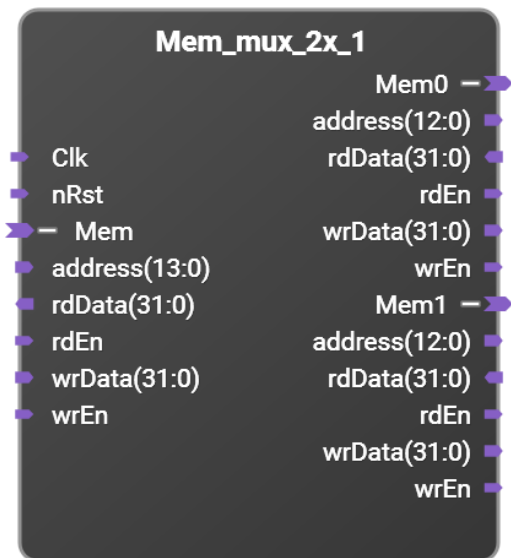
Read latency is 1 cycle.

Parameters

Data width: Sets the PortA and PortB data widths. Variable between 1 and 1024. Default is 16.

Address width: Sets the PortA and PortB address widths. Variable between 1 and 16. Default is 10.

supersample: Sets the supersample amount. Variable between 1 and 64. Default is 5.

Mem_mux_2x

MEM interface 1 to 2 multiplexor.

Input address space size = $2^{(\text{AXI4Slave Address Width})}$

Output address space size = Input address space size / 2

MEM0 offset = 0.

MEM1 offset = Output address space size.

Parameters

AXI4 Slave Address Width: Sets the address width on the Mem interfaces. Variable between 2 and 32. Default is 14.

Mem_mux_4x

MEM interface 1 to 4 multiplexor.

Input address space size = $2^{(\text{AXI4 Slave Address Width})}$

Output address space size = Input address space size / 4

MEM0 offset = 0.

MEM1 offset = 1*Output address space size.

MEM2 offset = 2*Output address space size.

MEM3 offset = 3*Output address space size.

Parameters

AXI4 Slave Address Width: Sets the address width on the Mem interfaces. Variable between 2 and 32. Default is 14.

As a convenience, the Streamer block can be configured to automatically generate this TLAST signal (in which case the external TLAST port signal is ignored). In order for this to correctly work, the stream must be inactive (that is, the TVALID signal is negated) until **after** the DMA transfer is started. As an example, if data is being read from DDR, sent to the stream (via the DDRtoStr interface), processed, and the results sent back to DDR (via the StrToDDR interface), then the StrToDDR DMA operation should be started **before** the DDRtoStr is started. When autoTlast is asserted, an internal counter will count the samples after the DMA is started and internally assert TLAST on the appropriate sample.

On the streaming interface side of the IP block the number of clock cycles per data word and the number of data words per packet burst vary depending on the hardware module used and the number of IP blocks accessing the DDR interface. The streamer efficiency is calculated as the total number of data words transferred divided by the total number of clock cycles for a streaming transaction. An efficiency of 100% would mean each data word only required one clock cycle to transfer. The streamer was benchmarked and will run with about 97% efficiency on the M3302A module. This performance was measured with all 4 streaming interfaces running using the M3302A 100 MHz clock and when no other IP was accessing the DDR. The efficiency of the streamer IP block may be less on other modules or when other IP blocks are accessing the DDR interface.

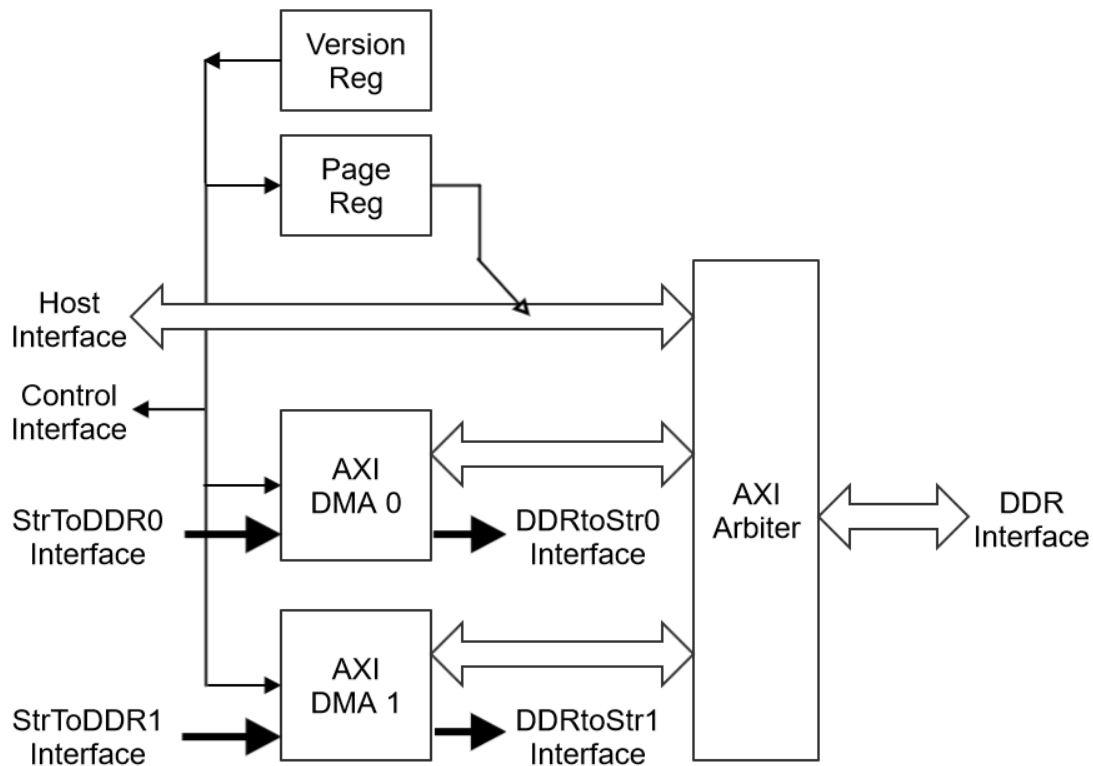
Parameters

autoTlast: if this box is checked, then the Streamer block will automatically generate the TLAST signal for the StrToDDR0/1 ports. In this case the TLAST signal supplied to the StrToDDR0/1 interface is ignored.

Signals

Signal name	Width (bits)	Description
clock	1	Clock input
nRst	1	Reset input (active low)
host	Multiple	Host AXI-MM slave interface with 17 address bits and 32 data bits for random access to DDR memory. This should be connected to a Host_aximm interface.
ctrl	Multiple	Control AXI-Lite slave interface with 12 address bits and 32 data bits for accessing the control registers in the streamer and DMA blocks. This should be connected to a Host_axilite interface.
DDR	Multiple	DDR AXI-MM master interface with 32 address bits and 128 data bits for accessing DDR memory. This should be connected to the DDR interface.
DDRtoStr0	Multiple	The channel 0 AXI-streaming master interface. Data from the DDR will stream out this interface using flow control.
DDRtoStr1	Multiple	The channel 1 AXI-streaming master interface. Data from the DDR will stream out this interface using flow control.
StrToDDR0	Multiple	The channel 0 AXI-streaming slave interface. Data will stream from this interface into DDR using flow control.
StrToDDR1	Multiple	The channel 1 AXI-streaming slave interface. Data will stream from this interface into DDR using flow control.

Block Diagram



Ctrl Interface Address Map

It is anticipated that an instrument driver API will be used for controlling the Stream32x2 block. Hence low level register access to this block should not be needed.

The Stream32x2 block consists of two copies of the Xilinx AXI DMA v7.1 block used in the Direct Register Mode, a page register, and AXI interconnects. More information on the Xilinx AXI DMA IP block can be found in the Vivado pg021 AXI DMA v7.1 LogiCORE IP Product Guide.

The address space size of the DDR interface is considerably larger than the address space size available from the Host interface. In order to access the full memory space of the DDR memory, a page register is used to provide the MSBs of the DDR address (the LSBs of the address are provided by the address provided by the Host interface). Since the host interface uses 17 address bits, only 2^{17} bytes or 128 kB can be accessed without changing the page register. Bits 14:0 of the page register provides bits 31:17 of the DDR address.

Block	Start Address (Byte Addressing)	Size (Bytes)	Description
DMA0	0	1024	Control Registers for DMA channel 0
DMA1	1024	1024	Control Registers for DMA channel 1
Page	2048	4	Page Register that provides the MSBs of the address when using the Host interface to access DDR (Write only)

Version	2052	4	Version register (Read only)
---------	------	---	------------------------------

Version register

The version register is used to identify the version and configuration of the Streamer32x2/Streamer32x2b IP block.

Bits	Description
7:0	Version of the Streamer32x2/Streamer32x2b IP block
15:8	Number of streaming channels
23:16	Streaming channel data width (bits)
30:24	Transfer length register size
31	0=Simple DMA, 1=Scatter/gather DMA

Example Program

The **DDRExample** contains a PathWave FPGA project and a C++ program that demonstrates how to use the Streamer32x2 IP block. The DDRExample is in the PathWave FPGA installation directory example folder, which is typically C:\Program Files\Keysight\PathWave FPGA 2020 Update 1.0\examples. Please see the DDRExample readme.md file for more information about this example.

DSP Library IP

Included in the PathWave FPGA IP Repository ([PathWave FPGA IP Repository](#)) is a library of signal processing blocks that can be used to create things such as Digital Down Converters (DDCs) or Digital Up Converters (DUCs). These blocks do functions such as frequency translation (mixing with an internally generated local oscillator) and sample rate changes (both decimation and interpolation). While all of these IP blocks are general purpose, some of them are optimized for use in the M3xxx series of boards.

Scope

The purpose of this document is to explain the operation of the signal processing blocks, the purpose of their ports and interfaces, and how to modify the blocks via parameters. It is not intended to explain the underlying signal processing theory of sample rate changes. It is assumed the user has an understanding of basic signal processing such as the concept of aliasing as well as an understanding of sample rate changes (decimation and interpolation).

Data Formats

These IP blocks operate on streaming data using the AXI-streaming bus interface as described in [Keysight Standard Interfaces](#). This data could be either arbitrarily long streams of data (e.g. from an ADC) or a finite block of data (e.g. data read from DDR memory). These blocks support variable data bit widths (controlled via parameters) with the default width being 16 bit data as used in the M3xxx series of modules.

Sometimes the data is "supersampled". This means that multiple samples are processed for every clock. This allows processing of data sample rates faster than the allowed clock rate of the FPGA. In the M3xxx series of modules, the streaming sandbox interfaces (e.g. the ADC data or the AWG data) is supersampled by 5. Thus on every clock, five 16 bit samples are transferred using a $5 \times 16 = 80$ bit wide data bus. Note that this wider bus does not appear as 5 separate ports. The data for all five samples are combined into one wider bus. This shows up as one TDATA bus that is 80 bits wide rather than five busses each being 16 bits wide. With supersampled data, the least significant samples (e.g. bits 15:0) represent samples earlier in time while the most significant samples represent samples later in time.

Many of these IP blocks operate on complex data. This means that each sample consists of a real part and an imaginary part. Thus for complex data using 16 bit samples, the entire complex sample uses 32 bits of data width. Both the real and imaginary parts of each complex sample

are sent on the same AXI-streaming bus in an interleaved fashion. The details of how supersampled and/or complex data is encoded in the data stream can be found in [Keysight Standard Interfaces](#). For each complex sample, the real part occupies the less significant word (e.g. bits 15:0) while the imaginary part represents the more significant word (e.g. bits 31:16).

For supersampled complex data the real and imaginary parts of a sample are kept adjacent in the bus. Thus for 5X supersampled complex data, if (R0, R1, R2, R3, R4, R5, R6, R7, ...) represents the real samples with R0 being earlier in time, and (I0, I1, I2, I3, I4, I5, I6, I7, ...) represents the imaginary samples, as shown (time increasing from left to right):

R0	R1	R2	R3	R4	R5	R6	R7	...
----	----	----	----	----	----	----	----	-----

I0	I1	I2	I3	I4	I5	I6	I7	...
----	----	----	----	----	----	----	----	-----

then TDATA for one bus transaction would look like {I4, R4, I3, R3, I2, R2, I1, R1, I0, R0} where R0 is the LSBs of TDATA and I4 is the MSBs of TDATA as shown:

TDATA(159:144)	I4(15:0)	I9(15:0)	...
TDATA(143:128)	R4(15:0)	R9(15:0)	...
TDATA(127:112)	I3(15:0)	I8(15:0)	...
TDATA(111:96)	R3(15:0)	R8(15:0)	...
TDATA(95:80)	I2(15:0)	I7(15:0)	...
TDATA(79:64)	R2(15:0)	R7(15:0)	...
TDATA(63:48)	I1(15:0)	I6(15:0)	...
TDATA(47:32)	R1(15:0)	R6(15:0)	...
TDATA(31:16)	I0(15:0)	I5(15:0)	...
TDATA(15:0)	R0(15:0)	R5(15:0)	...

These blocks support full AXI streaming flow control (forward flow control and backward flow control). TVALID is the forward flow control signal, sent from Master to Slave, indicating that the Master has valid data on TDATA. TREADY is the reverse flow control signal, optionally sent from the Slave to the Master, indicating that the Slave is ready to accept data (if TREADY is not used, then it is assumed that the slave can always accept data at any time). Data is transferred when both TREADY and TVALID are asserted. Please see the the AXI4Lite specification for more details.

In addition to the streaming interfaces, some IP blocks use the Vector interface for control information. This might be the frequency value for a local oscillator or the bandwidth information for an adjustable filter. This signals can be tied to constants or connected to a user controllable register.

Handling of TUSER and TLAST

These IP blocks support the optional AXI-streaming signals TUSER and TLAST in addition to the main data bus TDATA. The connection or use of TUSER or TLAST is not required. These signals may be ignored if they are not being used. The TLAST signal indicates the last sample in a data block. It is passed through the IP block unchanged along with the data. For decimators where multiple input samples correspond to each output sample, the TLAST of all those samples are OR'ed together to form the TLAST of the corresponding output sample.

TUSER bits can be used to associate some data with some particular sample. Some example uses include triggers and overload/overflow information. The TUSER bits follow the data through the IP block accounting for things like pipeline delays and filter group delay. This is the best mechanism for associating an output sample with a particular input sample.

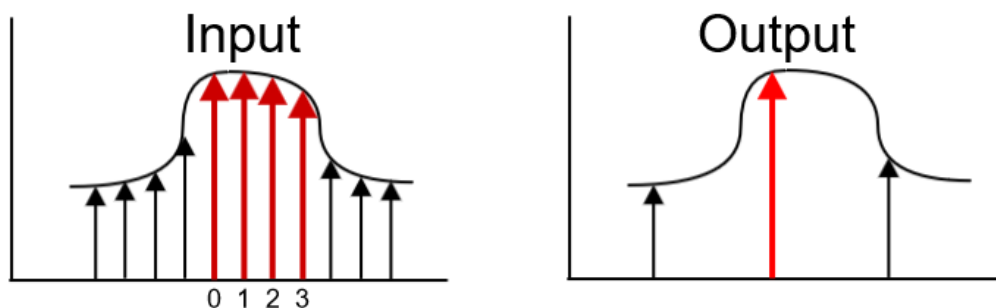
The number of TUSER bits per data sample can be changed from the default one via a parameter. Typically TUSER[0] is used to mark or tag a sample with trigger or timestamp information. For the decimation blocks, TUSER[0] is used internally, as well being passed through, to latch the state of the decimation counter when TUSER[0]=1. This latched information can be used to determine for which input sample TUSER[0] was asserted.

For blocks that include filtering, such as the decimators and interpolators, the TUSER bits are delayed to correspond to the group delay of the filter. For example, if the input stream was a single impulse, and if the TUSER input was asserted for this sample, then the TUSER output will be asserted at the midpoint (peak) of the output impulse response. TUSER[0] is the only TUSER bit used internally. Any other TUSER bits are merely passed to the output. Note that the TLAST bit is not delayed to account for group delay. Thus if TUSER and TLAST are asserted for the same input sample, they will occur at different output samples. For decimators where multiple input samples correspond to each output sample, the TUSER vector of all those samples are bitwise OR'ed together to form the TUSER of the corresponding output sample.

Note that some blocks may require the use of these optional signals. For example, the Streamer32x2 and Streamer32x2b IP blocks require the TLAST signal be asserted on the last sample of a DMA transfer (unless they are configured to internally generate the required TLAST signal). Please see [Streamer32x2 IP documentation](#) for more details.

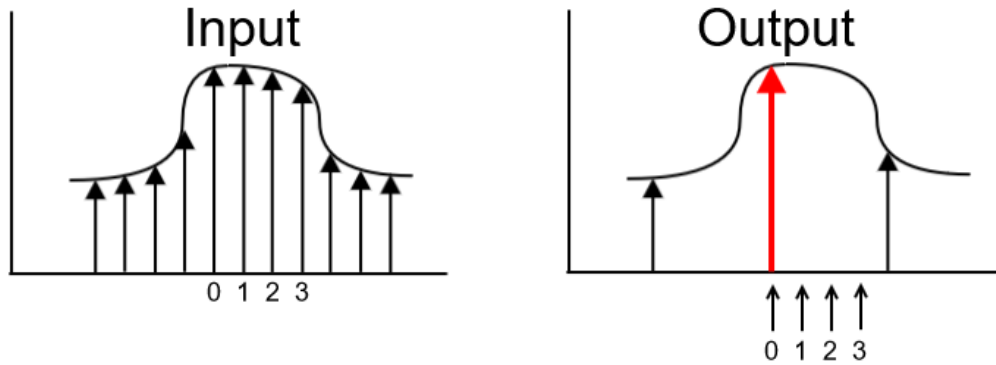
Decimation Trigger Corrections

The TUSER[0] signal can be used as a trigger signal to associate an output sample with a particular input sample as noted in the previous section. However, in the case of decimation filters, where the input sample rate is N times the output sample rate, there is some inherent ambiguity in the timing of this TUSER signal. Since the output sample rate is lower than the input sample rate, asserting the input TUSER for any of N different input samples would result in the output TUSER being asserted on the same output sample. The input to the decimation filter has a time resolution of the input sample rate whereas the output only has a time resolution of the output sample rate which is N times worse. If the output sample rate trigger resolution is sufficient for one's application, nothing further needs to be done. However, it is possible to increase the trigger resolution to the input sample rate by means of the DelayOut value. As the TUSER signal propagates through the decimation filter, the state of each decimation is recorded. After the output TUSER signal has been asserted, the DelayOut value reflects the state of each decimation.



For example, if the filter is decimating by a factor of four there is only one output sample for every four input samples. A trigger for any of the four red input samples would result in the same red output sample being marked. The DelayOut out value indicates which of these four actually caused the trigger event. A DelayOut value of 0 means the first red sample caused the trigger event. A value of 1 means the next red sample caused the trigger event, etc.

There is another, equivalent way to consider trigger corrections. When corrected for the filter's group delay, the time of an input trigger event corresponds to a particular output time. Due to the decimations in the filter, this output time may fall upon one of the output samples, or it may fall upon one of the samples that has been decimated away. The output TUSER signal indicates the latest sample on or before the ideal output trigger time. The DelayOut value reflects the time from the marked output sample to when the ideal trigger time would have been, as a fraction of the output sample period.



In this example, if the input sample labeled "0" had TUSER asserted, then the ideal output trigger time would be the time marked "0". The red sample would have TUSER asserted, and DelayOut would be zero. If the sample labeled "1" had TUSER asserted, then the ideal output trigger time would be the time marked "1". Note that this does not correspond to any output samples. Instead, the red sample would have TUSER asserted, and DelayOut would be one. Likewise for times two and three. The delay from the marked output sample to the ideal trigger time is $\text{DelayOut}/N$ where N is the decimation ratio.

Decimation Trigger Corrections for DecimateBy5 Blocks

The DelayOut for the DecimateBy5 blocks operates in the same way though with a slight modification. The DecimateBy5 blocks take as input five supersampled samples per clock. After a trigger event, the DelayOut indicates which of the 5 supersampled values caused the trigger. DelayOut = 0 means that filter_in_tuser[0] caused the trigger. DelayOut = 1 means that filter_in_tuser[User Data Width] caused the trigger. In general, the trigger was caused by filter_in_tuser[(DelayOut)*(User Data Width)].

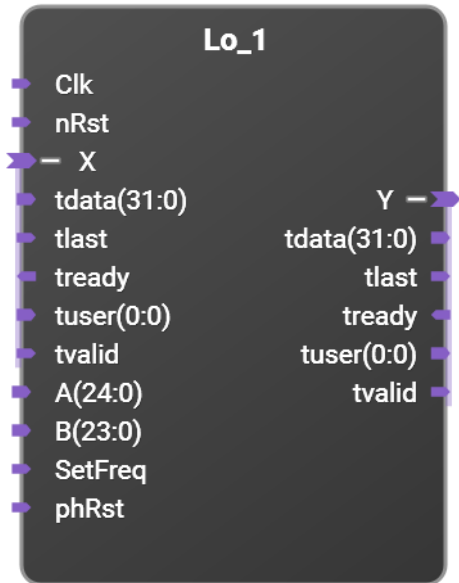
To find the ideal output trigger time, an offset needs to be subtracted from the DelayOut value. The ideal trigger time is $(\text{DelayOut} - 1.5)/5$ output samples after the output trigger. Note that this value may be negative, in which case the ideal trigger time is just before the marked output sample.

Detail IP Block Descriptions

Local Oscillator

This is a general purpose local oscillator/mixer block configured through the use of parameters. By default it generates a local oscillator signal and mixes (multiplies) this LO signal with the X input to generate the Y output. Optionally the block can output just the local oscillator signal without mixing it with the input.

It supports both supersampled and non-supersampled data. Both the input and output data can be independently selected as real or complex data. The Lo can be chosen to mix up (multiply by $e^{j\omega t}$) or mix down (multiply by $e^{-j\omega t}$).



This block supports supersampled data. In this description let S be the supersample factor. For non-supersampled data, set S to 1. The bit width of each data sample can be changed via the "Tdata size" parameter. Note that this parameter denotes the width of each individual sample, not the supersampled data width. The width of a real tdata port will be S times the Tdata size parameter while the width of complex tdata port will be $2S$ times the Tdata size parameter.

When the input port is set to real data, the imaginary part is assumed to be zero. When the output port is set to real data, only the real part is calculated - the imaginary part is discarded.

Optionally the output mixer stage can be bypassed. In this case the LO outputs the full scale local oscillator signal and ignores the input X interface. If Shift Direction is 0, then Y is $e^{j\omega t}$ so that the real part of Y is $\cos(\omega t)$ and the imaginary part (if complex output is selected) is $\sin(\omega t)$. If Shift Direction is 1, then Y is $e^{-j\omega t}$ so that the real part of Y is $\cos(\omega t)$ and the imaginary part (if complex output is selected) is $-\sin(\omega t)$. The output is scaled to full scale based on Tdata size. The resolution of the underlying trig lookup tables are fixed. The output data is merely shifted to present a full scale output depending on Tdata size. When Trig Only is selected the X interface (including the X_valid signal) is ignored. Y_tready will still control how fast output samples are generated.

By default, there are S TUSER bits, one bit per sample. The number of TUSER bits per sample can be changed via the "Tuser size" parameter. The TUSER and TLAST signals are not used inside these blocks - they are just passed from input to output with the data.

A and B control the local oscillator's frequency. Let T be the smallest power of 2 greater than or equal to S (so that $S \leq T < 2S$). The LO frequency is given by $f = f_s * (A+B/5^{10})/((S/T)*2^{25})$. Frequencies can be positive or negative. Valid input ranges for A and B are such that $-1/2 \leq f/f_s \leq 1/2$. Values of A and B that are outside this range will give incorrect results. Note that f_s is the sample rate of the data, not the clock rate of the FPGA which is $1/S$ of the sample rate. The LO is designed so that with a sample rate f_s of 1 Gs/s, the LO can produce LO frequencies with a decimal frequency resolution of 0.1 Hz or better. That is to say, any frequency that is a multiple of 0.1 Hz can be produced without frequency error. The internal frequency value of the LO block is updated when SetFreq is asserted. This allows A and B to be changed at different times and still have the LO cleanly change frequencies. It can also be used to change the frequency of multiple LOs synchronously if all the SetFreq signals are asserted at the same time. If this feature isn't required, SetFreq can be tied high and the LO will change frequency whenever A or B changes.

When asserted, phRst will reset the phase of the phase accumulator to zero and flush data in the LO's pipelines without resetting the programmed frequency. For phase continuous frequency changes, leave phRst negated. Note that in this case due to pipeline stages, the results of the frequency change will not be visible at the output for several samples. To

eliminate this delay in seeing the effects of frequency changes, assert `phRst` on or after the new frequency is set. This may easily be done by driving `phRst` with the same signal as `setFreq`. Note that `phRst` should not be tied high as that will prevent operation of the LO.

If the input is complex, sufficiently large values of the real and imaginary parts of `X` can result in a magnitude of the complex `X` being larger than the full scale input value (for example if both the real and imaginary parts of `X` are `+full_scale`, then the magnitude of `X` would be $\sqrt{2}$ times `full_scale`). In this case, the calculated output may not fit within the full scale output range. If this happens, the output will be clamped to \pm full scale. Note: this will cause distortion so it is recommended that the magnitude of the complex input be kept less than full scale.

When calculating the phase values (used to calculate the local oscillator value for each sample) typically dithering is used. Dithering adds a pseudorandom value smaller than 1 LSB to each phase value prior to the phase-to-amplitude lookup. This can convert potential spurious errors into more noise like errors. Note that this means that even when the LO's period is an integral number of samples, the waveform may not exactly repeat period to period. If this is not desired (e.g. for repeatable simulation results), it can be disabled by setting the Dither parameter to 0.

Starting at version 1.1 a new parameter *Pipeline* has been added. When checked (set to 1) this adds a pipeline stage to the phase accumulator and several pipeline stages to the sine/cosine calculation. This improves the speed of the LO block and can facilitate meeting timing in higher performance designs. These pipeline stages do not change the LO calculation - for the same input one would get the same output. It does increase the time after a reset before the LO will start outputting data by four clock cycles. It also slows down the effect of changing LO frequency by the same amount.

Parameters

Tdata size: Sets the data width for each sample (real data) or for each of the real and imaginary parts of each sample (complex data).

Tuser size: Sets the number of tuser bits per sample.

Complex Input: If set, then the input data is complex. If cleared, then the input data is real only.

Complex Output: If set, then the output data is complex. If cleared, then only the real part of the output data is generated.

Supersample: This sets the supersample value and determines how many parallel samples are processed at the same time.

Shift Direction: If Shift Direction = 0, the input is multiplied by $e^{j\omega t}$ (shift frequencies up).
If Shift Direction = 1, the input is multiplied by $e^{-j\omega t}$ (shift frequencies down).

Dither: Enables phase dithering to help convert spurious signals into more noise like signals (default is Dither=1 or enabled).

Trig Only: Enables output of the LO's trig functions and bypasses the mixer. If enabled, the `X` interface is ignored.

Pipeline: (Version 1.1 and later) When checked, this adds an extra pipeline stages in the phase accumulator and sine/cosine calculation to facilitate timing (default is Pipeline=1 or enabled).

Behavior of `phRst`

The calculation of local oscillator signal (phase accumulator and phase-to-amplitude converter) includes several stages of pipelining. This pipeline is normally kept full. One effect of this is that if the LO's frequency is changed, the results of that change is not seen by the data stream until several data samples have been processed. In this case, it is possible to update or change the LO frequency in a phase continuous manner. This means that the frequency changes without discontinuities in the LO waveform.

Sometimes the delay between setting the frequency and having the frequency change seen in the data stream is not desired. One example is block mode processing where a block of data is read from memory, and run through the LO. In this case, it is desired to have the programmed LO frequency available immediately. Otherwise the first few output points would be indeterminate based on the contents of the LO pipeline. To prevent the initial output values

using old pipeline data, the LO pipeline can be flushed by asserting phRst. This will clear out old pipeline data so that the first output sample would reflect new frequency values. To do this, assert phRst on or after the new frequency is set (using SetFreq) and before data is streamed through the LO. One way to do this is to connect the phRst to the same signal as SetFreq (note: if so, then these signals can't be tied high permanently else the LO would be held in reset). If phRst is used to flush the LO pipeline, that will result in non-phase continuous behavior. That is, frequency changes (and the flushing of the LO pipeline) will result in the LO waveform being discontinuous at the frequency change.

In continuous real time processing, such as when used with ADCs or DACs, phase continuous frequency changing is probably desirable.

In block mode processing, non-continuous frequency changing (without the pipeline delay before the new frequency is seen) is probably desirable.

1.1.1.1.1.1.1.1.1 Frequency Programming

The frequency of the LO is controlled by two inputs, A and B. A can be thought of as the coarse frequency setting and B as the fine frequency setting. A is a signed 25 bit number and B is a 24 bit number. The LO frequency is given by $f = f_s * (A+B/5^{10})/((S/T)*2^{25})$ where S is the supersample value and T is defined above. The following pseudocode can be used to calculate A and B. It is up to the user to ensure this math is done with sufficient precision for their application.

```
Let  $x = (f/f_s) * ((S/T) * 2^{25})$ 
Let  $A = \text{int}(x)$ 
Let  $B = \text{int}((x-A) * 5^{10})$ 
```

1.1.1.1.1.1.1.1.2 Example 1

Consider using the LO in a M3202A AWG. The clock rate in the FPGA is 200 MHz and data is 5X supersampled so the sample rate, f_s , is 1 Gsps. In this case S=5, and T would be 8 (next power of two greater than or equal to 5). So $f = (1 \text{ GHz}) * (A+B/5^{10})/((5/8)*2^{25})$. Suppose the desired LO frequency, f , was 123,456,789.1 Hz. Then:

$$x = (f/f_s) * ((S/T) * 2^{25}) = (123456789.1/10^9) * ((5/8) * 2^{25}) = 2589076.5217464320$$

$$A = \text{int}(x) = 2589076$$

$$B = \text{int}((x-A) * 5^{10}) = 5095180$$

and f exactly matches the desired frequency.

1.1.1.1.1.1.1.1.3 Example 2

Consider an LO processing data from an M8131A digitizer. In this example, the data is 16X supersampled with an FPGA clock of 400 MHz so the sample rate, f_s , is 6.4 Gsps. In this case S=16, and T would also be 16 (next power of two greater than or equal to 5). So $f = (6.4 \text{ GHz}) * (A+B/5^{10})/((2^{25})$. Suppose the desired LO frequency, f , was 1,234,567,891.2 Hz. Then:

$$x = (f/f_s) * ((S/T) * 2^{25}) = (1234567891.2/(6.4*10^9)) * (2^{25}) = 647269.13054146560$$

$$A = \text{int}(x) = 647269$$

$$B = \text{int}((x-A) * 5^{10}) = 2982565$$

and f exactly matches the desired frequency.

Porting Legacy Designs

To port designs using the legacy Lo5_dc and Lo5_uc block to use the new Lo block, use the following parameter values:

For **Lo5_dc**:

- Supersample = 5
- Complex Input = 0
- Complex Output = 1
- Shift Direction = 1

For **Lo5_uc**:

- Supersample = 5

- Complex Input = 1
- Complex Output = 0
- Shift Direction = 0

Local Oscillator (Legacy - not recommended for new designs)

Note: due to pipeline latency in the calculation of local oscillator waveform, these blocks have uncertain behavior for several samples following a reset. It is recommended that new designs use the Lo block described above which is more flexible and does not have these start up issues.

The DSP library contains two local oscillator blocks, Lo5_dc which is designed for down converter applications, and Lo5_uc which is designed for up converter applications. The difference between these is that Lo5_dc has real input data and complex output data while Lo5_uc has complex input data and real output data.



These blocks operate on 5X supersampled data, thus they process five samples in parallel. The bit width of each data sample can be changed via the "Tdata size" parameter. Note that this parameter denotes the width of each individual sample, not the 5X supersampled data width. The width of the Lo5_dc X_tdata port will be 5 times the Tdata size parameter while the width of the Y_tdata port will be 10 times the Tdata size parameter (since the output is complex while the input is real, the output is twice as wide due to having both real and imaginary components for each sample).

By default, there are 5 TUSER bits, one bit per sample. The number of TUSER bits per sample can be changed via the "Tuser size" parameter. The TUSER and TLAST signals are not used inside these blocks - they are just passed from input to output with the data.

The two input vectors A and B determine the frequency of the local oscillator. If the sample rate is f_s , then the LO frequency is $f_s * (A+B/5^{10})/(5*2^{22})$. Note that f_s is the sample rate of the data, not the clock rate of the FPGA which is 1/5 of the sample rate. The LO is designed so that with a sample rate f_s of 1 Gs/s, the LO can produce LO frequencies with a decimal frequency resolution of 0.01 Hz. That is to say, any frequency that is a multiple of 0.01 Hz can be produced without frequency error. The internal frequency value of the LO block is updated when SetFreq is asserted. This allows A and B to be changed at different times and still have the LO cleanly change frequencies. It can also be used to change the frequency of multiple LOs synchronously if all the SetFreq signals are asserted at the same time. If this feature isn't required, SetFreq can be tied high and the LO will change frequency whenever A or B changes.

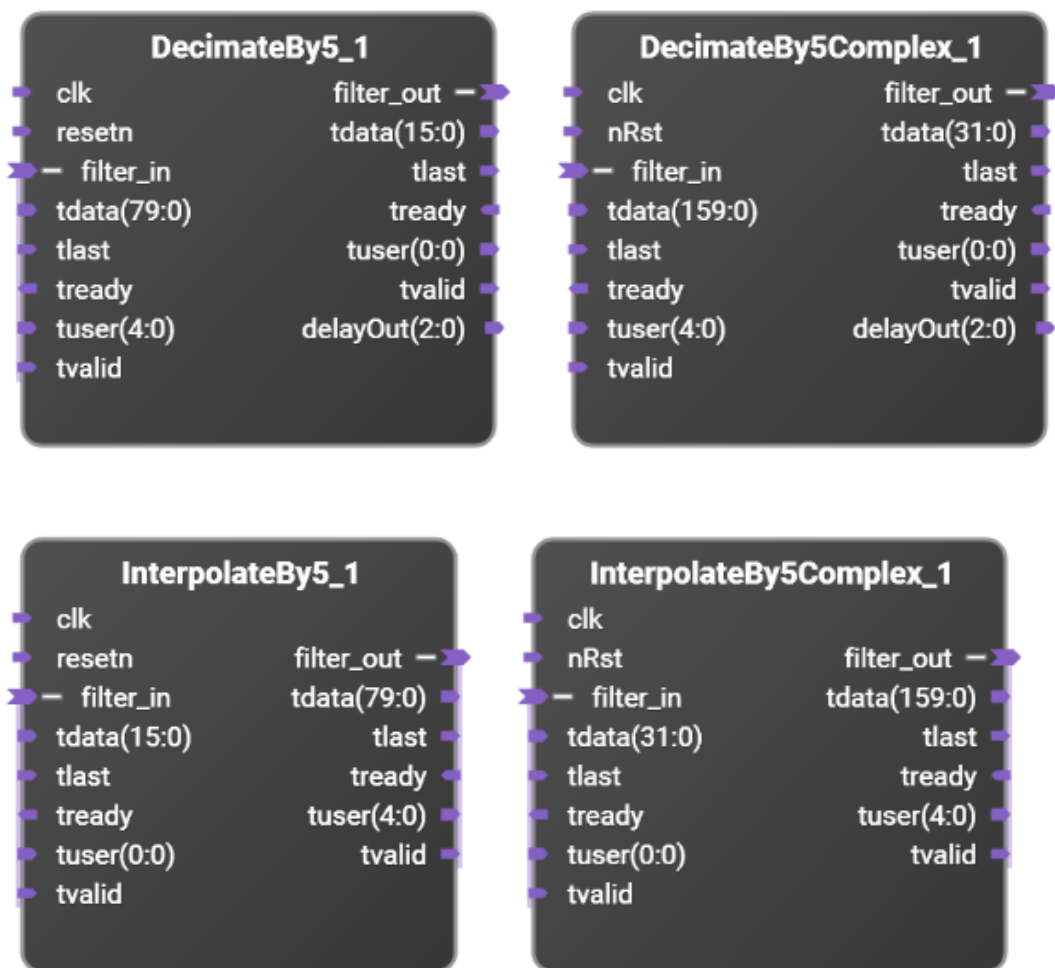
Lo5_dc will multiply the real input stream X with the complex local oscillator and generate the complex output stream Y. This block multiplies the real input by $e^{-j\omega t}$.

Lo5_uc will multiply the complex input stream X with the complex local oscillator and output the real part of the result as the real output stream Y. This block multiplies the complex input by $e^{j\omega t}$ and takes the real part for output. Since the input is complex, sufficiently large values of the real and imaginary parts of X can result in a magnitude of the complex X being larger than the full scale input value (for example if both the real and imaginary parts of X are +full_scale, then the magnitude of X would be $\sqrt{2}$ times full_scale). In this case, the calculated output may not fit within the full scale output range. If this happens, the output will be clamped to \pm full scale. Note: this will cause distortion so it is recommended that the magnitude of the complex input be kept less than full scale.

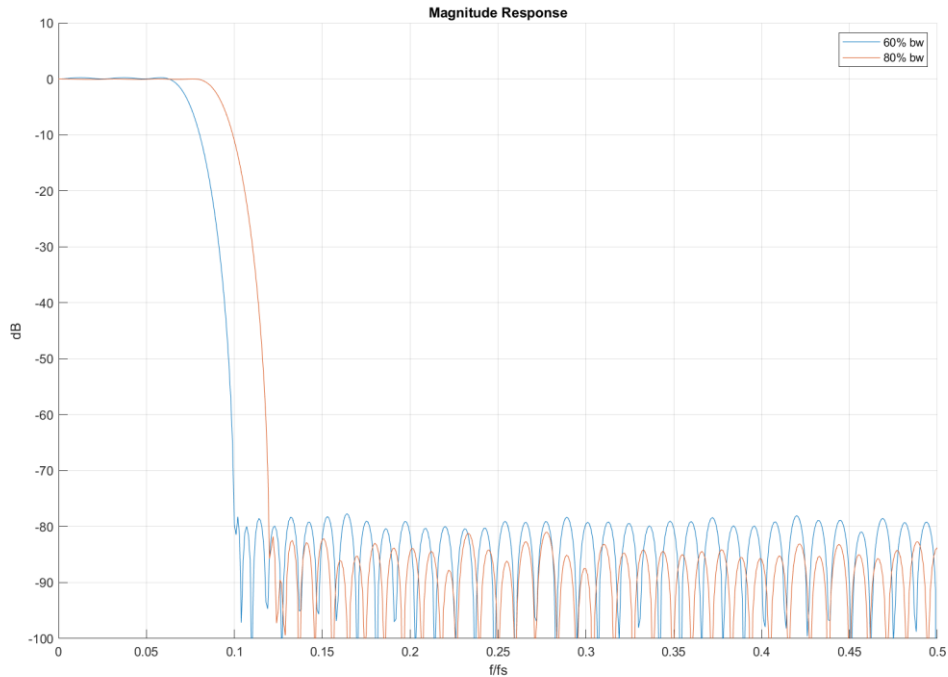
Note that for these blocks f/f_s is limited to the range ± 0.4 .

DecimateBy5/InterpolateBy5

There are both real and complex versions of the DecimateBy5 and InterpolateBy5 blocks. These blocks are used to convert between 5X supersampled data (5 samples per clock) and 1X supersampled data (1 sample per clock).



The DecimateBy5 block first low pass filters the input to protect against aliasing and then decimates by 5 (discarding 4 of every 5 output samples). The InterpolateBy5 block first interpolates by 5 by inserting 4 zero samples between each input sample and then low pass filtering to protect against aliasing. Both IP blocks can use one of two filters. One filter has a passband approximately 60% of Nyquist. This filter eliminates any aliasing at the expense of passband width. The other filter has a passband of 80% of Nyquist. It has an alias protected passband though there may be aliasing in the transition bands. These filters have the frequency responses:



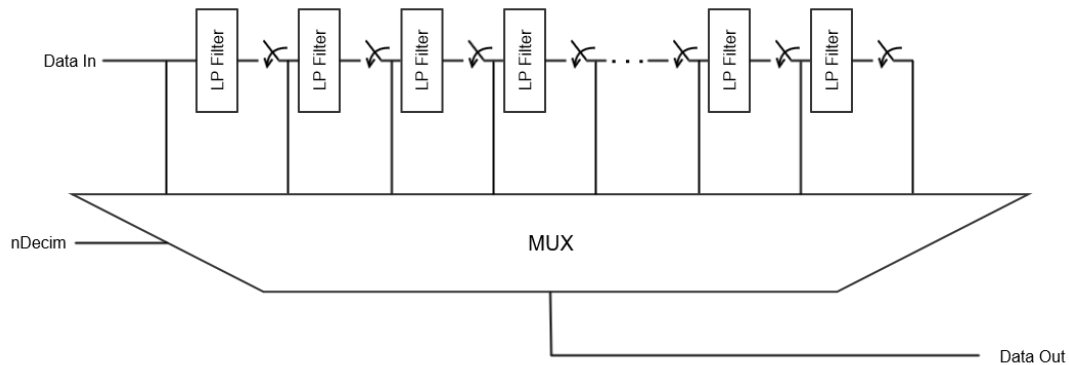
Note that the x-axis frequency is normalized to f_s , where f_s is the higher sample rate of the filter. That is the input sample rate for the decimator and the output sample rate for the interpolator. For the 60% filter, the passband extends up to $0.0625 f_s$ with the stopband starting at $0.12 f_s$. For the 80% filter, the passband extends up to $0.08 f_s$ with the stopband starting at $0.16 f_s$. For example, the M3102 digitizer has a sample rate of 500 Ms/s. Thus for the 60% filter, $f_s/2$ is 250 MHz and the passband is ± 31.25 MHz with the stopband above 50 MHz. For the 80% filter, the passband is ± 40 MHz with the stopband above 60 MHz. Note that these numbers are only for a sample rate of 500 Ms/s. For other sample rates, the passband and stopband frequencies would scale accordingly.

Power2Decimator/Power2Interpolator

These blocks operate on non-supersampled (a maximum of 1 sample per clock) data that can be either real or complex, and can decrease or increase the sample rate by 2^N where $N=0$ to 16. ($N=0$ is a bypass mode where the data is passed through the filter unchanged).

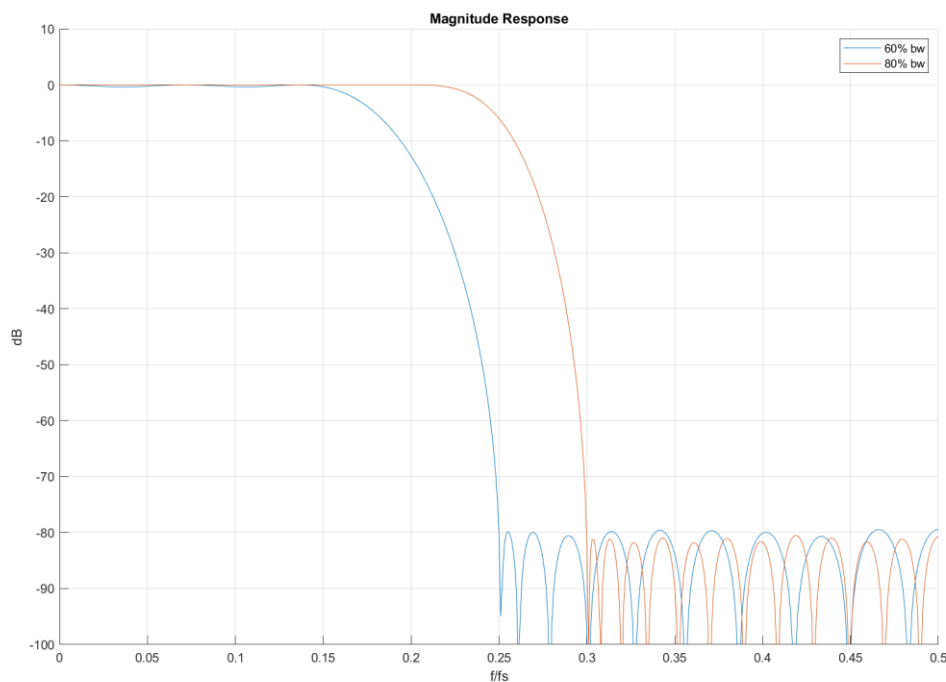


Conceptually, the Power2Decimator can be thought of as a set of 16 cascaded decimate by 2 stages (the internal design uses a more efficient architecture). Each stage first low pass filters its input and then decimates by two. A MUX controlled by $nDecim$ selects the output of one of these filters.

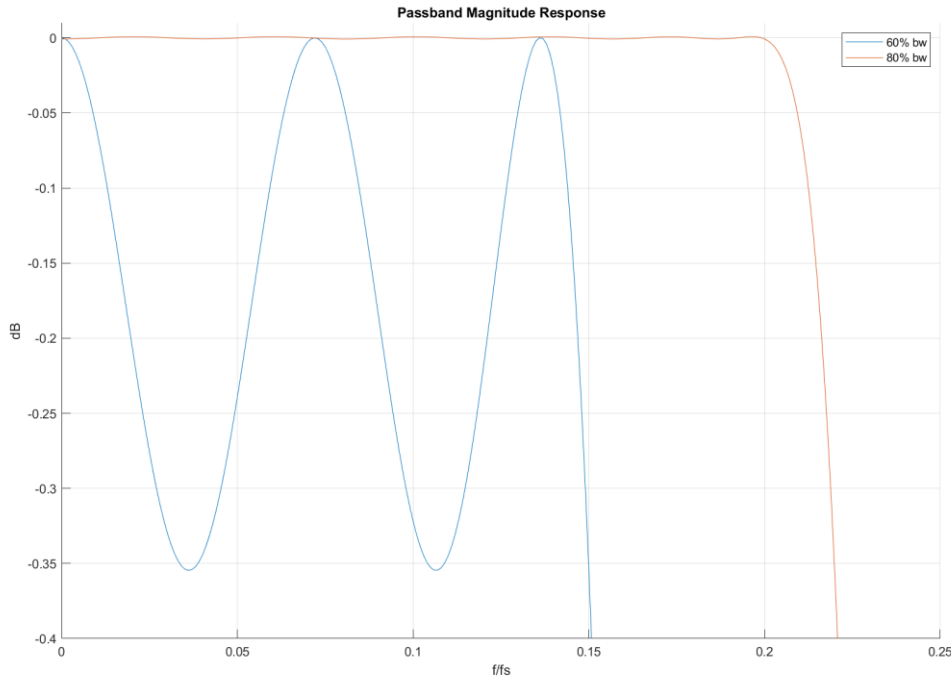


The Power2Interpolator does the reverse. It can be thought of as 16 cascaded interpolate by 2 stages. Each stage first interpolates by 2 by inserting a zero between each input sample and then low pass filters to eliminate aliased signals. In this case, $nInterp$ selects which stage receives the input data stream. All stages after that use the output of the previous stage.

Both the Power2Decimator and Power2Interpolator can use one of two filters. One filter has a passband approximately 60% of Nyquist. This filter eliminates any aliasing at the expense of passband width. The other filter has a passband of 80% of Nyquist. It has an alias protected passband though there may be aliasing in the transition bands. These filters have the frequency responses:



The 80% filter uses a halfband design and hence has an extremely flat passband. The 60% filter uses a plain FIR filter with more passband ripple:



Note that the x-axis frequency is normalized to f_s , where f_s is the higher sample rate of the filter. That is the input sample rate for the decimator and the output sample rate for the interpolator. For the 60% filter, the passband is $\pm 0.15 f_s$ which is 60% of the Nyquist rate, while the stopband starts at $0.25 f_s$. For the 80% filter, the passband is $\pm 0.2 f_s$, while the stopband starts at $0.3 f_s$. As an example, if the input sample rate to the Power2Decimator is 100 Ms/s, the bandwidth of the first stage of decimation would be ± 15 MHz sampled at 50 Ms/s for the 60% filter and ± 20 MHz sampled at 50 Ms/s for the 80% filter. For the 60% filters, the bandwidth of the second stage of decimation would be ± 7.5 MHz sampled at 25 Ms/s. The bandwidth of the third stage of decimation would be ± 3.75 MHz sampled at 12.5 Ms/s. For the 80% filters, the bandwidth of the second stage of decimation would be ± 10 MHz sampled at 25 Ms/s. The bandwidth of the third stage of decimation would be ± 5 MHz sampled at 12.5 Ms/s.

The bit width of each data sample as well as the width of the TUSER signal can be modified, if desired, via parameters. Note that the Tdata size parameter denotes the bit width of each component (real and imaginary) of each sample. Thus the width of the TDATA bus will be twice the value of this parameter for the complex version of these blocks. The Tuser size parameter denotes how many TUSER bits are associated with each (real or complex) sample.

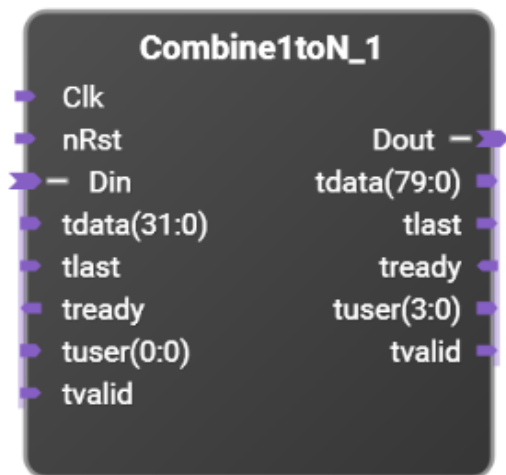
The TUSER and TLAST bits are passed through the decimation stages along with the data. Due to the filter response, there is no one output sample that corresponds to each input sample. A input consisting of an impulse will result in a broad output consisting of the impulse response of the filter. Thus tagging a particular input sample will result in an output sample being tagged that corresponds to the group delay of the filter which is close to the midpoint of the impulse response.

Since the output sample rate is less than the input sample rate (by a factor of 2^N), any of 2^N different input triggers would result in the same output trigger. The output port DelayOut can be used to determine which of these 2^N input samples caused the particular output trigger. As the trigger (TUSER[0]) signal propagates down the decimation stages, each decimate-by-two stage records the state of the decimation when the trigger passes. To interpret DelayOut, after a trigger has passed through the decimator, take the nDecim number of LSBs of DelayOut (i.e. AND DelayOut with $2^{nDecim}-1$), and this represents the number of input sample periods that needs to be added to the time of the marked input sample to get the time of the marked output sample.

Both the Power2Decimator and Power2Interpolator use 32 DSP blocks for the complex filter, and 16 DSP blocks for the real filter. This is the same regardless of which filter shape is chosen.

Combine1toN

Sometimes there is a need to combine multiple input samples into a wider output stream. One example of this would be to convert non-supersampled data (i.e. data at a rate of at most one sample per clock) into a supersampled output. The Combine1toN block will every N input samples into one output where N can be an integer or a half-integer (e.g. 2-1/2). This can be used to connect the non-supersampled output of the Power2Decimator to the supersampled Daq1 port. The IP block's "N" parameter is the integer part of this multiplier. To combine N+1/2 inputs into each output, select the "Add 1/2 to N" parameter.



To convert a real, non-supersampled 16 bit data sample to a 5X supersampled 80 bit data stream is straight forward. For every five 16 bit input samples, one 80 bit output is generated. Things are more complicated when dealing with complex data. In that case, the input is 32 bits wide (16 bits of real data, and 16 bits of imaginary data). To convert this to 80 bits wide, 2-1/2 input samples are collected for each output. So for an input of:

Din_tdata[31:16]	I0	I1	I2	I3	I4	...
Din_tdata[15:0]	R0	R1	R2	R3	R4	...

Then the output stream would look like:

Dout_tdata[79:64]	R2	I4	R7	...
Dout_tdata[63:48]	I1	R4	I6	...
Dout_tdata[47:32]	R1	I3	R6	...
Dout_tdata[31:16]	I0	R3	I5	...
Dout_tdata[15:0]	R0	I2	R5	...

To set up the Combine1toN block for this case, the parameter "N" should be "2", and the parameter "Add 1/2 to N" should be selected.

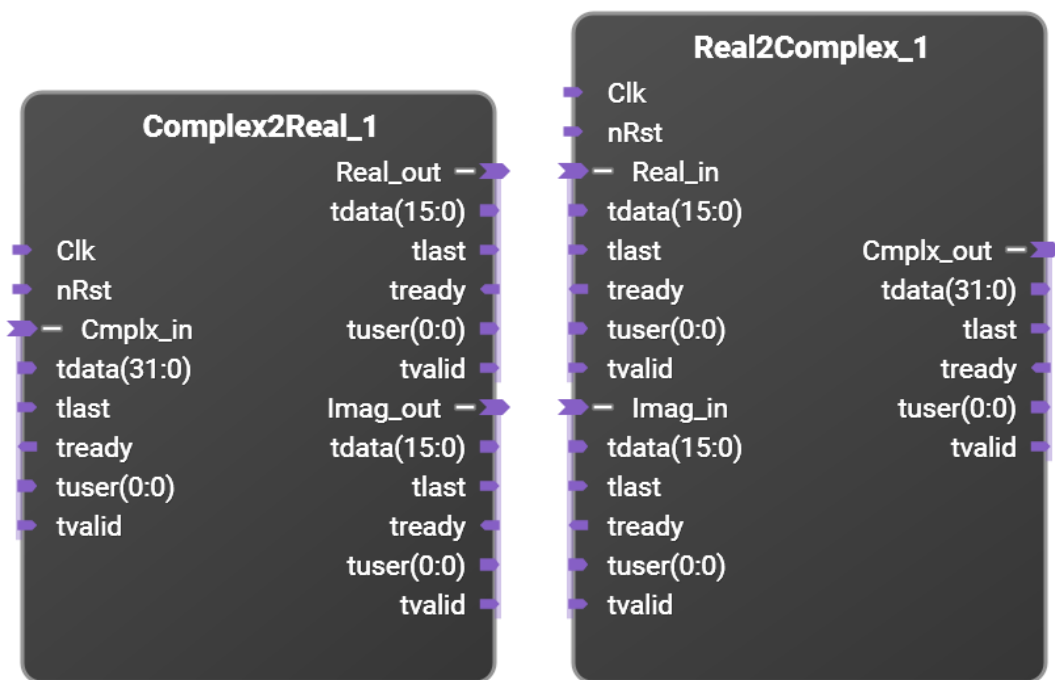
When the combination factor, N, is an integer, then the Dout_tdata is N times the size of Din_tdata, Dout_tuser is N times the size of Din_tuser. However, if the combination factor is N+1/2 the port sizing is more complicated (since ports can't be a half bit wide). Furthermore an extra bit is added to the Dout_tuser to indicate whether the half sample is at the LSBs or MSBs of the output. For combining N+1/2 samples, Dout_tdata is N+1/2 times the size of Din_tdata, Dout_tuser is N+1 times the size of Din_tuser + 1.

Logically in this case, R0 and I0 are parts of the same (complex) sample. Hence they share the same Din_tuser bit(s). However, some samples, such as R2/I2 are output in different bus cycles. The tuser bits for the R2/I2 input are output for both output bus cycles where R2 or I2 are output. So in this case the output would be (where Tn represents Din_tuser for sample n):

Dout_tuser[3]	0	1	0	...
Dout_tuser[2]	T2	T4	T7	...
Dout_tuser[1]	T1	T3	T6	...
Dout_tuser[0]	T0	T2	T5	...
Dout_tdata[79:64]	R2	I4	R7	...
Dout_tdata[63:48]	I1	R4	I6	...
Dout_tdata[47:32]	R1	I3	R6	...
Dout_tdata[31:16]	I0	R3	I5	...
Dout_tdata[15:0]	R0	I2	R5	...

Complex2Real / Real2Complex

These blocks convert between one complex stream of data and two independent streams (one for the real part, and one for the imaginary part) of data.



In order to know how to correctly interleave the complex data, these blocks need to know the size of the real data sample and any supersample value. The above pictures show a "Tdata size" of 16 and a "Supersample" of 1 (no supersampling). This means that the Real and Imaginary tdata busses are 16 bits wide, and the Cmplx tdata bus is twice this or 32 bits wide.

Supersampled Decimate/Interpolate By 2

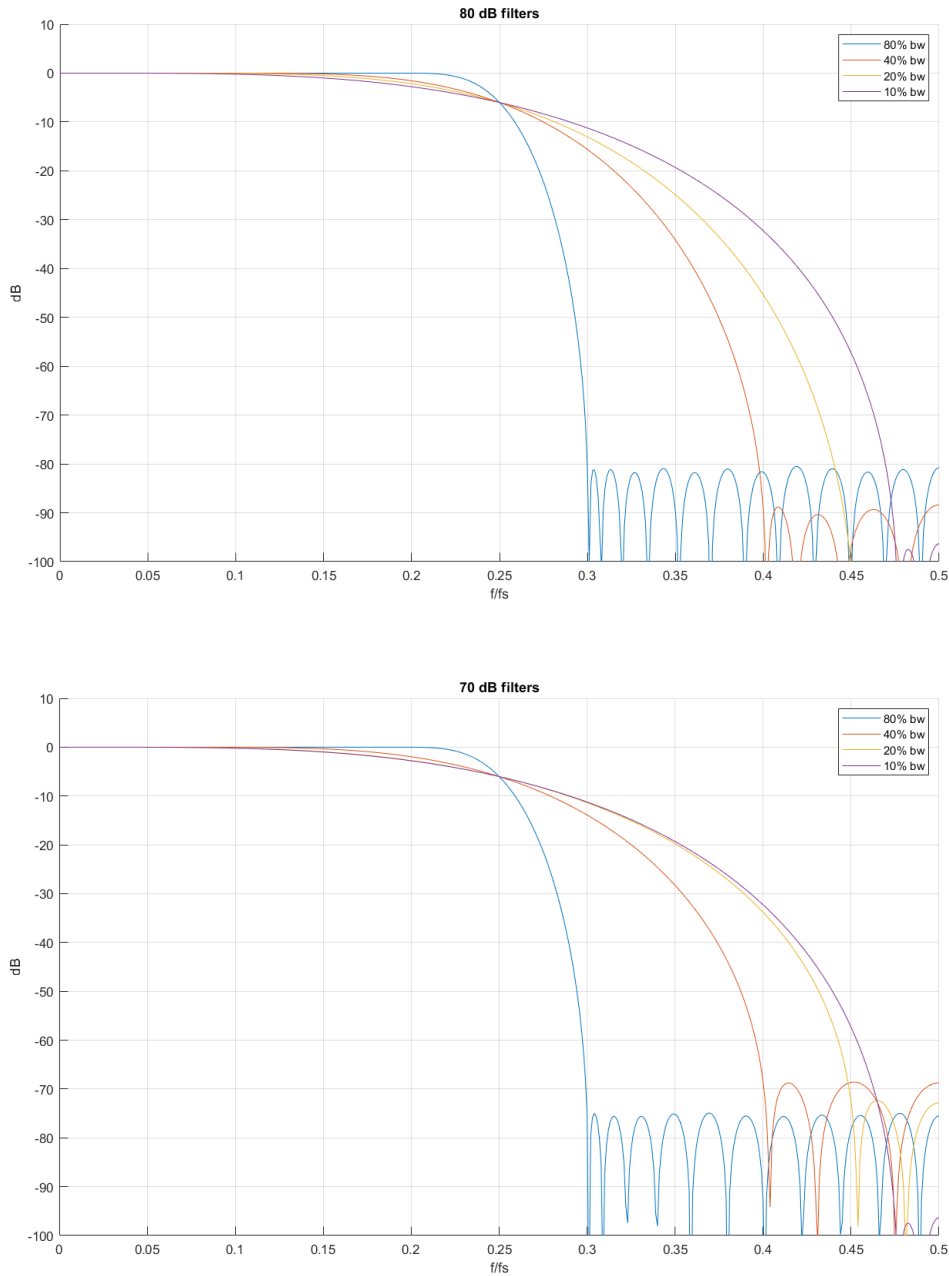
These blocks operate on supersampled data and can decimate or interpolate by two using real or complex data and offer choices of several filters (with different bandwidths and stopband rejection).



The decimator block takes in N samples/clock (where N has to be even), low pass filters the data, and outputs $N/2$ samples/clock. The interpolators take in N samples/clock, interpolate by two by adding a zero sample between each input sample, low pass filters the results, and outputs $2N$ samples/clock.

These blocks support several different filter options that can trade off performance for resource utilization. The stopband rejection can be selected as either 70dB or 80dB. The passband (and stopband) width can be selected as being either 10%, 20%, 40%, or 80% of Nyquist. When a number of these blocks are cascaded (e.g. to decimate by 4 or 8), then the higher sample rate filters can safely use smaller passbands which result in hardware and latency savings.

The following plots show the filter performance of the various filters plotted against normalized frequency. In these plots f_s represents the input sample rate of the decimator or the output sample rate of the interpolator.



The following table shows the filter sizes and resources used for the various filters. Note that these numbers are for each filter. To get the total number of DSP blocks used, this number needs to be multiplied by the output supersample value. The passband ripple of these filters (as is typical for halfband filters) is very good. For the 70dB rejection filters the passband ripple is $< 0.003\text{dB}$ while for the 80dB rejection filters the passband ripple is $< 0.001\text{dB}$.

Passband Width	Stopband Rejection	Filter Order	Number of Non-trivial Coefficients	DSP Per Filter
10%	70dB	6	2	2
20%	70dB	6	2	2
40%	70dB	10	3	4
80%	70dB	42	11	15

Passband Width	Stopband Rejection	Filter Order	Number of Non-trivial Coefficients	DSP Per Filter
10%	80dB	6	2	2
20%	80dB	10	3	4
40%	80dB	14	4	5
80%	80dB	46	12	16

The number of DSP block used per filter may be higher than the number of coefficients since for the larger filters some DSP blocks are used in the adder tree in addition to the coefficient multiplication. In the above table, the number of DSP blocks used is given per filter. In the decimator case, this is the number of supersampled outputs as each output has its own filter. In the interpolator case, this is the number of supersampled inputs (rather than outputs). While there are twice as many output samples as input samples, the nature of the halfband filters used is such that half the output sample filters only have trivial coefficients (0 or 1), thus only half the output samples use filters that require DSP blocks.

The DC gain through the interpolator and decimator can be set to either 1 or 1/2. The choice of which gain to use is application dependent. For an arbitrary input signal, the chance of the output exceeding the valid output range (and hence clipping) can be minimized by selecting a gain of 1/2. If the input signal is constrained, for example if it is the output of an earlier decimator/interpolator stage, then a gain of 1 may be more appropriate to maximize signal to noise.

The propagation of the TUSER signals is delayed to account for the group delay of the filter. For the decimator, the ideal location for the TUSER output may fall on one of the output samples or between two output samples depending on the timing of the input TUSER due to the decimation by two. The DelayOut signal differentiates these two cases. If the ideal output TUSER falls on an output sample, DelayOut will be zero. If the ideal output TUSER falls between two output samples, then the output sample just before the ideal location will have TUSER set, and DelayOut will be set to one. See the earlier section on Decimation Trigger Corrections for a more detailed discussion.

Both the ssDecim2 and the ssInterp2 blocks only support forward flow control. There are no TREADY signals. The ssInterp2fc block includes additional logic in order to support reverse flow control (TREADY) for situations where this is needed (e.g. prior to the reshapeM1 block).

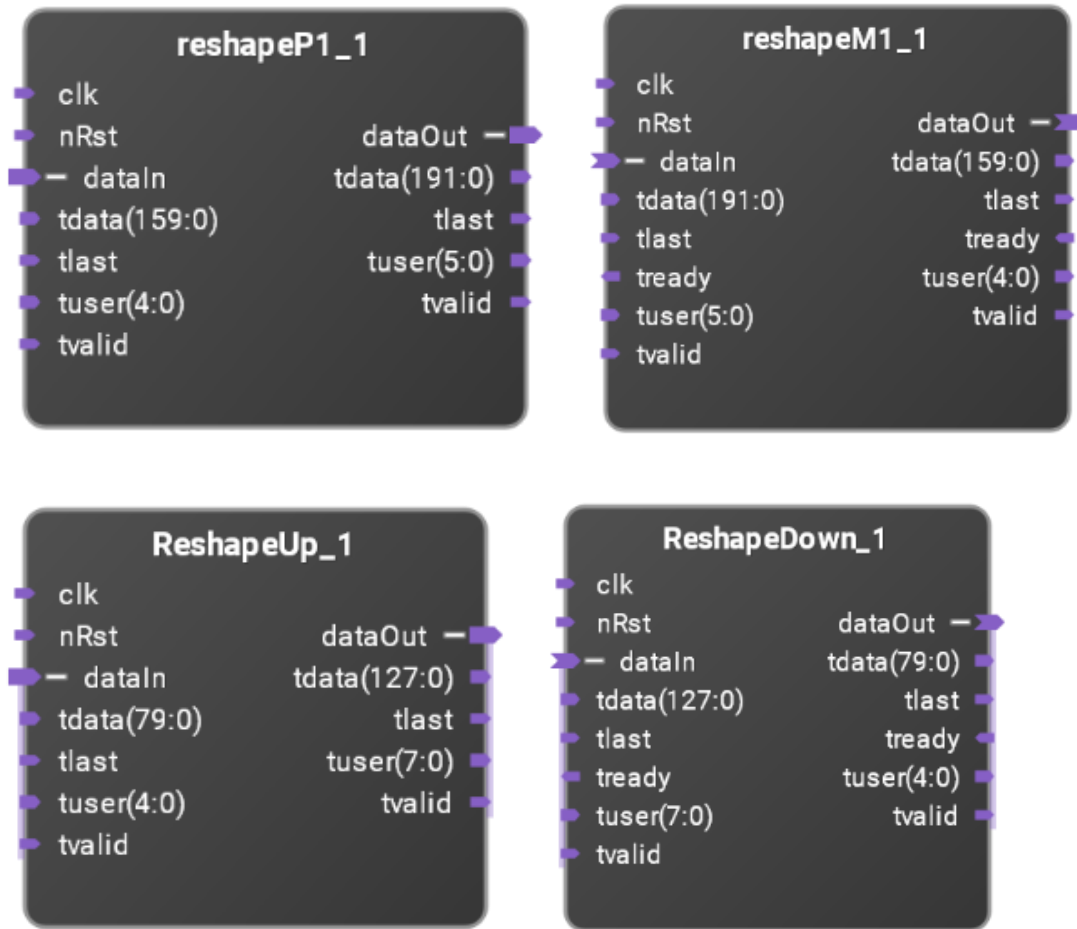
The ssDecim2 block requires the input supersample value to be even (since the output supersample value is half this and needs to be an integer). In order to support situations where the actual source has an odd supersample value, the reshapeP1 block may be used to convert the odd supersample signal to an even supersample signal.

The ssInterp2 blocks always output an even supersampled signal. To connect either of these to a destination requiring an odd supersample value, the reshapeM1 block may be used to convert from an even supersampled signal to an odd supersampled signal. Note: in this case the blocks driving the reshapeM1 must support reverse flow control.

In designs with a high supersample value, the reset signal can have a large fanout. To assist in meeting timing in such designs, the internal reset signal can be registered and duplicated to limit fanout. This incurs a one clock latency in the reset signal but may help in meeting timing in large designs.

Reshape

These blocks are used to convert between even and odd supersample signals or more generally between different supersample values. The reshapeP1 will convert from N to N+1 samples/clock while the reshapeM1 will convert from N to N-1 samples per clock. The reshapeUp will convert from N to M (M>N) samples/clock while the reshapeDown will convert from N to M (M<N) samples per clock.



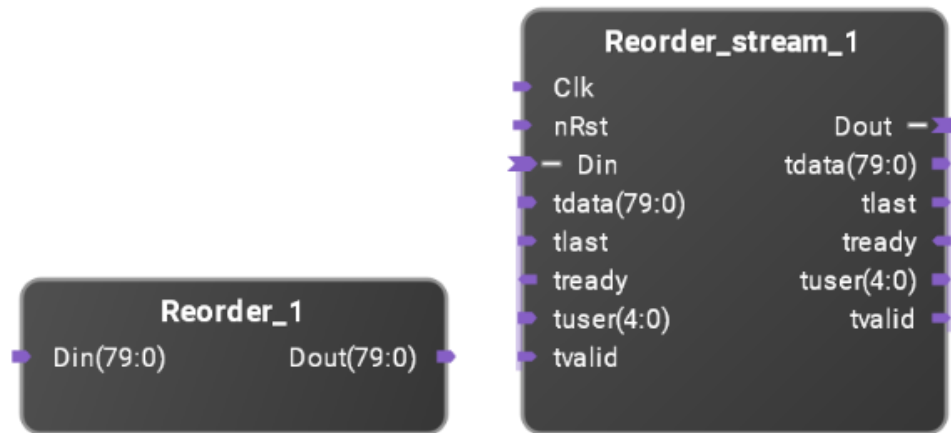
The reshapeP1 block is intended to be used in "push" data streams where data flow is controlled by the source and pushed from input to output. In particular it is intended to be used between instances of the ssDecim2 block when needed. It can accept data up to full rate, that is, a transaction every clock cycle. If the input supersample value is N , then the output supersample value is $N+1$, so the output data width is $(N+1)/N$ times the input width. Consequently the output transaction rate is $N/(N+1)$ times the input transaction rate. Therefore the reshapeP1 block will never generate output transactions every clock. The reshapeUp block is a generalization of the reshapeP1 block. In reshapeUp, the output supersample value can be any value greater than the input supersample value. In reshapeUp, the output supersample value is not limited to only one more than the input supersample value as it is in reshapeP1.

The reshapeM1 block is intended to be used in "pull" data streams where data flow is controlled by the destination and pulled. In particular it is intended to be used between instances of the ssInterp2fc block when needed. It can output data up to full rate, that is, a transaction every clock cycle. If the input supersample value is N , then the output supersample value is $N-1$, so the output data width is $(N-1)/N$ times the input width. Consequently the input transaction rate is $(N-1)/N$ times the output transaction rate. Therefore the reshapeM1 block will never accept input transactions every clock. This requires whatever is driving data to the reshapeM1 block to support reverse flow control with the TREADY signal. The reshapeDown block is a generalization of the reshapeM1 block. In reshapeDown, the output supersample value can be any value less than the input supersample value. In reshapeDown, the output supersample value is not limited to only one less than the input supersample value as it is in reshapeP1.

The TUSER signals are passed along with the data. The TLAST signal is handled differently. On the input interface, the TLAST signal is associated with the last sample of a transaction. This is the most significant sample of the supersampled input. When that sample is

output (not necessarily as the most significant sample of the supersampled output) the output TLAST signal will be asserted.

Reorder / Reorder_stream



These blocks will reorder the samples in supersampled data. These blocks can reserve the sample ordering within the supersampled word as well as optionally swap real/imaginary parts of each sample.

Reversing the samples would change {x5,x4, x3, x2, x1} to {x1,x2,x3,x4,x5}.

Reversing complex data would change {i3,r3,i2,r2,i1,r1} to {i1,r1,i2,r2,i3,r3}. Note that the complex samples are reordered, but the ordering of real/imag is preserved.

Reversing and swapping complex data would change {i3,r3,i2,r2,i1,r1} to {r1,i1,r2,i2,r3,i3}. Note that the complex samples are reordered, and the ordering of real/imag is swapped.

Just swapping complex data would change {i3,r3,i2,r2,i1,r1} to {r3,i3,r2,i2,r1,i1}. Note that the order of complex samples is preserved, and the ordering of real/imag is swapped.

Both of these blocks are purely re-routing. There is no logic or delay/latency added. The **Reorder_stream** block does not use the **Clk** or **nRst** ports. However, these ports are included for compliance with the AXI-streaming specification.

The **Supersample** parameter sets the number of samples per clock in the input and output streams.

The **Data Width** parameter sets the number of data bits in each sample. The **Din/Dout** sizes are **Supersample*Data Width** bits for real data, and **2*Supersample*Data Width** bits for complex data.

The **Reverse** parameter, when set, will reverse the order of samples in the supersampled word.

The **Complex Data** parameter determines if the data is complex or real only.

The **Swap** parameter, when set, will swap the real and imaginary parts of complex data. This is ignored for real data.

The **Register** parameter, when set, adds a register stage to the output, else the IP block is purely combinatorial (only on **ConvertBitWidth_stream**).

The **Tuser** size parameter sets the number of **tuser** bits per sample in the axi-streaming interface (only on **ConvertBitWidth_stream**).

Reordering Real Data

For real data (**Complex Data** = 0), the **Reorder** block can reserve the order of the data in the supersampled word. For **Supersample**=5:

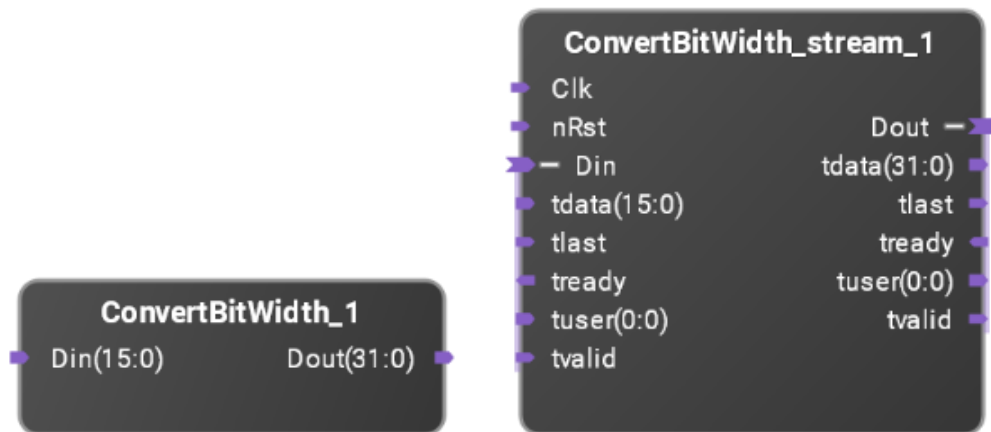
Din	Dout for Reverse=0	Dout for Reverse=1
X5, X4, X3, X2, X1	X5, X4, X3, X2, X1	X1, X2, X3, X4, X5

Reordering Complex Data

For complex data (Complex Data = 1), the Reorder block can reserve the order of the data in the supersampled word. It can also swap the real/imaginary parts of each sample. For Supersample=3:

Din	Dout for Reverse=0, Swap=0	Dout for Reverse=0, Swap=1	Dout for Reverse=1, Swap=0	Dout for Reverse=1, Swap=1
I3,R3, I2,R2, I1, R1	I3,R3, I2,R2, I1, R1 (This doesn't change anything)	R3,I3, R2,I2, R1,I1 (Keeps sample order, but swaps real/imag)	I1,R1, I2, R2, I3,R3 (Reverses sample order, but keeps real/imag ordering)	R1, I1, R2, I2, R3,I3 (Reverses sample order and swaps real/imag)

ConvertBitWidth / ConvertBitWidth_stream



Convert sample data between different bit widths with or without rounding/clamping.

The justification of the data needs to be specified. If the data is left justified, then the MSBs of the input and output samples will be the same (subject to rounding). This is typically the case when the data is a fixed point value representing values between +/-1. If the data is right justified, then the LSBs of the input and output will be the same (subject to clamping). This is typically the case when the data represents integers.

When the sample size increases (the output data width > input data width), then for left justified data the data is zero padded on the LSB side to extend the length, and for right justified data, the MSBs are either zero padded or sign extended, depending on whether the data is signed or not), on the MSB side to extend the length.

When the sample size decreases (the output data width < input data width), then there is the possibility of data loss. These IP blocks can do this safely by rounding and/or clamping as necessary. This adds some amount of logic latency but prevents data rollover and any truncation bias. If the user knows that the input data will fit in the output data size without problem, the IP can be set to not round or clamp. In this case the IP adds zero delay to the signal as it then becomes just wires. However in this case left justified data will be truncated possibly introducing a bias and right justified data may roll over.

The rounding algorithm used is convergent rounding. Numbers with a fractional part less than 1/2 will round down. Numbers with a fractional part greater than 1/2 will round up. Numbers

with a fractional part of exactly 1/2 will round towards the even integer. This rounding minimizes rounding noise while being unbiased for large enough signals.

The ConvertBitWidth IP block is purely combinatorial. If registering is needed for timing, then it should be added externally to this block. The ConvertBitWidth_stream IP block supports full flow controlled AXI-streaming interfaces. This block optionally always adds a register stage which can assist with timing closure. If this is not included, then the data path between the input and output is combinatorial. If the register stage is included, then this block will add latency.

The Supersample parameter sets the number of samples per clock in the input and output streams.

The Input Data Width parameter sets the number of data bits in each input sample. The Din size is Supersample*Input Data Width bits for real data, and 2*Supersample*Input Data Width bits for complex data.

The Output Data Width parameter sets the number of data bits in each output sample. The Dout size is Supersample*Output Data Width bits for real data, and 2*Supersample*Output Data Width bits for complex data.

The Justify parameter determines if the data is left justified (the MSBs are the more important) or right justified (the LSBs are the more important). The two choices are "Left (Keep MSBs)" and "Right (Keep LSBs)". This determines which bits are kept and which are discarded/padded.

The Round/Clamp parameter determines if the bit width conversion should use rounding/clamping or not. The two choices are "Round/Clamp" and "No Round/Clamp".

The Signed Data parameter determines if the data is signed or unsigned.

The Complex Data parameter determines if the data is complex or real only.

The Register parameter adds a register stage to the output when set, else the IP block is purely combinatorial (only on ConvertBitWidth_stream).

The Tuser size parameter sets the number of tuser bits per sample in the axi-streaming interface (only on ConvertBitWidth_stream).

Increasing the bit width

When the number of bits per sample is increased, then the data is either left-padded with zeros or signed extended or right-padded with zeroes depending on the data justification selected. The following table shows example conversions when going from 8 to 12 bits per sample:

Input Value	Left Justified Output	Right Justified Signed Output	Right Justified Unsigned Output	Notes
0x24	0x240	0x024	0x024	Positive values are just zero padded
0xA4	0xA40	0xFA4 ^A	0x0A4	^A The signed output is sign extended

Decreasing the bit width

When the number of bits per sample is decreased, then there is the possibility of data corruption if the data isn't rounded and clamped appropriately. This rounding/clamping can introduce latency. If the user knows the limits of the size of the data samples and knows that overflowing is not possible and can live with possible truncation bias, then rounding/clamping can be omitted resulting in no additional delays (since the IP block will just be wires in this case). The following table shows example conversions when going from 12 bits to 8 bits. This table also includes examples of bad output data if the input range doesn't fit in the output range:

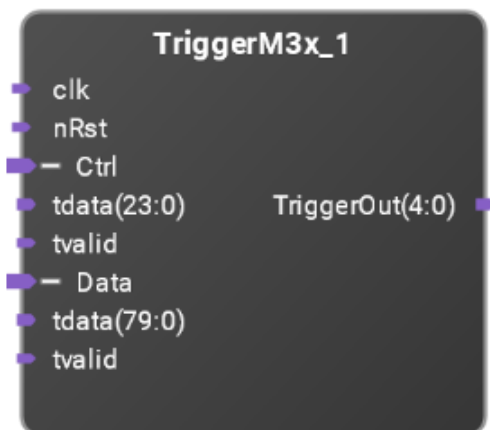
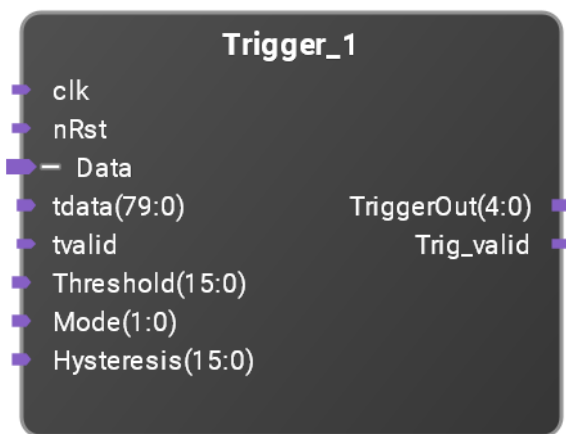
Input Value	Left Justified Output	Right Justified Output	Notes
0x050	0x05	0x50	This value converts exactly
0x05C	0x05 ^A	0x5C	^A This value got truncated
0x530	0x53	0x30 ^B	^B The input was too large for the output resulting in incorrect output data
0x0C5	0x0C ^C	0xC5 ^D	^C This value got truncated ^D This is okay for unsigned data, but incorrect for signed data

To protect right justified data, clamping can be used. While the output will still be distorted, it is often preferable to having the data wrap around so that a positive input value results in a negative output value. Left justified data incurs a 1/2 LSB bias when truncation is used. To protect against this, convergent rounding can be used. This also adds a bit of clamping. Normally a full scale input signal would round up, but the rounded value isn't representable in the output so the output needs to be clamped to the maximum output value.

Input Value	Left Justified Signed Output	Left Justified Unsigned Output	Right Justified Signed Output	Right Justified Unsigned Output	Notes
0x050	0x05	0x05	0x50	0x50	No rounding or clamping needed
0x053	0x05 ^A	0x05 ^A	0x53	0x53	^A This value got rounded down
0x05C	0x06 ^B	0x06 ^B	0x5C	0x5C	^B This value got rounded up
0x068	0x06 ^C	0x06 ^C	0x68	0x68	^C This value, with fractional part 1/2, rounded down to an even number
0x078	0x08 ^D	0x08 ^D	0x78	0x78	^D This value, with fractional part 1/2, rounded up to an even number
0x092	0x09	0x09	0x7F ^E	0x92 ^F	^E This value got clamped to positive full scale ^F This value didn't need clamping
0x7FF	0x7F ^G	0x80 ^H	0x7F ^I	0xFF ^I	^G This would normally round up but got clamped to positive full scale ^H This could round up and fit in the output size ^I This got clamped to positive full scale
0xFF8	0x00 ^J	0xFF ^K	0xF8	0xFF ^L	^J This value, with fractional part 1/2, rounded up to an even number ^K This would normally round up but got clamped to positive full scale ^L This got clamped to positive full scale

Input Value	Left Justified Signed Output	Left Justified Unsigned Output	Right Justified Signed Output	Right Justified Unsigned Output	Notes
0x812	0x81	0x81	0x80 ^M	0xFF ^N	^M This got clamped to negative full scale ^N This got clamped to positive full scale

Trigger / TriggerM3x



The Trigger block is an analog trigger detector that will assert a one clock wide trigger pulse for each detected trigger event. For supersampled data, there is one TriggerOut bit for each of the supersampled input values. This trigger supports hysteresis and depending on the Mode, can be off (Mode=0), rising edge (Mode=1), falling edge (Mode=2), or both edges (Mode=3). Internally, there are three threshold levels, an upper threshold equal to Threshold + Hysteresis, a middle threshold equal to Threshold, and a lower threshold equal to Threshold - Hysteresis. Note that the input data and Threshold can be configured as signed (the default) or unsigned values. For correct operation, Threshold +/- Hysteresis should not exceed full scale limits.

The hysteresis in this trigger block is used to prevent multiple triggers when the input signal is slowly varying and noisy.

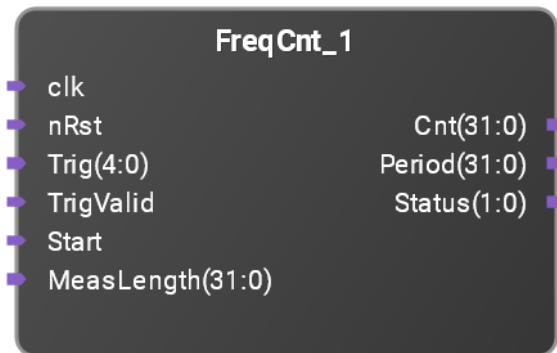
A rising edge trigger is detected only on the first sample above the middle threshold after the signal has gone below the lower threshold. Thus a rising edge trigger requires the input signal

to drop below the lower threshold and then rise above the middle threshold. Likewise, a falling edge trigger is detected only on the first sample below the middle threshold after the signal has gone above the upper threshold. Thus a falling edge trigger requires the input signal to rise above the upper threshold and then drop below the middle threshold. Note that multiple triggers will not be generated if the input crosses threshold multiple times but doesn't cross the upper/lower thresholds.

The latency through the Trigger block from Data/tdata to TriggerOut is 2 clock cycles. Trig_valid is Data/tdata delayed by two clocks. Trig_valid indicates when TriggerOut may be driven. When Trig_valid is negated, TriggerOut will always be driven to zero.

The TriggerM3x block is the same as the Trigger block except the ports have been modified to match the ports of the Analog Trigger block in the M3xxxx series of digitizers. This is to facilitate replacing the Analog Trigger block (that does not include hysteresis) with a trigger block that does include hysteresis. The Ctrl interface replaces the AnalogTrigger interface. With the TriggerM3x, the level of hysteresis is fixed at design time via a parameter rather than being a run time setting input port. To support both supersampled and non-supersampled M3xxxx digitizers, the TriggerM3x includes a Supersample parameter that should be set to 1 or 5 accordingly.

FreqCnt



FreqCnt is a frequency counter block. It measures an integral number of periods during the measurement interval and reports the number of trigger events (the number of signal periods) and the time between the first and last trigger event during the measurement interval. The input is the Trig port which consists of one bit per supersampled data. These trigger signals might come from the Trigger IP block. FreqCnt supports supersampled data.

MeasLength sets the length of the measurement interval in clock cycles (which is the sample rate divided by the supersample factor). Note that FreqCnt only counts clocks when TrigValid is asserted. For a given value of MeasLength, if the Trig values only come in every other clock, the elapsed time for the measurement will be twice as long as it would be if Trig values come in every clock.

Start begins a new measurement. Note that Start can be tied high in which case a new measurement will start immediately after the previous measurement. The Cnt and Period outputs are latched at the end of a measurement so they may be read while a new measurement is in progress. Depending on the SyncStart parameter, the measurement interval will either begin when Start is asserted (SyncStart = 0) or on the first trigger event after Start is asserted (SyncStart = 1).

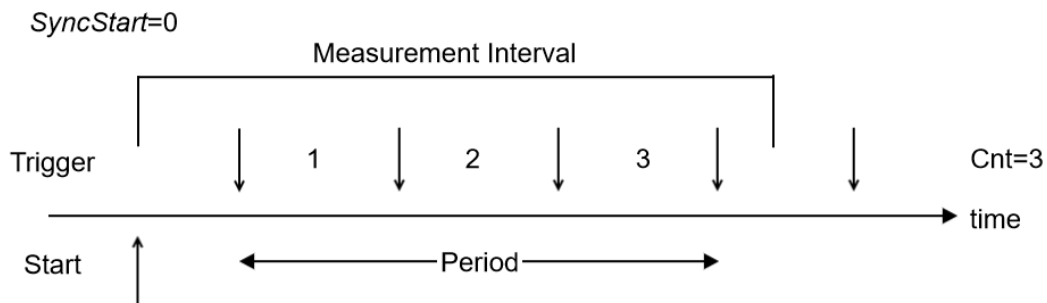
Cnt outputs the number of whole signal periods during the measurement interval. This is one less than the total number of trigger events (each indicated by a Trig bit being 1).

Period outputs the time between the first and last trigger event measured in samples. The frequency of the input signal can then be calculated $f = f_s * Cnt / Period$ where f_s is the sample rate.

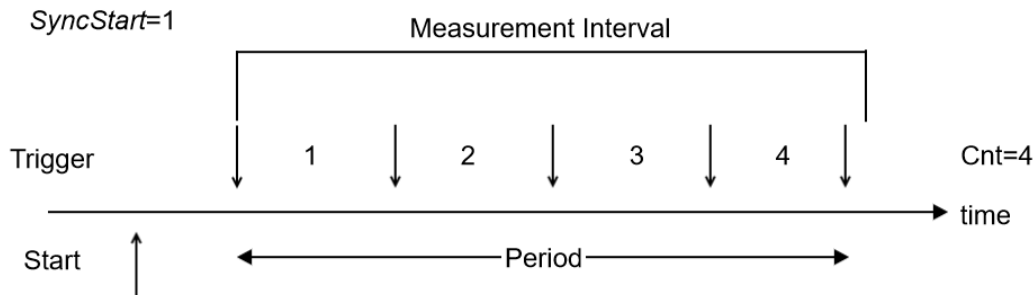
Status indicates the internal state of the measurement. Status[0] = 1 indicates that a measurement is in progress. Note that if Start is tied high, a measurement will almost always

be in progress since the next measurement will start immediately after the previous one finishes. Status[1] = 1 indicates that the the first trigger event has been observed. If *SyncStart* = 1 and Status[1:0] = 01, it means that the FreqCnt is awaiting the first trigger event which will then start the actual measurement interval.

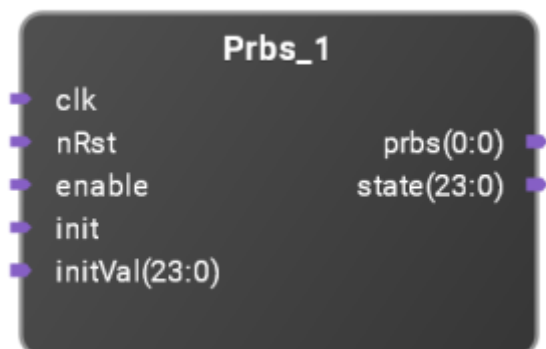
The following diagram shows the FreqCnt operation when *SyncStart* = 0. In this case, the measurement interval begins when Start is asserted. In this example, four trigger events occur within the measurement interval. This means there are three periods during the measurement interval, so Cnt=3. The Period is the number of samples between the first and last triggers during the measurement interval.



This diagram shows the same example as above except that *SyncStart* = 1. In this case, the measurement interval starts synchronously with the first trigger event. After Start has been asserted, the actual measurement interval does not start until the first trigger event. The time for the measurement interval is the same, it is just shifted later in time. In this example, the shift in measurement interval results in five trigger being observed rather than four. Thus there are four periods in the measurement interval, and Cnt = 4. Because more trigger events occurred during the measurement interval, the value of Period will be larger so that the calculated frequency, $f_s * Cnt / Period$, remains the same.



PRBS - Pseudo Random Bit Sequence generator



This block generates a Pseudo Random Bit Sequence using a Linear Feedback Shift Register (LFSR). A LFSR with an N-bit state vector can sequence through a pseudo random sequence consisting of at most 2^N-1 states. There is always one state value that isn't used. For the XOR structure, the all zero state is not used. For the XNOR structure, the all one state is not used. Sequences that go through all 2^N-1 states are known as Maximum Length Sequences. PRBS generators can be used to make pseudo random noise and whitening sequences.

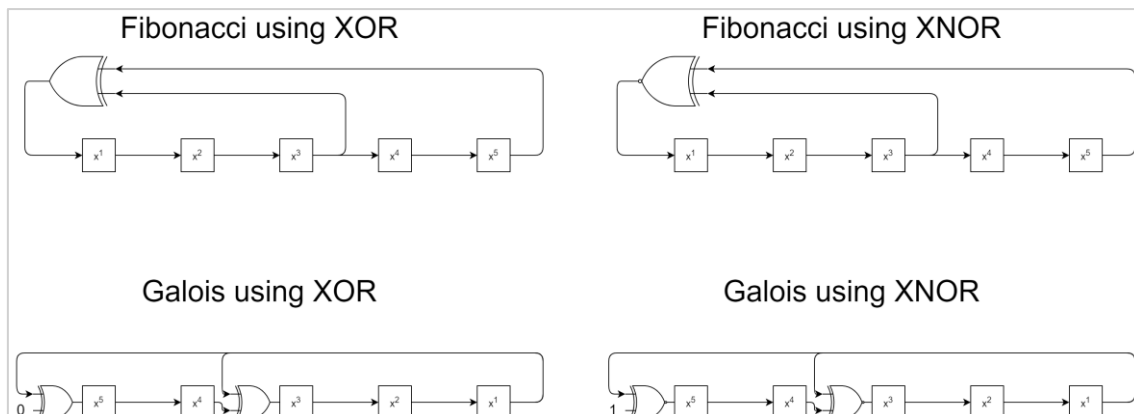
In a Fibonacci LFSR, the shift-in bit is determined by the XOR or XNOR of a subset of the state bits. The subset of state bits used to determine the shift-in values is known as the taps. The implementation consists of one wide XOR/XNOR gate where the width of the gate is determined by how many taps are used.

An alternate structure is the Galois LFSR. In this structure the shift register is interrupted in various places by the inclusion of a two-input XOR/XNOR gate that combines the next higher bit in the shift register with the LSB of the shift register. This structure consists of a number of 2-input XOR/XNOR gates rather than a single wide gate. Because of the distributed nature of this structure, it has the potential to run somewhat faster in FPGAs.

For the same polynomial, both the Fibonacci and Galois structures generate the same bit sequence, though the sequences will be offset in time if both structures start off in the same state.

The taps of the LFSR are often expressed as a polynomial where the coefficients of the polynomial are either 1 (for a tap) or 0 (no tap), and each term of the polynomial is associated with one of the shift register stages. The polynomial is expressed as an N-bit wide vector where bit i of the vector is the coefficient of the x^{i+1} term of the polynomial. There is always an implicit $x^0 = 1$ term in the polynomial, but this is not included in the vector.

As an example, the vector 0x14 denotes a 5-bit wide state vector. The binary value 0x14 = 10100 represents the polynomial $x^5 + x^3 + 1$ which taps at positions 3 and 5. The following figure shows the LFSR structure for this polynomial.



The following table shows the progression of the 5 bit state vector for the various LFSRs using the polynomial 0x14 starting at the value 1:

Fibonacci XOR	Fibonacci XNOR	Galois XOR	Galois XNOR
0x01 = 00001	0x01 = 00001	0x01 = 00001	0x01 = 00001
0x10 = 10000	0x00 = 00000	0x14 = 10100	0x10 = 10000
0x08 = 01000	0x10 = 10000	0x0a = 01010	0x0c = 01100
0x04 = 00100	0x18 = 11000	0x05 = 00101	0x02 = 00010

Fibonacci XOR	Fibonacci XNOR	Galois XOR	Galois XNOR
0x12 = 10010	0x1c = 11100	0x16 = 10110	0x05 = 00101
0x09 = 01001	0x0e = 01110	0x0b = 01011	0x12 = 10010
0x14 = 10100	0x07 = 00111	0x11 = 10001	0x0d = 01101
0x1a = 11010	0x13 = 10011	0x1c = 11100	0x16 = 10110
0x0d = 01101	0x09 = 01001	0x0e = 01110	0x0f = 01111
0x06 = 00110	0x04 = 00100	0x07 = 00111	0x17 = 10111
0x13 = 10011	0x02 = 00010	0x17 = 10111	0x1b = 11011
0x19 = 11001	0x11 = 10001	0x1f = 11111	0x1d = 11101
0x1c = 11100	0x08 = 01000	0x1b = 11011	0x1e = 11110
0x1e = 11110	0x14 = 10100	0x19 = 11001	0x0b = 01011
0x1f = 11111	0x0a = 01010	0x18 = 11000	0x15 = 10101
0x0f = 01111	0x15 = 10101	0x0c = 01100	0x1a = 11010
0x07 = 00111	0x1a = 11010	0x06 = 00110	0x09 = 01001
0x03 = 00011	0x1d = 11101	0x03 = 00011	0x14 = 10100
0x11 = 10001	0x1e = 11110	0x15 = 10101	0x0e = 01110
0x18 = 11000	0x0f = 01111	0x1e = 11110	0x03 = 00011
0x0c = 01100	0x17 = 10111	0x0f = 01111	0x11 = 10001
0x16 = 10110	0x1b = 11011	0x13 = 10011	0x18 = 11000
0x1b = 11011	0x0d = 01101	0x1d = 11101	0x08 = 01000
0x1d = 11101	0x16 = 10110	0x1a = 11010	0x00 = 00000
0x0e = 01110	0x0b = 01011	0x0d = 01101	0x04 = 00100
0x17 = 10111	0x05 = 00101	0x12 = 10010	0x06 = 00110
0x0b = 01011	0x12 = 10010	0x09 = 01001	0x07 = 00111
0x15 = 10101	0x19 = 11001	0x10 = 10000	0x13 = 10011
0x0a = 01010	0x0c = 01100	0x08 = 01000	0x19 = 11001
0x05 = 00101	0x06 = 00110	0x04 = 00100	0x1c = 11100
0x02 = 00010	0x03 = 00011	0x02 = 00010	0x0a = 01010
0x01 = 00001	0x01 = 00001	0x01 = 00001	0x01 = 00001

This LFSR generates the 31 bit sequence: 1 0 0 0 0 1 0 0 1 0 1 1 0 0 1 1 1 1 1 0 0 0 1 1 0 1 1 1 0 1 0 (for the XOR case, or the inverse of this for the XNOR case) which then repeats. This pattern is shifted depending on whether the Fibonacci or Galois structure is used as well as depending on the initial state. The PRBS generator can output one or more bits of this sequence every clock.

Typically the PRBS generator uses a maximal length polynomial. There are numerous online resources that list maximal length polynomials. The following table gives example polynomials of various length:

Size	Taps	Polynomial
2	1 2	0x3
3	2 3	0x6
4	3 4	0xc
5	3 5	0x14
6	5 6	0x30
7	6 7	0x60
8	4 5 6 8	0xb8
9	5 9	0x110
10	7 10	0x240
11	9 11	0x500
12	1 4 6 12	0x829
13	1 3 4 13	0x100d
14	1 3 5 14	0x2015
15	14 15	0x6000
16	4 13 15 16	0xd008
17	14 17	0x12000
18	11 18	0x20400
19	1 2 6 19	0x40023
20	17 20	0x90000
21	19 21	0x140000
22	21 22	0x300000
23	18 23	0x420000
24	17 22 23 24	0xe10000
25	22 25	0x1200000
26	1 2 6 26	0x2000023
27	1 2 5 27	0x4000013
28	25 28	0x9000000
29	27 29	0x14000000
30	1 4 6 30	0x20000029
31	28 31	0x48000000
32	1 2 22 32	0x80200003
33	20 33	0x100080000
34	1 2 27 34	0x204000003

Size	Taps	Polynomial
35	33 35	0x500000000
36	25 36	0x801000000
37	1 2 3 4 5 37	0x100000001f
38	1 4 5 38	0x2000000031
39	25 39	0x400100000
40	19 21 38 40	0xa000140000
41	38 41	0x12000000000
42	19 20 41 42	0x300000c0000
43	37 38 42 43	0x63000000000
44	17 18 43 44	0xc0000030000
45	41 42 44 45	0x1b000000000
46	25 26 45 46	0x30000300000
47	42 47	0x42000000000
48	20 21 47 48	0xc00000180000

Design Examples

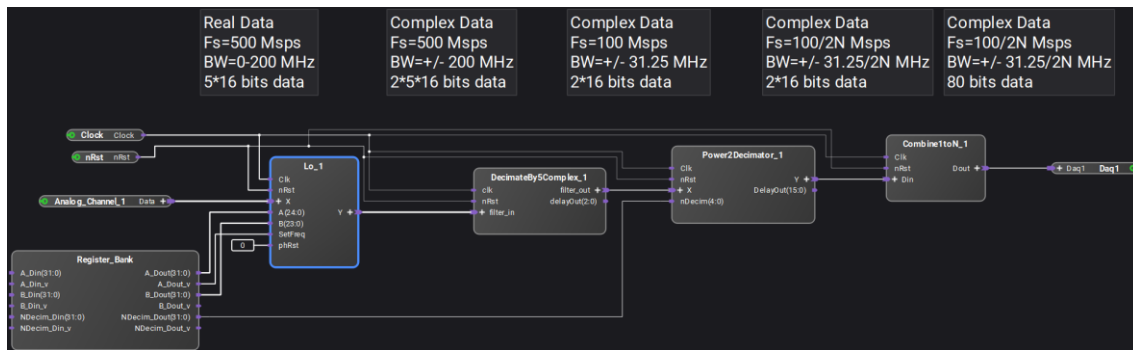
To see how these IP blocks can be used to build up and down converters, consider the following designs. The first is for a digital down converter, and the second for a digital up converter. These examples are built in a M3302, 500 Msps Combination AWG and Digitizer. Following these are examples showing how to make supersampled down and up converters.

Digital Down Converter (DDC)

For a digitizer to analyze signals with narrower bandwidth than the full digitizer bandwidth, it is common to employ a digital down converter. This allows the instrument to only look at a smaller portion of the total spectrum. It can also filter out extraneous signals that may be located in other frequency bands. It filters out noise and thus decreases the noise floor and increases the signal to noise ratio.

The basic steps for down conversion are to first mix the input with a complex LO to frequency translate the desired signal to baseband (DC). This is then low pass filtered to remove extraneous signals and prevent aliasing in the decimation step. Then it is decimated by discarding samples to lower the sample rate. Often the filter/decimate process is carried out in multiple steps for implementation efficiency.

In this real time data flow, the ADCs (Analog_Channel_1) are always running. There is no way to hold off or delay the ADC data. In this case, the data is "pushed" from the left to the right in this diagram using forward flow control only. The reverse flow control, though present, isn't really utilized.



In this example, the input ADCs of the M3302 are running at 500 Msps. The FPGA only runs at 100 MHz, so the input (Analog_Channel_1) presents 5 ADC samples every FPGA clock. This is called supersampling by 5. The five 16-bit input samples are combined into one 80 bit wide AXI-streaming bus.

The Lo (Local Oscillator Down Converter) block does the frequency translation by multiplying the real input by a complex quadrature LO signal. The output is a complex (real and imaginary) stream with the same sample rate as the input. The Lo block is configured to operate on data that is 5X supersampled. Since the output of the LO is complex, there is now 160 total data bits.

The DecimateBy5Complex block is really just a pair of real decimate by five blocks, one operating on the real data, the other operating on the imaginary data. This block reduces the data rate down to one sample per clock by first low pass filtering the input and then reducing the sample rate by a factor of 5. The output is a complex stream with a sample rate of 100 Msps and a bandwidth of +/- 31.25 MHz. Note that since the data is complex, negative frequencies aren't necessarily the complex conjugate of the positive frequencies. Thus the signal has a total bandwidth of 62.5 MHz.

This data is fed to a complex decimate by 2^N block. This can reduce the sample rate and bandwidth further (or be bypassed if $N=0$). The output of this is a complex stream of data at a sample rate potentially less than the FPGA clock rate.

In this example, the output of the entire DDC is sent to the Daq1 port of the M3302. This sends the data into DDR memory where the user can read it out and use it. Note that the output of the Power2Decimator is at most one sample per clock (2 16-bit parts due to the data being complex). The Daq1 port is expecting five 16 bit samples of data at a time. To convert between these rates, the Combine1toN block is used to combine 2-1/2 input samples (each one 2*16 or 32 bits wide) into one 80 bit output that is sent to the Daq1 port.

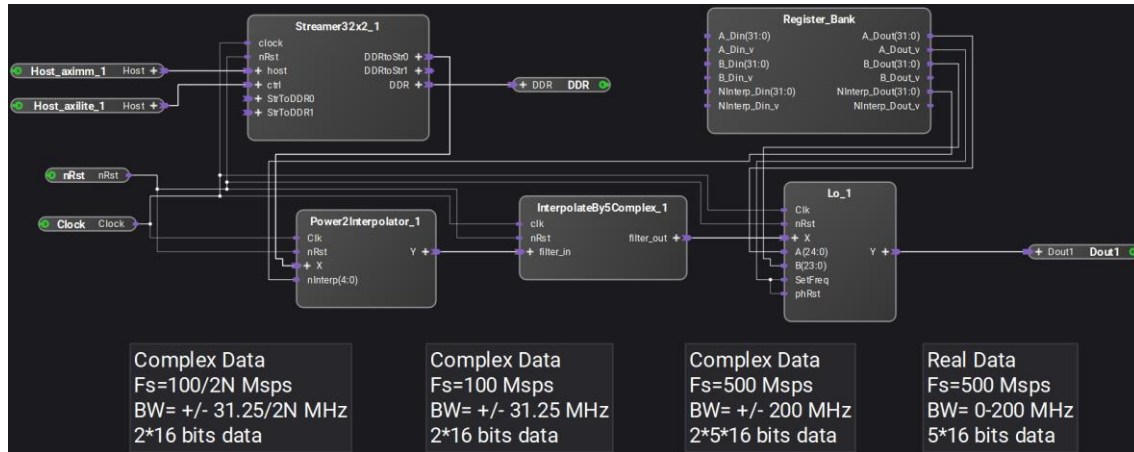
This results in a data record in memory consisting of complex pairs, each consisting of the real part of a sample and the imaginary part of the sample.

Digital Up Converter (DUC)

When a source or AWG is generating a narrow band signal, it is often easier to generate it at a lower sample rate and then upsample it and move it to the correct frequency later. This is called digital up conversion. Consider generating an AM radio signal. Rather than trying to generate the RF signal directly, it is easier to generate the signal at baseband and then move it up to whatever center frequency it needs.

The basic steps for up conversion are the reverse of the steps for down conversion. First the input signal is interpolated to a higher sample rate by adding zeroes between each input sample to increase the sample rate. This process introduces alias signals in the frequency domain. So following the interpolation step, a low pass filter is used to remove these aliasing artifacts. Finally this signal is mixed with a complex LO to translate it from baseband to the desired center frequency. At this point, only the real part of the data is used, and this is sent to the ADCs. Just as in the case of a down converter, often this interpolate/filter process is carried out in multiple steps for implementation efficiency.

In this real time data flow, the DACs (Dout1) are always running. There is no way to hold off or delay the DAC data. New data needs to be provided every clock cycle. In this case, the data is "pulled" from the right to the left in this diagram using reverse flow control only. The forward flow control, though present, isn't really utilized. Since the AWG ports in the M3302 do not support reverse flow control, they can't be used as data sources for the DUC. Instead, the Streamer32x2 block is used to pull data out of DDR memory as a data source.



Following the signal flow from the output back towards the input, the output DACs of the M3302 are running at 500 Msps. The FPGA only runs at 100 MHz, so the output (Dout1) presents 5 DAC samples every FPGA clock. This is called supersampling by 5. The five 16-bit output samples are combined into one 80 bit wide AXI-streaming bus.

The Lo (Local Oscillator) block does the frequency translation by multiplying the complex input by a complex quadrature LO signal and taking the real part. The output is a real stream with the same sample rate as the input. The Lo block is configured to operate on data that is 5X supersampled. Since the input of the LO is complex, it is 160 total data bits.

The InterpolateBy5Complex block is really just a pair of real interpolate by five blocks, one operating on the real data, the other operating on the imaginary data. This block increases the data rate up to five samples per clock by first inserting four zero samples between input points and then low pass filtering to remove images. The input is a complex stream with a sample rate of 100 Msps and a bandwidth of +/- 31.25 MHz. Note that since the data is complex, negative frequencies aren't necessarily the complex conjugate of the positive frequencies. Thus the signal has a total bandwidth of 62.5 MHz.

The input to the InterpolateBy5Complex block is generated by the complex interpolate by 2^N (Power2Interpolator) block. This can increase the sample rate and bandwidth from a lower sample rate (or be bypassed if $N=0$). The input to this block is a complex stream of data at a sample rate potentially less than the FPGA clock rate.

Since the input to the Power2Interpolator can be less than the FPGA clock rate, its data must be sourced from something that supports reverse flow control (so that the Power2Interpolator indicates when and how fast it needs new data). The AWG blocks of the M3302 do not support reverse flow control and can not be used in this application. Instead, the data for the Power2Interpolator is sourced from the Streamer32x2 block which reads data from DDR memory.

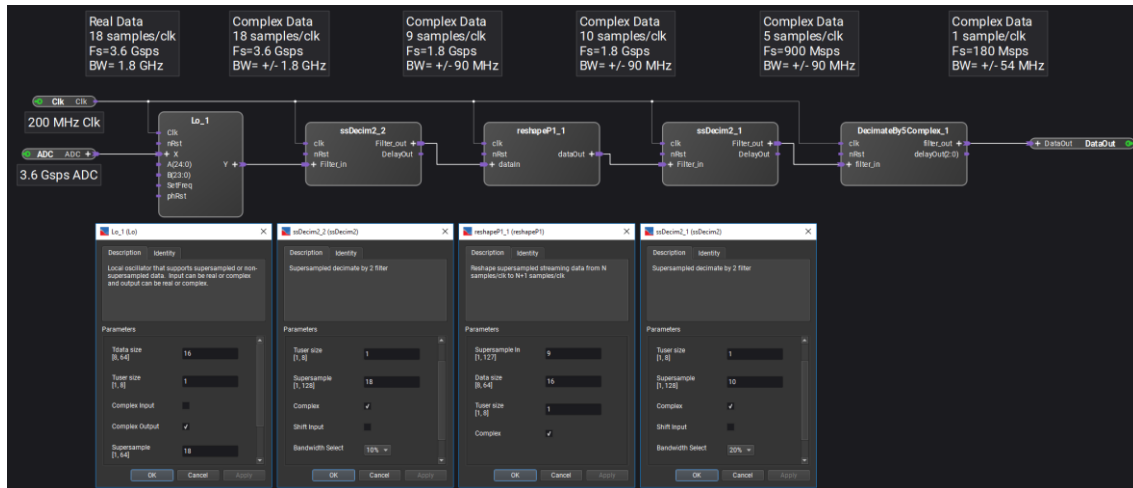
The data record in DDR memory consisting of complex pairs, each consisting of the real part of a sample and the imaginary part of the sample.

Supersampled Digital Down Converter

In today's instruments, it is not uncommon for data rates to be higher than the usable clock rates of FPGAs. In these cases, for real time processing the FPGA has to process multiple samples per FPGA clock. For example, you might have a digitizer where the ADC runs at a rate of 3.6 Gsps while the FPGA can only run at 200 MHz. That means the FPGA must process 18 samples/clock in order to keep up with the ADC. Sometimes the signal of interest is relatively narrow band. In this case, a digital down converter can be used to move the signal of interest

down to baseband. Then decimation filtering is used to reduce the sample rate without generating alias signals.

The block set in Pathwave FPGA's DSP library make it easy to make digital down converters for supersampled ADCs. Suppose we wish to reduce the sample rate for this example down enough so that it is no longer supersampled. That is, reduce the sample rate to at most one sample per clock. This can be done by the following design which shows the blocks used, how they are configured, and signal size and bandwidth. For clarity, the control and reset signals are not shown connected here.



The 3.6 Gps ADC signal comes in as 18 parallel real samples at a rate of 200 MHz. The first step is the LO to mix the signal of interest to baseband. This is configured for real input, complex output, and a supersample value of 18. After the LO, the signal is complex so the remaining blocks are configured for complex data. The ssDecim2_2 block does the first level of decimation filtering, cutting the sample rate (and the supersample value) in half, down to 9 samples/clk. Since we know the final bandwidth of the DDC is going to be +/- 54 MHz, we can use the smaller ssDecim2 filter that only has a 10% passband. Signals in the transition band of this filter may alias, but these will be filtered out in later stages.

Before we can do another decimate by two operation, we need to "reshape" the data stream. The output supersample value for the ssDecim2_2 block is 9 samples/clk. But the next ssDecim2 block requires that the input supersample value must be even. To change this, we use the reshapeP1 block to convert from 9 samples/clk to 10 samples/clk. Note that after the reshapeP1 block, data will not necessarily flow every clock. Only 9 out of 10 clocks will have new data. This is handled by the forward flow control mediated by the TVALID signal in the AXI-streams. The reshapeP1 block does not change the signals sample rate - that is 1.8 Gps both before and after the reshapeP1. The supersample value and the data's transaction rate do change.

The ssDecim2_1 cuts the sample rate in half again. This time we needed to 20% filter to support the final output bandwidth of the DDC. We could use 3 more stages of decimate-by-two filters (along with more reshapeP1 blocks) to bring the final sample rate down to 112.5 Msps, but we can also use the DecimateBy5Complex block to bring the final sample rate down to 180 Msps as shown here.

If further levels of decimation are desired, the Power2Decimator block can be used to reduce the sample rate further.

The following table shows the resources (multipliers or DSP blocks) used in this example. Even though the earlier IP blocks are more highly supersampled, they use fewer DSP blocks due to the reduced fractional bandwidth requirements.

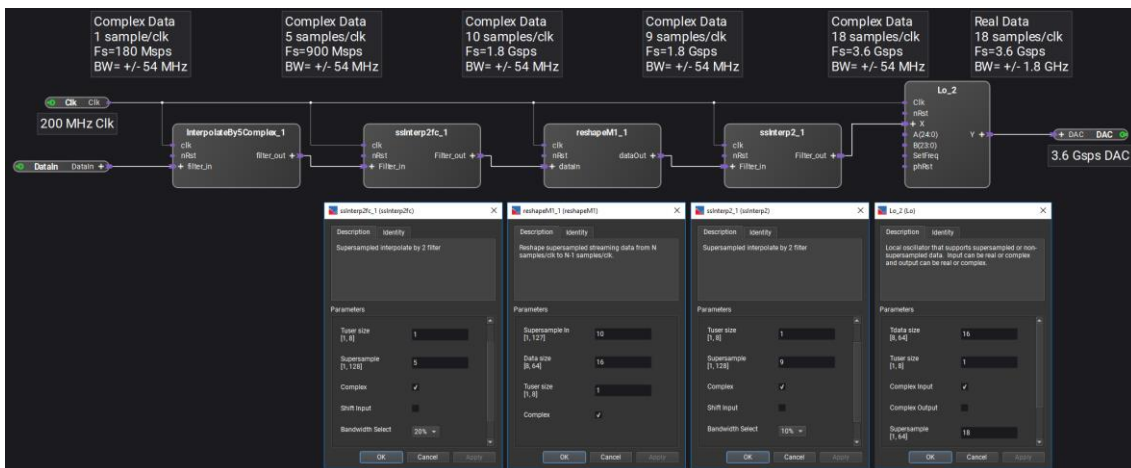
Block	Lo_1	ssDecim2_2	ssDecim2_1	DecimateBy5Complex	Total
DSP/sample	2	2	4	102	
Number of outputs	2*18	2*9	2*5	1	

Block	Lo_1	ssDecim2_2	ssDecim2_1	DecimateBy5Complex	Total
Total DSP	72	36	40	102	250

Supersampled Digital Up Converter

In today's instruments, it is not uncommon for data rates to be higher than the usable clock rates of FPGAs. In these cases, for real time processing the FPGA has to process multiple samples per FPGA clock. For example, you might have a source where the DAC runs at a rate of 3.6 Gsps while the FPGA can only run at 200 MHz. That means the FPGA must process 18 samples/clock in order to keep up with the DAC. Sometimes the signal of interest is relatively narrow band. In this case, a digital up converter can be used to interpolate the input signal up to the DAC's sample rate followed by an LO to move the signal to the desired location in the output spectrum.

The block set in Pathwave FPGA's DSP library make it easy to make digital up converters for supersampled DACs. Suppose we wish to increase the sample rate for this example starting from a rate that it is no longer supersampled. That is, start with a sample rate of at most one sample per clock. This can be done by the following design which shows the blocks used, how they are configured, and signal size and bandwidth. For clarity, the control and reset signals are not shown connected here.



In this case, it is more convenient to follow the data from the output DAC back through the input DataIn. That is because unlike the case of the DDC, where the data flow rate was driven by the constant rate ADC, for the DUC shown here, the data flow rate is driven by the constant rate DAC output.

The 3.6 Gsps DAC requires 18 parallel real samples at a rate of 200 MHz. To get this, the LO is used to mix a lower bandwidth baseband signal up to the desired center frequency. This LO is configured for complex input, real output, and a supersample value of 18. Prior to the LO, the signal is complex so the remaining blocks are configured for complex data. The ssInterp2_1 block does the final level of interpolation, doubling the sample rate (and the supersample value) from 9 samples/clock up to 18 samples/clock. Since we know the input bandwidth of the DUC is only +/- 54 MHz, we can use the smaller ssInterp2 filter that only has a 10% passband. Due to the earlier interpolation filters, there are no signals in the transition band of this filter and hence no aliased outputs.

Since the output of the ssInterp2 block is always an even number of samples per clock and ssInterp2_1 needs an odd number of samples per clock, we need to "reshape" the data stream. To change from an even number of samples per clock to an odd number of samples per clock we use the reshapeM1 to convert from 10 samples/clock input to 9 samples/clock output. Note that prior to the reshapeM1 block, data will not necessarily flow every clock. Only 9 out of 10 clocks will have new data. This is handled by the reverse flow control mediated by the TREADY signal in the AXI-streams. All blocks prior to the reshapeM1 block will need to support reverse flow control and the TREADY signal. The reshapeM1 block does not change

the signals sample rate - that is 1.8 Gsps both before and after the reshapeM1. The supersample value and the data's transaction rate do change.

The ssInterp2fc_1 supports reverse flow control and doubles the sample rate. This time we needed to 20% filter to support the input bandwidth of the DUC. We could use 3 more stages of interpolate-by-two filters (along with more reshapeM1 blocks) to start from an initial sample rate of 112.5 Msps, but we can also use the InterpolateBy5Complex block to end up with an initial sample rate of 180 Msps as shown here.

If further levels of interpolation are desired, the Power2Interpolator block can be used to start from even lower sample rates.

The following table shows the resources (multipliers or DSP blocks) used in this example. Even though the later IP blocks are more highly supersampled, they use fewer DSP blocks due to the reduced fractional bandwidth requirements.

Block	InterpolateBy5Complex	ssInter2fc_1	ssInterp2_1	Lo_1	Total
DSP/sample	98	4	2	2	
Number of elements	1	2*5	2*9	18	
Total DSP	98	40	36	36	210

IP Repositories

IP repositories are libraries of blocks that are loaded into PathWave FPGA. There are four types of IP repositories supported inside PathWave FPGA:

- Default PathWave FPGA IP repository: a repository that is shipped inside the PathWave FPGA Installation directory structure and is permanent. IPs defined in this repository will be loaded for all projects, as long as they meet the hardware support criteria.
- BSP IP repository: a IP repository that is shipped inside a BSP installation.
- User defined IP repository: a machine scoped user-defined list of directories that include IP definitions. These directories can be defined in the Settings dialog (**Tools > Settings**). To load an IP repository, use the [Settings Dialog](#).
- Project defined IP repository: a project scoped user-defined list of directories that include IP definitions. These directories can be defined in the Project Settings dialog (**Project > Project Settings**). To load an IP repository, use the [Project Settings Dialog](#).

To learn how to create an IP repository, refer to the [IP Developers Guide](#).

IP will be found recursively in each repository location. All valid IP will be added into the library blocks. If any problems are encountered with loading, a dialog will popup to display the errors. Xilinx Vivado IP is excluded from this search.

Imported User IP


In addition to IP developed using the Library tools, the PathWave FPGA software allows importing and integration of custom IP into a project. User IP is developed using external FPGA tools; the PathWave FPGA software is not intended for developing IP from scratch. However, once the user has created an IP, the IP may be imported by the PathWave FPGA software.

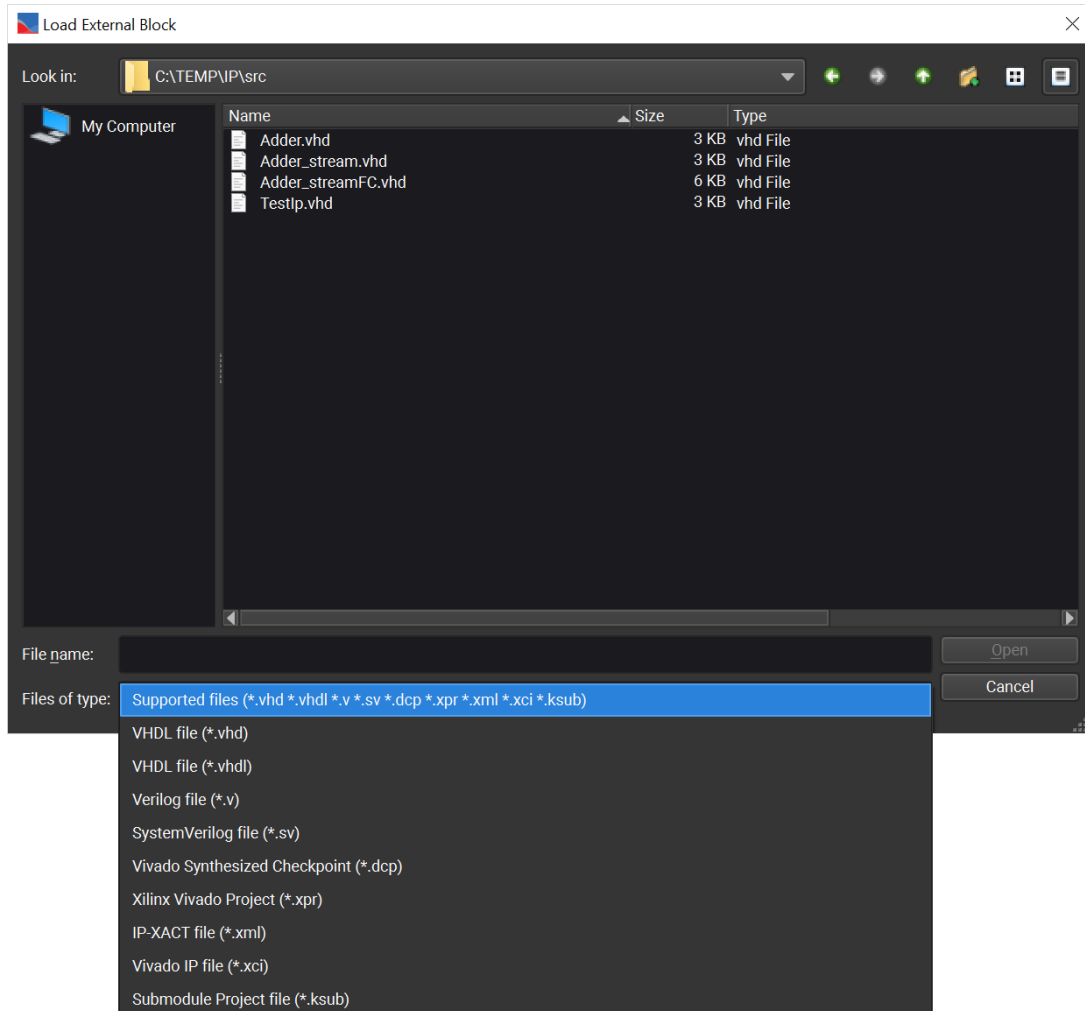
The user can import IP from different source files, including the following:

- VHDL source files (*.vhd, *.vhdI)
- Verilog source files (*.v).
- Xilinx Vivado projects (*.xpr).
- Vivado Synthesized Checkpoints (*.dcp).
- IP-XACT files (*.xml).
- Vivado IP files (*.xci)

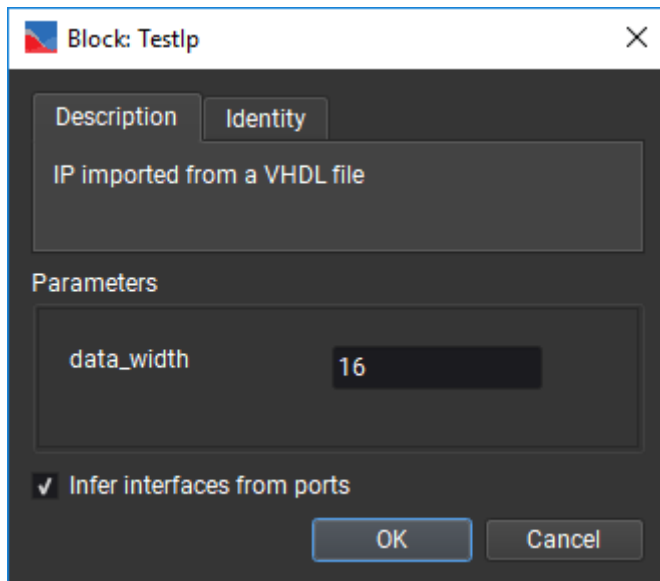
- [PathWave FPGA Submodules](#)

To import a user IP:

1. Click the  **Add External Block** button on the main toolbar, or select **Project > Add External Block...** from the menu. In the image below, notice the file types that are available for importing.

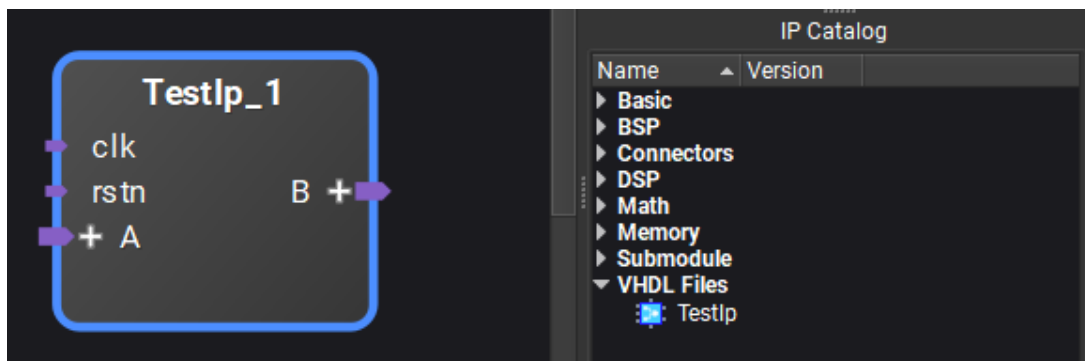


2. Navigate to select the file to be imported into the project. Click **Open** to import the file.
3. Some imported IP may have parameters that can be configured, such as bus widths. Change the initial parameter value as appropriate for your design.



4. Some imported IP may not have the ports already grouped into easy to use [interfaces](#). The import dialog will have a check box to infer interfaces from these ports. If the interface inference gives undesired results, remove the IP and import it again with the box unchecked. If interface inference is usually *not* desired, clear the Infer Interfaces checkbox in the [Settings Dialog](#).

The IP is inserted in the project, where it can be connected to other blocks.



The block name appears in the IP Catalog for reuse as shown above. To remove a block, right-click the block name and choose **Remove**.

- If the User IP file is moved, the icon appears at the top of the block indicating the file cannot be found. Once the file is moved back, or the path is changed, right-click the block to reload the IP and remove the icon on the block.
- If the underlying code for the IP is changed, the icon can appear to signify an alert condition. Once the code is corrected, the block can be reloaded to remove the icon on the block.
- If there is an error in the IP, the icon appears. Hover the mouse cursor over the icon to see what the error is.

Importing an HDL file with Dependencies

If you want to import an HDL file with dependencies, you will need to create an IP-XACT file for the desired HDL entity following the instructions in the [IP Developers Guide](#). Then, inside the `<ipxact:fileSet>` where the source files for “synthesis” are defined, add as many `<ipxact:file>` entries as required to define the source VHDL file along with all the files that it depends on.

For example, assume that the desired component is called “Filter” and is defined in “C:\MyIPs\FilterIP\FilterTop.vhd”. Then, assume that the implementation of “Filter” depends on another component, named “Tap”, which is defined in “C:\MyIPs\FilterIP\Tap.vhd”. To successfully load the component “Filter” in PathWave FPGA, you need to create an IP-XACT (e.g. in “C:\MyIPs\FilterIP\Filter.xml”) file with the following statements in the fileset entry:

Code Block 1 IP-XACT fileset snippet

```
<ipxact:fileSets>
  <ipxact:fileSet>
    <ipxact:name>synthesis</ipxact:name>
    <ipxact:file>
      <ipxact:name>FilterTop.vhd</ipxact:name>
      <ipxact:fileType>vhdlSource</ipxact:fileType>
    </ipxact:file>
    <ipxact:file>
      <ipxact:name>Tap.vhd</ipxact:name>
      <ipxact:fileType>vhdlSource</ipxact:fileType>
    </ipxact:file>
  </ipxact:fileSet>
</ipxact:fileSets>
```

When the IP-XACT file is created, you can use the process above to load the IP-XACT xml file.

Importing an HDL file without Dependencies

When an HDL file is imported without dependencies, only the module or entity declaration will be examined in order to determine the ports that will be available for connections within a PathWave FPGA graphical design. Any syntax issues or errors that may exist elsewhere in an imported HDL file may not be detected or flagged.

For Verilog HDL files, module declarations should be limited to the features and format shown in the following examples:

```
module foo (clk, d_out);
  input wire clk;
  output reg [31:0] d_out;

  endmodule
```

or:

```

module foo
#(
    parameter myParam1 = 14,
    parameter myParam2 = 32
)
(
    input wire clk,
    output reg [31:0] d_out
);

endmodule

```

or:

```

module mymodule(input      clk,
                 input [7:0] inBus, // Comments are okay
                 output     outWire,
                 output [15:0] outBus);

endmodule

```

For VHDL source files, entity declarations should be limited to features shown in the following example:

```

library ieee;
use ieee.std_logic_1164.all;

entity foo is
    generic (
        width : integer := 4
    );
    port (
        clk : in  std_logic;
        d_out: out std_logic_vector(width-1 downto 0)
    );
end foo;


```

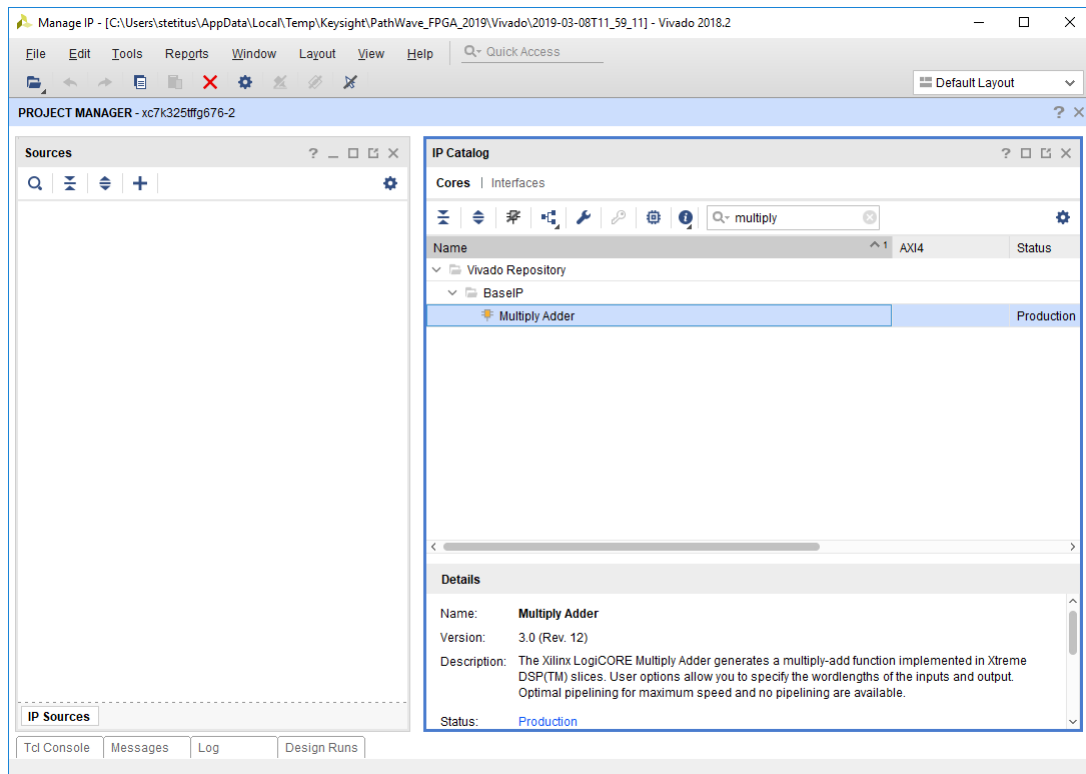
A list of known limitations for IP import can be found in [VHDL Support](#) and [Verilog Support](#) sections.

Vivado XCI (Xilinx Core Instance)

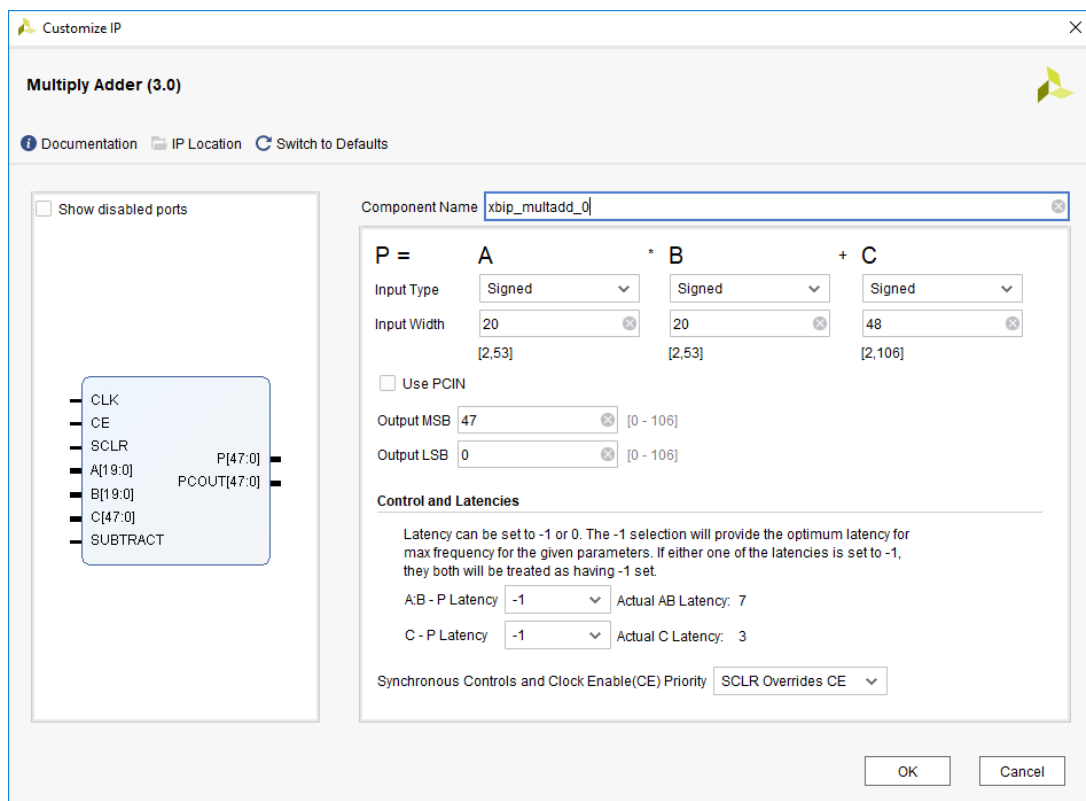
Invoking Vivado IP tool

PathWave FPGA allows you to import Vivado IPs from the Xilinx Vivado IP Catalog and integrate them into your project.

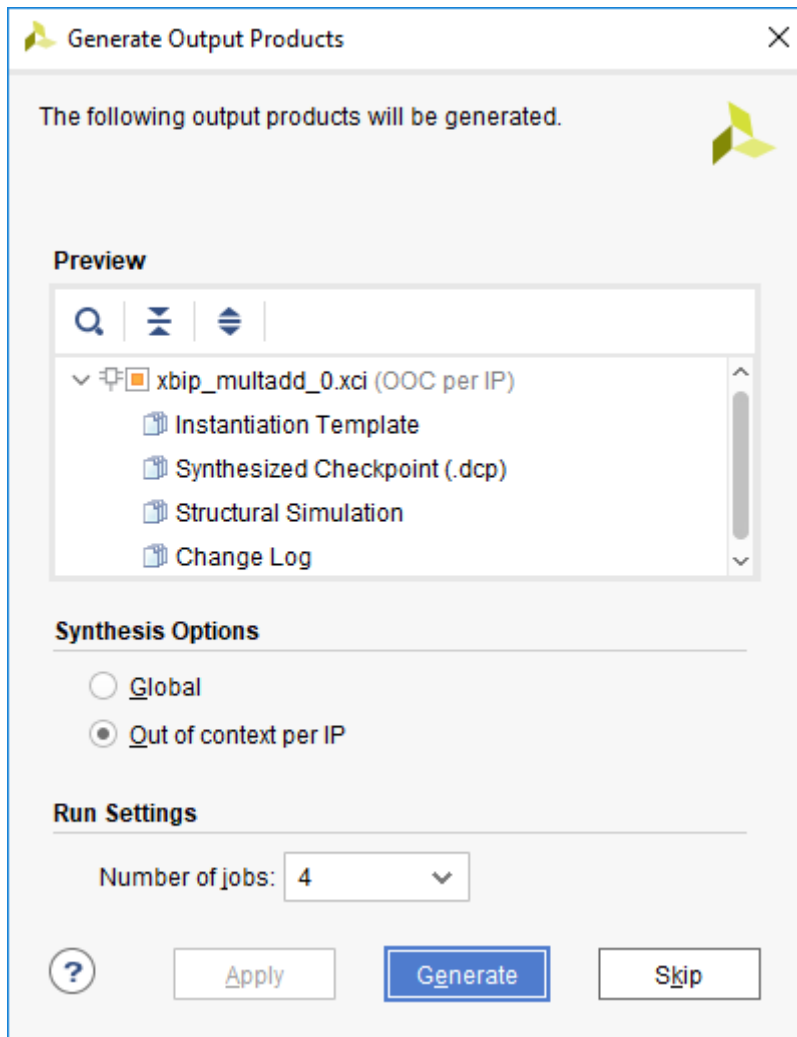
1. Click on the  **Launch Vivado IP Tool** button on the main toolbar.
2. Select a Vivado IP block from the IP Catalog and double-click it.



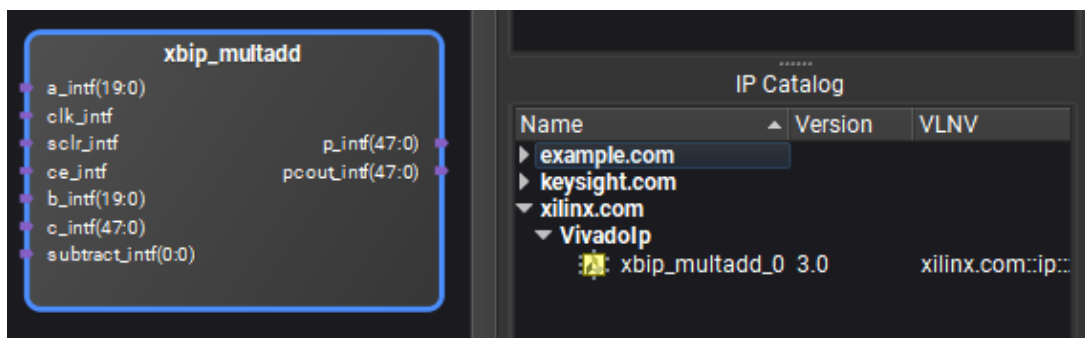
- Configure the IP properties and then press OK.




- Click the Skip button. PathWave FPGA always regenerates Vivado IP during bitfile generation, so the output products created by clicking Generate are not needed.



- If you need any other Vivado IP, repeat steps 2-4 to generate them. When you are done, close Vivado.
- PathWave FPGA will show the configured IP in the IP Catalog section under vendor *xilinx.com* and library *ip*. Add an instance to your design in the same way as any other IP.



Importing a Vivado XCI File

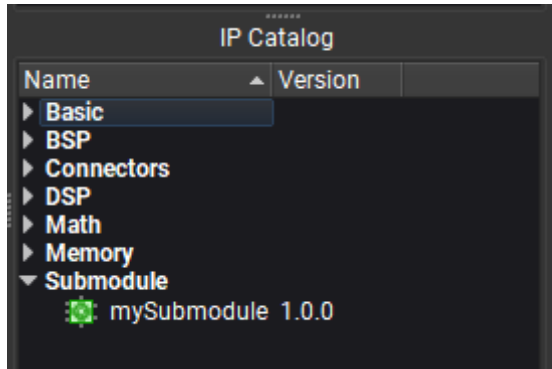
Vivado IP may also be imported from another location by browsing for the .xci file with  **Add External Block**. See [Imported User IP](#) for more details.

Note that for some IP blocks, Vivado will generate an IP-XACT file that does not conform to the IP-XACT specification. PathWave FPGA will report errors when trying to import such an IP block. Please see [Importing IP with Invalid IP-XACT](#) in the appendix for more information.

PathWave FPGA Submodule

PathWave FPGA submodules allow you to define your design hierarchically. In addition, you can share submodules in [IP repositories](#).

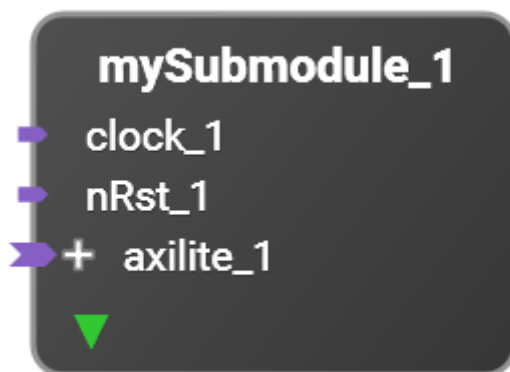
The submodules that can be added to your design are displayed in the IP Catalog pane with an icon of a chip in green color.



When a submodule is created from a sandbox project (see [Creating a New Submodule Project](#)), it is added to the Submodule pane for that project.

Submodules may also be added to a project by selecting **Project > Add External Block...** and navigating to the desired submodule project file with the `.ksub` filename extension.

Submodules can be visually distinguished from other blocks in the canvas with a small green triangle in the bottom left corner of the block.



Naming Conventions

Within PathWave FPGA, things like Instance names and Register names must be unique and valid HDL identifiers. Specifically they must follow these rules:

1. A name must start with an alphabetic character (A-Z or a-z).
2. A name can only consist of alphanumeric characters and underscores (A-Z, a-z, 0-9, _).
3. A name must end with an alphanumeric character (A-Z, a-z, 0-9).
4. A name can not be a reserved word (listed below).
5. Names are not case sensitive. Thus *myreg*, *MYREG*, *MyReg* are all considered to be the same name.

6. Register names must be unique inside their Register Block.

The rules for display names on blocks are relaxed for user convenience. All displayable Unicode characters are allowed within a display name, and the name does not need to be valid HDL. The one restriction on display names is that all display names must be unique in their sandbox or submodule schematic. For instance, you cannot have two blocks named "my block" in the same schematic.

Reserved Words

The following are reserved words and can not be used as names:

```
abs, access, after, alias, all, always, always_comb, always_ff,
always_latch, and, architecture, array, assert, assign, assume,
attribute, automatic, before, begin, bind, bins, binsof, bit, block,
body, break, buf, buffer, bufif0, bufif1, bus, byte, case, casex,
casez, cell, chandle, class, clocking, cmos, component, config,
configuration, const, constant, constraint, context, continue, cover,
covergroup, coverpoint, cross, deassign, default, defparam, design,
disable, disconnect, dist, do, downto, edge, else, elsif, end,
endcase, endclass, endclocking, endconfig, endfunction, endgenerate,
endgroup, endinterface, endmodule, endpackage, endprimitive,
endprogram, endproperty, endsequence, endspecify, endtable, endtask,
entity, enum, event, exit, expect, export, extends, extern, file,
final, first_match, for, force, forever, fork, forkjoin, function,
generate, generic, genvar, group, guarded, highz0, highz1, if, iff,
ifnone, ignore_bins, illegal_bins, import, impure, in, incdir,
include, inertial, initial, inout, inout, input, inside, instance,
int, integer, interface, intersect, is, join, join_any, join_none,
label, large, liblist, library, linkage, literal, local, localparam,
logic, longint, loop, macromodule, map, matches, medium, mod, modport,
module, nand, negedge, new, next, nmos, nor, nor, noshowcancelled,
not, notif0, notif1, null, of, on, open, or, others, out, output,
package, packed, parameter, pmos, port, posedge, postponed, primitive,
priority, procedure, process, program, property, protected, pull0,
pull1, pulldown, pullup, pulsestyle_ondetect, pulsestyle_onevent,
pure, rand, randc, randcase, randsequence, range, rcmos, real,
realtime, record, ref, reg, register, reject, release, rem, repeat,
report, return, rmos, rol, ror, rpmos, rtran, rtranif0, rtranif1,
scalared, select, sequence, severity, shared, shortint, shortreal,
showcancelled, sig, signal, signed, sla, sll, small, solve, specify,
specparam, sra, srl, static, string, strong0, strong1, struct,
subtype, super, supply0, supply1, table, tagged, task, then, this,
throughout, time, timeprecision, timeunit, to, tran, tranif0, tranif1,
transport, tri, tri0, tril, triand, trior, trireg, type, typedef,
unaffected, union, unique, units, unsigned, until, use, uwire, var,
variable, vectored, virtual, void, wait, wait_order, wand, weak0,
weak1, when, while, wildcard, wire, with, within, wor, xnor, xor
```

Name Collisions

IP Catalog Level

PathWave FPGA uses VLVN for uniquely identifying IP inside the IP Catalog. VLVN stands for Vendor-Library-Name-Version and is a concept introduced by IP-XACT. For files that do not define a VLVN, such as HDL files, PathWave FPGA uses the IP module name to create one. When loading an IP in the [IP Catalog](#), these possible name collisions have been identified:

- **Two IPs have the same VLVN.** In this case, PathWave FPGA will give the user the option to update to the desired definition. This option is not available if the IPs are coming from an IP repository. In the latter case, user will have to resolve it using one of the workarounds.

- **Two IPs have the same module name, but they do not define a VLNV.** The user will have to resolve this case using one of the workarounds.
- **An IP is using a module name of a [reserved word](#), even if it has a VLNV.** A possible workaround in this case is to create a wrapper for that IP which will have a non-colliding name
- **Every other case is allowed.**

Design Canvas Level

PathWave FPGA allows any number of instances of an IP, that is loaded in the [IP Catalog](#), to be used inside the design canvas. This is true even for different IP that have the same module name.

To achieve this, PathWave FPGA identifies the conflicting instances and wraps them in Vivado out-of-context builds, bringing the output product back to the design with a different name. This ensures that no conflicts will appear during the build of the complete design, with an **exception** discussed below. The total build time of the project is increased during the synthesis portion of the build. The increase is proportional to the number of conflicting IP instances that exist in the design, and the number of the different configurations (parameters initialization) of instances per IP. To avoid this overhead, user can try to apply one of the workarounds described below whenever applicable.

Exception

PathWave FPGA cannot identify module name collisions in case of sub-components inside the hierarchy of an IP.

As an example, imagine the case of an IP named `ArithmeticUnit` that uses a sub-component named `Adder`. The IP `ArithmeticUnit` is loaded in the [IP Catalog](#) and an instance of it is placed in the design canvas. At the same time, imagine that an IP named `Adder` exists in the [IP Catalog](#) which is different from the one the `ArithmeticUnit` is using in its definition. If an instance of the latter `Adder` is used also in the design, PathWave FPGA will not be able to identify the collision which will probably lead to a build failure. The same issue can arise if there are collisions between the sub-components of one IP and the sub-components of another.

To work around these cases, the user has to identify the colliding IP, and use one of the workarounds applicable below.

Workarounds

When a name collision is detected, the user will have to take action and resolve it.

- **Rename the IP to a non-conflicting name or VLNV.** This is simplest and fastest solution. If the user is not the owner of the IP, it might not be feasible and the user has to follow the second workaround.
- **Load only the IPs that are necessary for the project.** This is possible only if the conflicting IPs are not needed at the same time. Note that in the case of unwanted IPs that are loaded through an IP Repository location, user has to either remove the IP Repository location, which will also remove any other IP loaded from the same place, or move the conflicting IP definition file (IP-XACT file) outside of the IP Repository location or any sub-directory.
- **Create a [Submodule](#) design to wrap the IP.** This has the downside of build time overhead
- **Create a wrapper entity/module for the failing IP.** This option will only work if the reason of the name collision is a [reserved word](#) or the name of the IP matches the name of a sandbox interface. The wrapper entity has to use a non-conflicting name.

Building your FPGA Logic

- [Generating the Bit File](#)

Generating the Bit File

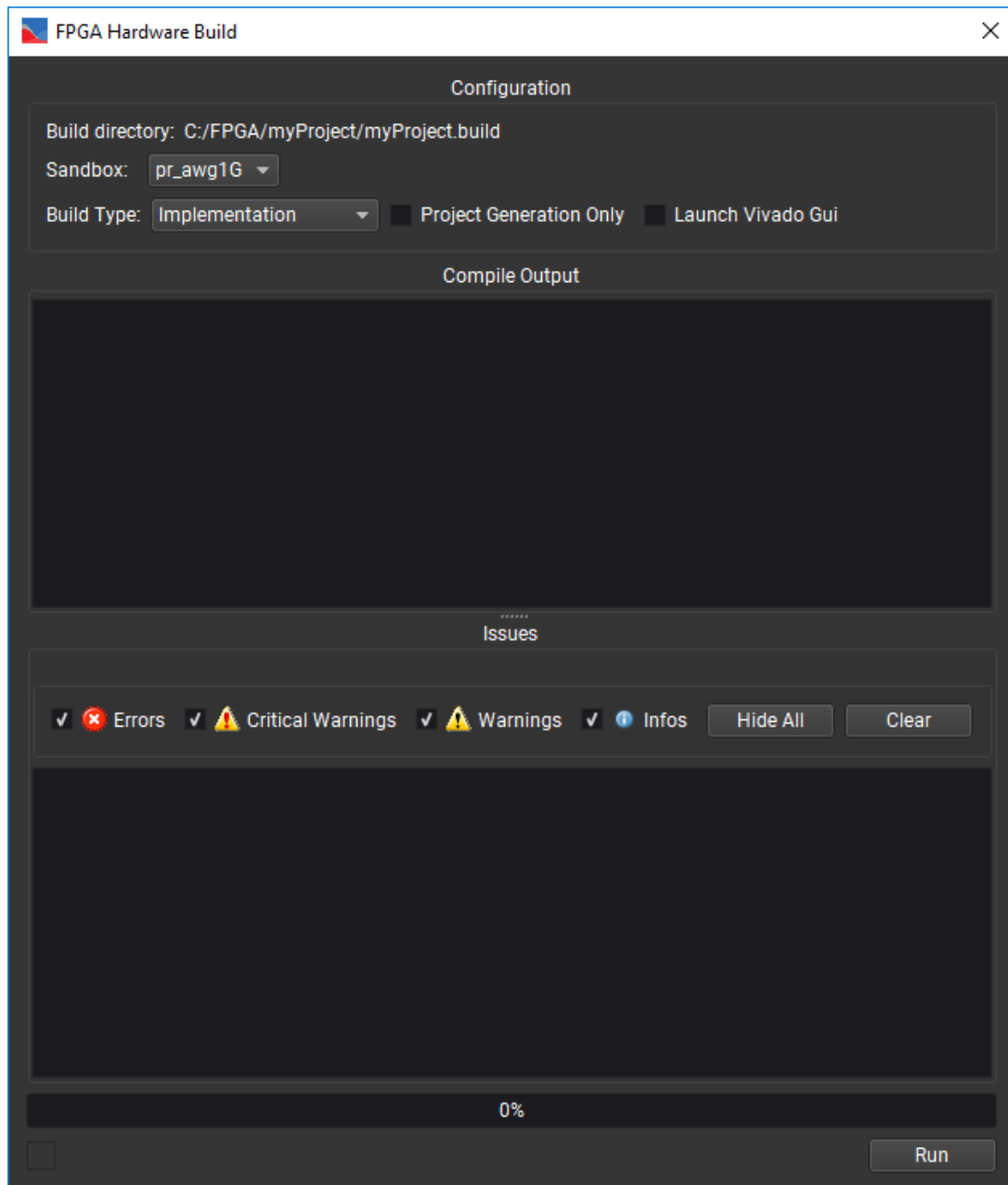
- [Synthesizing and Implementing your Design inside of PathWave FPGA](#)
 - [Different FPGA Build options](#)
 - [Monitoring the Build](#)
 - [Exploring the Build Output](#)
- [Building your Design using Vivado](#)
 - [Generating a Vivado Project](#)
- [Troubleshooting](#)
 - [Drive mapping remaining after build completion](#)
 - [Generated project synthesis fails because paths are too long](#)

Synthesizing and Implementing your Design inside of PathWave FPGA

After creating your new hardware project and adding your FPGA logic, you are ready to generate the bit file that implements your design.

To build the bitfile based on your design, complete the following steps:

1. Select **Module> Generate Bit File...** or click the toolbar icon with tooltip "Generate Bit File...". The FPGA Hardware Build dialog will appear.



2. Choose the sandbox that you want to target for this build.

Sandbox: pr_awg1G_410 ▼

3. Choose the **Implementation** build type. This will build the complete project, including the bit file.

Build Type: Implementation ▼

4. Click **Run** to start the build.

Different FPGA Build options

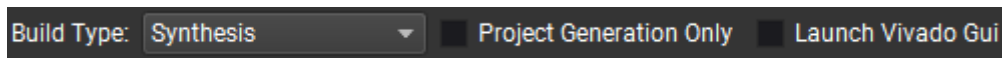
The FPGA Hardware Build has two different build options that affect what options are displayed by the build dialog. The version of the BSP affects what options are available. The same basic build types are available between each, but the newer BSPs add additional usability features.

Basic Build Types (common between all BSPs)

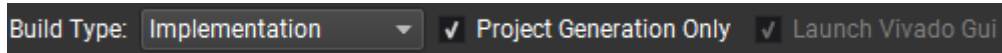
- Synthesis: Builds what is present in the sandbox only.
- Implementation: Builds what is present in the sandbox and places it into the static region of the selected BSP and runs to bit generation.
- Implementation from DCP: Takes a provided DCP and places it into the static region of the selected BSP and runs to bit generation.

Usability features (newer BSPs)

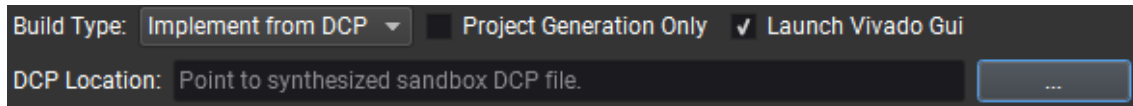
- Two new options are available, launch the Vivado GUI to monitor the build, and only run project generation on a design.



- When project generation is selected, the Vivado GUI will always be launched.



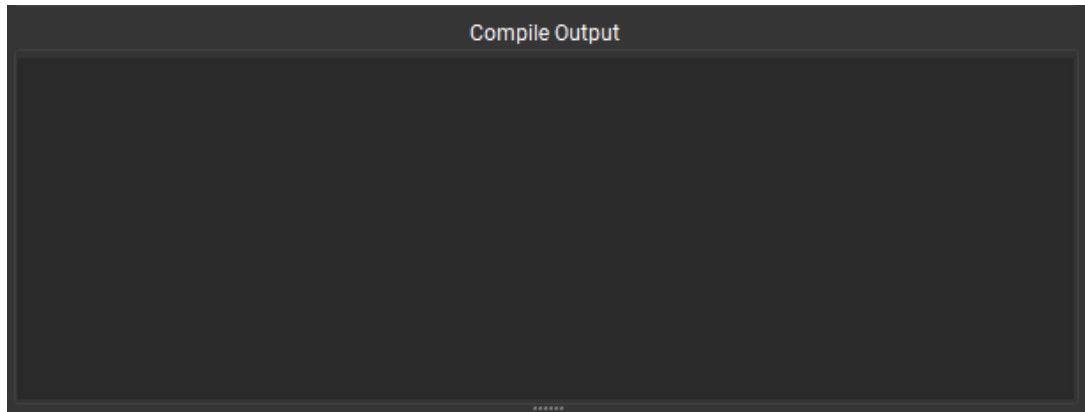
- The GUI can be selected to launch regardless of project generation



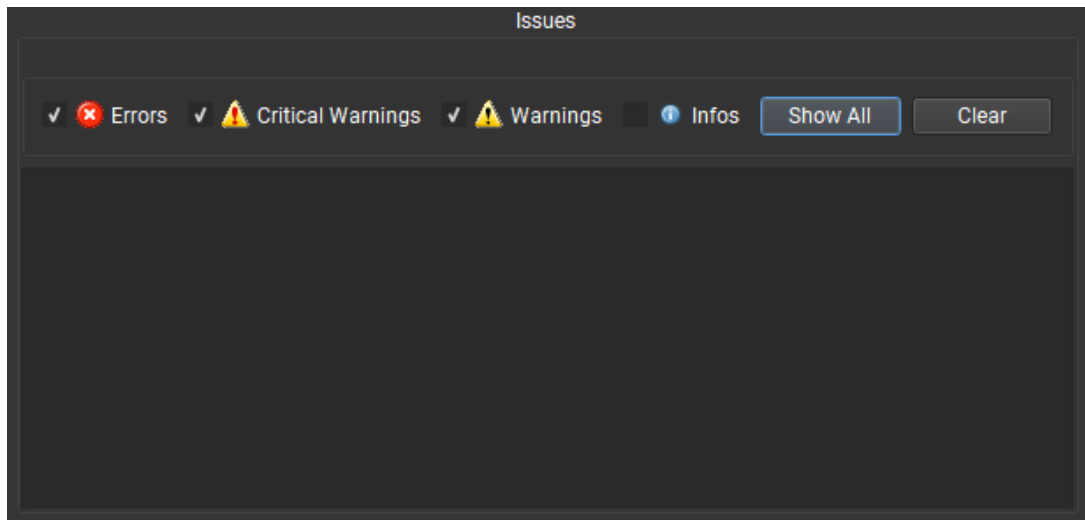
Monitoring the Build

The FPGA Hardware Build dialog contains several panes to monitor the progress of the build:

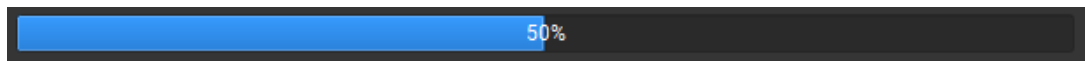
- The Compile Output pane displays all build output.



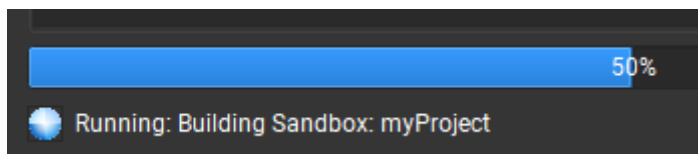
- The Issues pane shows filtered build output. You can set the filters by checking the boxes (Errors, Critical Warnings, etc.) at the top of the Issues pane. The filters can be set at any time while the build is running or after it is complete.



- The progress bar shows the approximate progress of the build.



- The status bar at the bottom left shows what step of the build is being performed. When the build is finished, the build status will be displayed.



- At the beginning of the build, a mapping will be created in the windows file system from the build directory to an open drive letter.
 - This mapping is used to ensure no windows path length limits are exceeded.
 - The mapping will be removed at the completion of the build.

Exploring the Build Output

The Build directory field in the Configuration pane specifies the parent directory of the build artifacts, including the generated bit file. The Program Archive of the generated bit file may be recognized by its k7z file extension.

Build directory: C:/FPGA/myProject/myProject.build

If the build was successful, the build artifacts are copied to an artifact directory for future reference. Each set of build artifacts has its own time and date stamped directory. In this example, one artifact directory could be named myProject.data\bin\myProject_2018-04-04T14_21_55.

To learn more about the build output structure, refer to the [Project Directory Structure](#) section.

Building your Design using Vivado

PathWave FPGA provides a path to a Vivado flow for users who want to use advanced features in Vivado, such as adding placement constraints.

Generating a Vivado Project

To start the advanced build flow and leave PathWave FPGA build environment, follow the steps listed below.

1. Open a new or existing PathWave FPGA project, and navigate to the FPGA Hardware Build dialog.
2. Select the sandbox you wish to implement with the sandbox drop down, and select the **Implementation** build type.
3. Check the **Project Generation Only** checkbox.
4. Click **Run**.
 - a. If any build errors are encountered, solve the errors before continuing.
5. After synthesis of the sandbox completes, Vivado will launch and link the sandbox into the static region.
 - a. The project folder for the design can be located in the .build folder of your project with a timestamped folder.
6. A Vivado project is now created and ready for development.
 - a. When finished with any additional Vivado steps, proceed to the next point.
7. In the Tcl command line, type FinishBuild and press enter.
 - a. FinishBuild is a custom command that PathWave FPGA generates and puts into the Vivado environment when the project is created.
 - b. If any problems are encountered, solve them and repeat this step
8. If no errors are found, the build will finish and the build outputs will have been generated in the project folder that this project resides in.
9. Close Vivado and return to PathWave FPGA.

At this point, PathWave FPGA will detect that Vivado has closed and will end the build process. The build outputs will be captured and stored in a timestamped .data folder.

Troubleshooting

In this section, we will discuss potential issues that can arise during the build process and possible solutions to those problems.

Drive mapping remaining after build completion

If the drive mapping that is established at the end of a build is not cleaned up successfully at the end of the build, either of the following can be done to remove the mapping.

- Open CMD
- Run "subst /D {drive letter}:"

or

- Restart your machine

Either of the above methods will remove the drive mapping from your machine.

Generated project synthesis fails because paths are too long

PathWave FPGA maps the build directory at the start of every build, but generated projects do not have this same feature. If your generated project fails synthesis because of windows paths exceeding 260 characters in length, do the following steps.

- Close Vivado project
- Open CMD
- Run "subst {Unmapped Drive Letter}: {Working Directory}"

- Navigate to new mapped drive and open Vivado project.

Your Vivado project will now have a shorter path and should get around the windows path length limit.

Simulating your FPGA Logic

- [Simulation Testbench Design](#)
- [Test Bench Address Mapping](#)

Simulation Testbench Design

Simulation Testbench

When testing a design, it is usually more efficient and easier to test various IP blocks by themselves. For system level testing, however, testing the entire sandbox design may be needed. This can be used to test the interactions between IP blocks as well as the interface between the sandbox design and the static region design.

To simulate the sandbox, a testbench that provides stimulus to the sandbox and receives the response from the sandbox must be written. The use of PathWave Standard Interfaces simplifies this process as it limits the number of different interfaces that need to be modeled.

After a PathWave FPGA Synthesis, the generated files are in a <build directory>. For a design called <design name>, the <build directory> is <design name>\<design name>.build\<design name>_impl_<date>. Due to file path length limitations in the Windows filesystem, a temporary drive letter is assigned to <build directory> during the PathWave FPGA build. The default is Z: unless it is already being used in which case another drive letter is chosen. In some build files you may see reference to this drive letter. It means the same as <build directory>.

The ports of the sandbox can be determined by inspecting the top level HDL block after PathWave FPGA does a synthesis. For a design called <design name>, this file can be found in <build directory>\sources\<design name>.vhd. For example, a design called "FreqCntTest" might have the top level HDL file called FreqCntTest\FreqCntTest.build\FreqCntTest_impl_2019-12-13_16_32_17\sources\FreqCntTest.vhd. The ports can also be determined by looking at the IP-XACT that describes the sandbox. This can be found in the BSP install directory, e.g. \Program Files\Keysight\M3202A BSP\R037300\bsp\templates\interfaces\M3202A\M3202A_ch4.2.0.xml.

Not all the interfaces for the sandbox need be simulated for all designs. For example, if the sandbox to be tested does not use the DDR interface, there is no need to provide an interface model for the DDR interface. However, the ports must still be declared and inputs tied off to avoid simulator errors.

Simulating PathWave FPGA Standard Interfaces

One thing a sandbox testbench needs to do is simulate the various interfaces between the sandbox and the static region. Some interfaces, such as the clock, reset, wire, or vector, are simple and easily handled. Other interfaces, especially ones with handshaking, can be more complicated to simulate correctly.

Simulating AXI/AXI-lite Interfaces

The AXI4 interface is an industry standard, high performance addressable bus architecture specified in the *AMBA AXI and ACI Protocol Specification*. The AXI4-Lite interface is a lighter weight, more limited subset of the full AXI4 protocol. The full AXI4 supports things such as burst transfers while the AXI4-Lite is limited to single word transactions. These can be used for random access to things like memories or control registers. Typical uses for an AXI interface in a BSP include Host interfaces and DDR interfaces. The Host interface would be an AXI

interface where the AXI Master is in the static region and controlled by the host processor. The AXI Slave is in the sandbox and may include things like register banks or memory maps for accessing and controlling sandbox IP. Some BSP may also include large, external memory (DDR memory) accessible from the sandbox. In this case, the AXI Master is in the sandbox with the AXI Slave in the static region.

Simulating a complete AXI interface can be complex. Fortunately, Xilinx provides IP that greatly simplifies simulating AXI interfaces. The Vivado Design Suite includes *AXI Verification IP*. This IP is simulation only (non synthesizable) System Verilog code that interfaces with the sandbox's AXI interface and allows reading and writing. This IP is described in the LogiCORE IP Product Guide PG267. This IP can be configured for various roles including AXI Master, AXI Slave, AXI-Lite Master, and AXI-Lite Slave. Furthermore, if the IP is configured as a Slave, it can optionally include a memory model. This makes simulating something like a DDR interface simpler as the testbench writer does not need to write their own memory model. The AXI Verification IP's memory model includes mechanisms for "back door" accessing the memory array. This allows for preloading data into the memory as well as reading it out.

A description of how to use the AXI Verification IP is beyond the scope of this document. Please see the AXI Verification IP Product Guide as well as Vivado's Example Design (which shows how to use the IP).

Simulating AXI Streaming Interfaces

The AXI4-Streaming interface is an industry standard non-addressable bus architecture designed for both streaming arbitrarily long sequences of data and for streaming finite sized data records. The AXI4-Streaming interface is specified in the *AMBA 4 AXI4-Stream Protocol Specification*. Implementations of the AXI4-Streaming interface can be as simple as data and a dataValid, or it can be more complex with reverse flow control and optional data fields. An AXI4-Streaming interface can be used for sending occasional control words for controlling sandbox IP blocks or for continuous high speed streams of data such as from an ADC.

Xilinx provides IP for simulating AXI-Streams. This IP is called *AXI4-Stream Verification IP* and is similar to the AXI Verification IP mentioned above. It is described in the LogiCORE IP Product Guide PG277.

While this IP is full featured, it can be complicated to use. For simple AXI4-Streaming interfaces, it may be easier to manually drive the interface signals directly. As an example, consider the M3100A PXIe Digitizer. The Analog Channel Control includes the Analog_Trigger interface which is a simple AXI4-Streaming interface. This interface sets the trigger threshold value and trigger mode for the sandbox's trigger detection logic. It includes 16 bits of threshold value and 2 bits of mode along with a valid signal. Simple testbench HDL code to drive this interface might look like:

```
task writeAnalogTrigger1;
    input [15:0] threshold;
    input [1:0] mode;
    begin
        ain1_analog_trig_tdata <= {6'b0,mode,threshold};
        ain1_analog_trig_tvalid <= 1'b1;
        @(posedge clk) ain1_analog_trig_tvalid <= 1'b0;
    end
endtask
```

Then to change the threshold, the testbench might simply:

```
writeAnalogTrigger1(thresholdValue,modeValue);
```

Continuously streaming data, such as from an ADC, is also easy to simulate. For example, if the testbench wanted to simulate an ADC that is digitizing a sine wave, the testbench might look like:


```

integer timeCount = 0;
real Freq = 0.01;
real Scale = 32767;
...
ainl_out_tvalid <= 1'b1;
always @(posedge clk) begin
    ainl_out_tdata <= Scale * $sin(2.0 * 3.14159 * Freq * timeCount);
    timeCount <= timeCount + 1;
end

```

Simulating Mem Interfaces

Mem Version 1

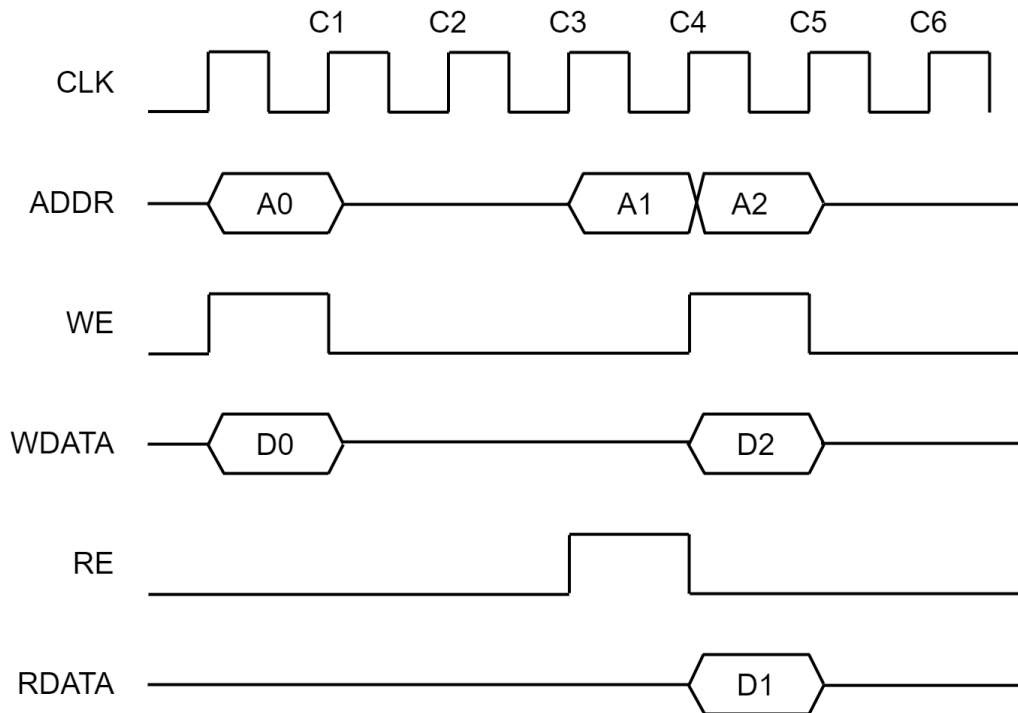
The mem Interface is a simple, time deterministic, addressable interface using 32 bit wide data words. Note that while the AXI4 interface uses byte addressing, the mem interface uses word addressing.

The mem interface has five ports:

Addr[n-1:0]	The address for a read or write transaction. Note: this is a word address, not a byte address.
Wdata[31:0]	Write data which is always 32 bits wide and should be valid when We is asserted.
We	Write enable. Indicates a write transaction.
Rdata[31:0]	Read data which is always 32 bits wide and should be valid the clock cycle after Re is asserted.
Re	Read enable. Indicates a read transaction.

A write transaction occurs when We is asserted. At that time the data on Wdata is written to the address specified by Addr.

A read transaction occurs when Re is asserted. The slave should return the data specified by Addr on the next clock cycle.



This diagram shows a mem write followed by a mem read. When WE is asserted at clock C1, the value D0 gets written to address A0. At clock C4 there is a mem read at address A1. Note that the slave device has to drive D1 on the very next clock from when RE is asserted at C5 (as shown above).

Transactions may be initiated on any clock cycle. There is no need to wait for a read transaction to complete before initiating the next transaction. Thus there may be two reads on adjacent clock cycles, two writes, or a read followed by a write or a write followed by a read on adjacent clock cycles. It is okay to have a read at C4 immediately followed by a write at C5 as shown above.

Mem Version 2

Version 2 of the mem interface is very similar to version 1 with the difference being that version 1 requires mem slaves to have a read latency of one clock cycle (i.e. they have to return read data the very next clock after RE), version 2 allows slaves to have longer read latencies. These latencies must still be fixed for each mem slave, but each different slave can have a different, fixed latency. For example, a mem slave that is a register might have a read latency of 1 (the same as the version 1 case) while a mem slave that is a block memory might have a read latency of 2. If a mem slave has a read latency other than 1, it must report its latency to PathWave FPGA.

Simulating Mem+ Interfaces

The mem+ interface is an extension of the mem Version 2 interface. It is only used between the static region and the sandbox. It is not exposed to users of the sandbox (they only see the mem interface). A simulation of the sandbox, though, needs to use the mem+ interface for BSPs that require it.

The mem+ interface is the mem Version 2 interface with two added signals, RDATAVALID and ERROR. When a sandbox user uses more than one mem slave, PathWave FPGA will automatically generate the logic necessary to split the single mem+ interface from the static region to the necessary number of separate mem interfaces in the sandbox. Note that although each mem interface has a fixed latency, since the mem+ interface may connect to multiple different mem slaves, the mem+ read latency can vary depending on which mem slave is being read from.

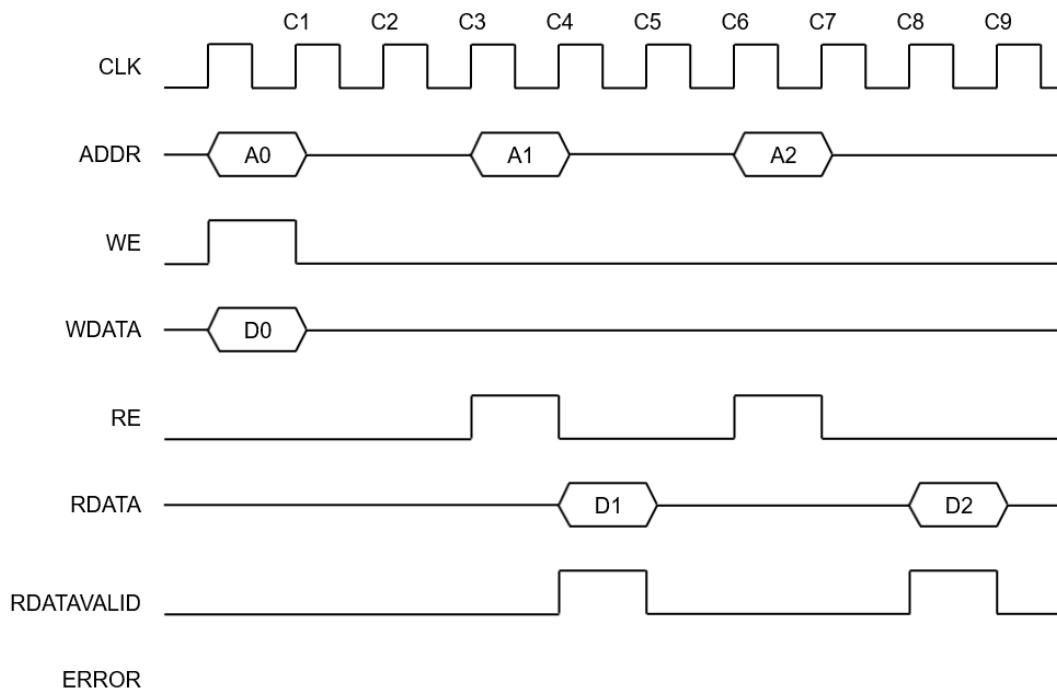
Note that because of the varying latency, there is the possibility of read data collisions. For example, reading from a mem slave with read latency of 3 immediately followed by a read from

a mem slave with read latency 2 would result in both read data values being returned on the same clock cycle. This is known as a collision and is an error. It is up to the test bench designer to ensure that reads are scheduled to avoid collisions.

The mem+ interface has seven ports:

Addr[n-1:0]	The address for a read or write transaction. Note: this is a word address, not a byte address.
Wdata[31:0]	Write data which is always 32 bits wide and should be valid when We is asserted.
We	Write enable. Indicates a write transaction.
Rdata[31:0]	Read data which is always 32 bits wide and should be valid the appropriate latency after Re is asserted.
Re	Read enable. Indicates a read transaction.
RdataValid	Indicates that Rdata is valid. Since the read latency is known ahead of time, this signal is for informational use only.
Error	Indicates a collision in the mem routing logic.

For a sandbox that uses the mem+ interface, the AddressMapping.json generated during the FPGA build process will include an element called *readLatency* that will specify the read latency for that particular mem slave. This latency will be the sum of the slave's read latency plus any additional latency added by the routing logic generated by PathWave FPGA. This value indicates how many clock cycles after Re the returned Rdata should arrive at the sandbox (testbench) boundary.

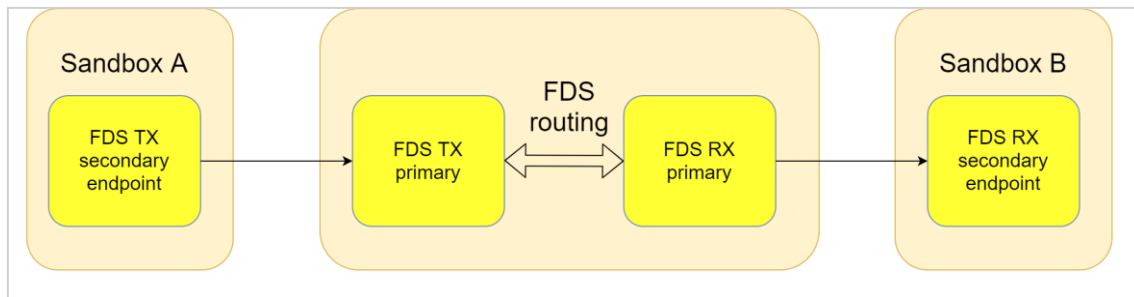


In this example, the mem slave at address A1 has a *readLatency* of 1 while the slave at address A2 has a *readLatency* of 2. The read at time C4 expects the read data to be returned one clock cycle later at time C5. The read at time C7 doesn't expect the read data to be returned until time C9 due to the increased read latency.

Note that if the test bench tried reading A2 at time C_n and reading A1 at time C_{n+1} , then there would be a collision at time C_{n+2} since the data from both slaves would be trying to arrive at the sandbox boundary at the same time. This would result in Error being asserted.

Simulating FDS Interfaces

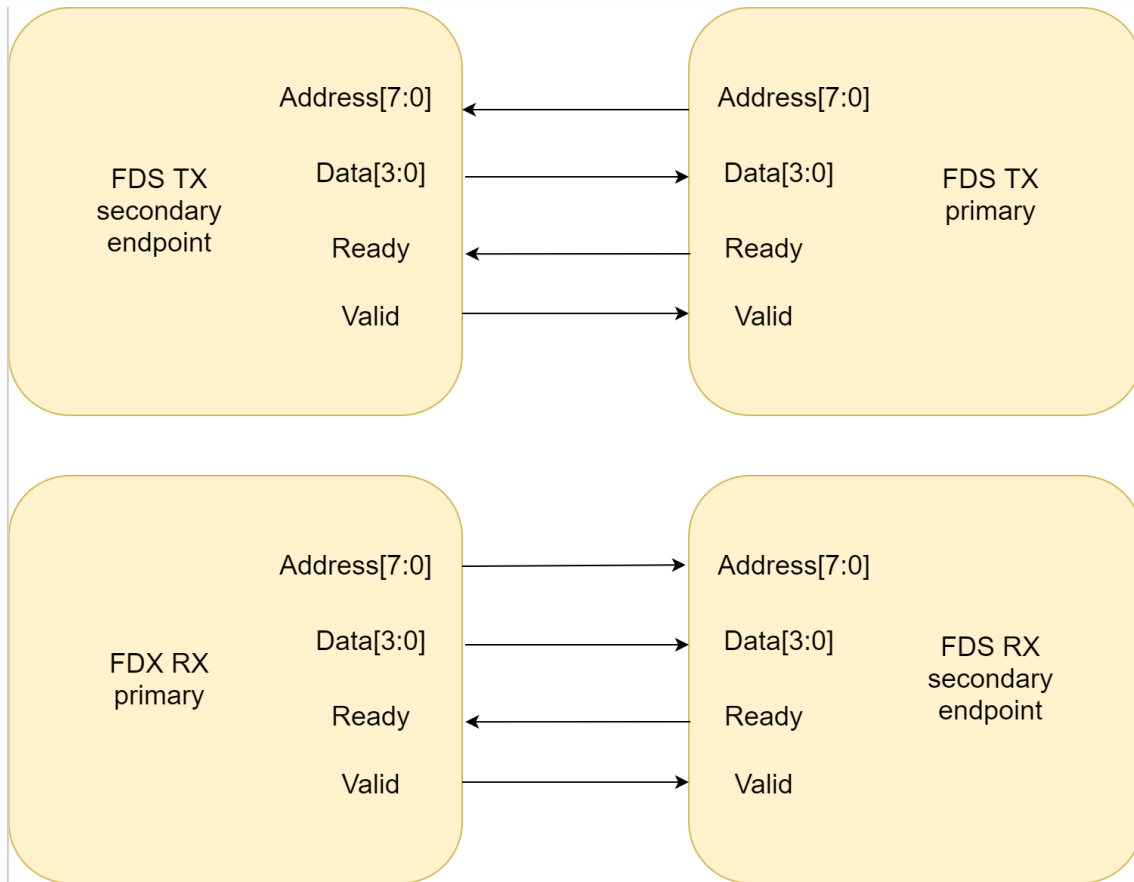
The FDS, or Fast Data Sharing, interface comes in two types: FDS_RX for the receive side and FDS_TX for the transmit side. The FDS interface uses an 8 bit address and a 4 bit (nibble) data bus. Multiple accesses at a single address may be used for wider data transfers. The FDS interface has deterministic timing and does not include back-pressure. Below is a diagram that shows the different FDS endpoints and how they are connected together. Each sandbox has an FDS secondary endpoint. This allows a separate module with FDS primary interfaces to read/write data into the sandboxes. In the diagram below data flows from left to right (data starts at the FDS TX secondary endpoint and is sent to the FDS primary side. FDS routing sends the data to the correct FDS RX primary side and then the data is passed to another sandbox FDS RX endpoint). The diagram below is simplified to just show one source and one destination sandbox, however there can be multiple sandboxes with TX endpoints or RX endpoints and the FDS data routing will be handled by a control module. It is not possible to connect an FDS TX secondary endpoint directly to a FDS RX secondary endpoint. A control module with the FDS primary interfaces must be used to route the data between the secondary endpoints.



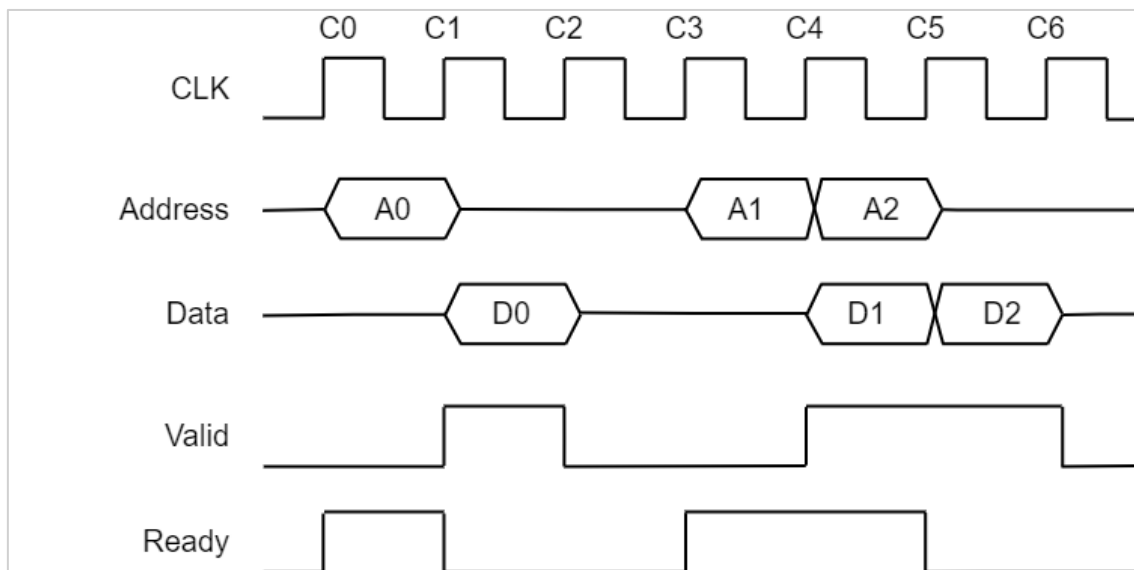
The FDS interface has four ports:

Address[7:0]	The address for a read or write transaction.
Data[3:0]	Data is 4 bits wide and should be valid when Valid is asserted.
Valid	Asserted when there is valid data on the Data port.
Ready	Asserted by data recipient when it is ready to accept data.

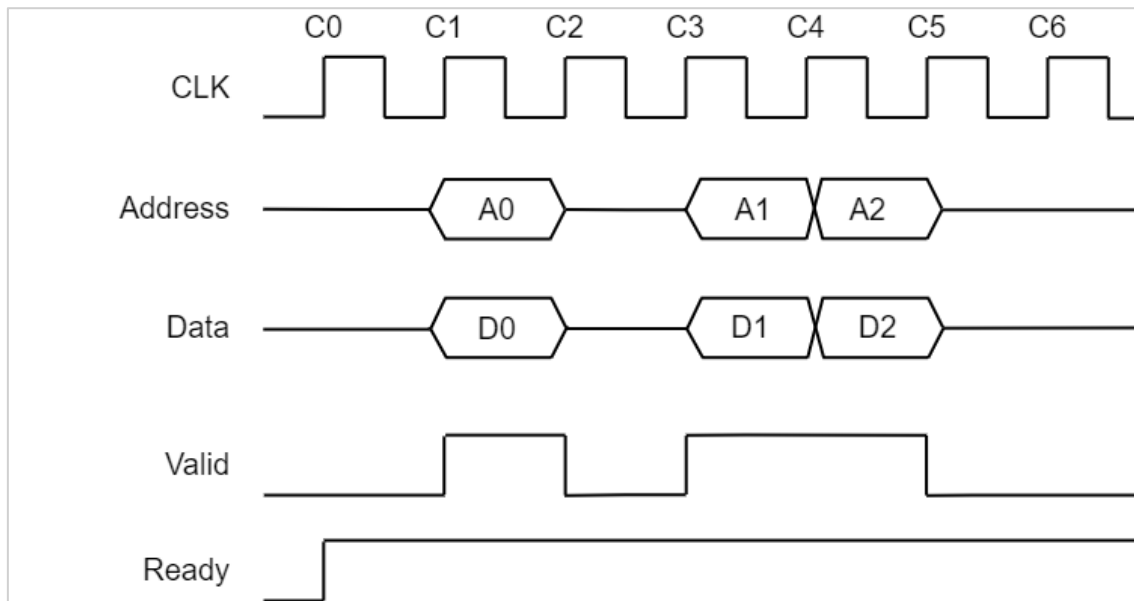
Below is a diagram showing the port directions for the primary and secondary sides of each TX and RX interface. The secondary endpoints are the FDS interface on a custom IP block in the sandbox and the primary sides are FDS interfaces in a control module.



The timing diagram below shows a single FDS TX transaction followed by two FDS TX transactions. A TX transaction occurs when Ready coming from the FDS TX primary side is asserted. Then the TX secondary endpoint is required to assert Valid and drive data on the next clock cycle. If the TX secondary endpoint does not assert Valid on the next clock cycle then it is considered an error. In the timing diagram below Ready and Address get asserted at C0 by the FDS TX primary side indicating the primary side is ready to read data at address A0. On the next clock cycle C1 the FDS TX secondary endpoint on the IP block in the sandbox asserts Valid and returns D0 on the Data port. This repeats at clock cycles C3 through C5 for two TX transactions.



The timing diagram below shows a single FDS RX transaction followed by two FDS RX transactions. An RX transaction occurs when Valid coming from the FDS RX primary side is asserted. Ready from the RX secondary endpoint is required to be asserted when Valid is asserted from the FDS RX primary side, otherwise it is considered an error. Practically, this means Ready should be asserted whenever there is a chance of an FDS transaction. In the timing diagram below the FDS RX secondary endpoint in the sandbox IP block asserts Ready at clock cycle C0. Then at clock cycle C1 the FDS RX primary side sets Address to A0, Data to D0, and asserts Valid. This process gets repeated during clock cycles C3 and C4 for two RX transactions.



There may be times when one wants to send data that is wider than 4 bits. To do this the data must be split into multiple 4-bit transactions at the same address. Data should be sent the least significant nibble first. For example, if a register at address 0 is 32 bits wide and a register at address 1 is also 32 bits wide, setting address to 0 and writing 8 nibbles to this address will write all 32 bits of this register starting with the least significant nibble. Setting the address to 1 and reading 8 nibbles from this address will read all 32 bits from this register starting with the least significant nibble.

Note: Both endpoints of a transaction must agree on the size of the data being transferred.

Simulation Fileset

Simulating the sandbox requires compiling all the source files for the test bench and any IP used by the test bench (e.g. the AXI-VIP files), and also all the HDL files used in the sandbox itself. The IP used in the sandbox includes the IP shown on the design canvas as well as IP connecting the user's design to the sandbox ports. The list of all the source files needed can be found in the sources.json file. This file is in the same build directory as the AddressMapping.json file.

sources.json

The sources.json file lists the sources used to build the sandbox design.

Due to file path length restrictions in the Windows operating system, the build directory is mapped to a temporary drive letter, usually Z: unless that drive is already in use. This drive denotes the build directory <design name>\<design name>.build\<design name>_impl_<date>.

The "ip" section lists the Vivado IP used in the design. These are xci files that describe the particular IP. During the synthesis process, Vivado is used to regenerate the IP for this particular design. Vivado also regenerates simulation netlists that can be used for simulation for these IP. For an IP block named <ipBlock> these simulation files can be found in:

```
<build directory>\<design name>_Synthesis\<design
name>.srcs\sources_1\ip\<ipBlock>\<ipBlock>_sim_netlist.v or
<build directory>\<design name>_Synthesis\<design
name>.srcs\sources_1\ip\<ipBlock>\<ipBlock>_sim_netlist.vhdl
```

One of these should be compiled for simulation.

The "sources" section lists the HDL files needed as well as which library they could be compiled into. It may also list a ".bd" file. This is a Vivado Board Design file used in IP Integrator. For a block called <bdName> the information needed to simulate the block can be found in:

```
<build
directory>\<bdName>\<bdName>.ip_user_files\sim_scripts\<bdName>\<simul
ator>
```

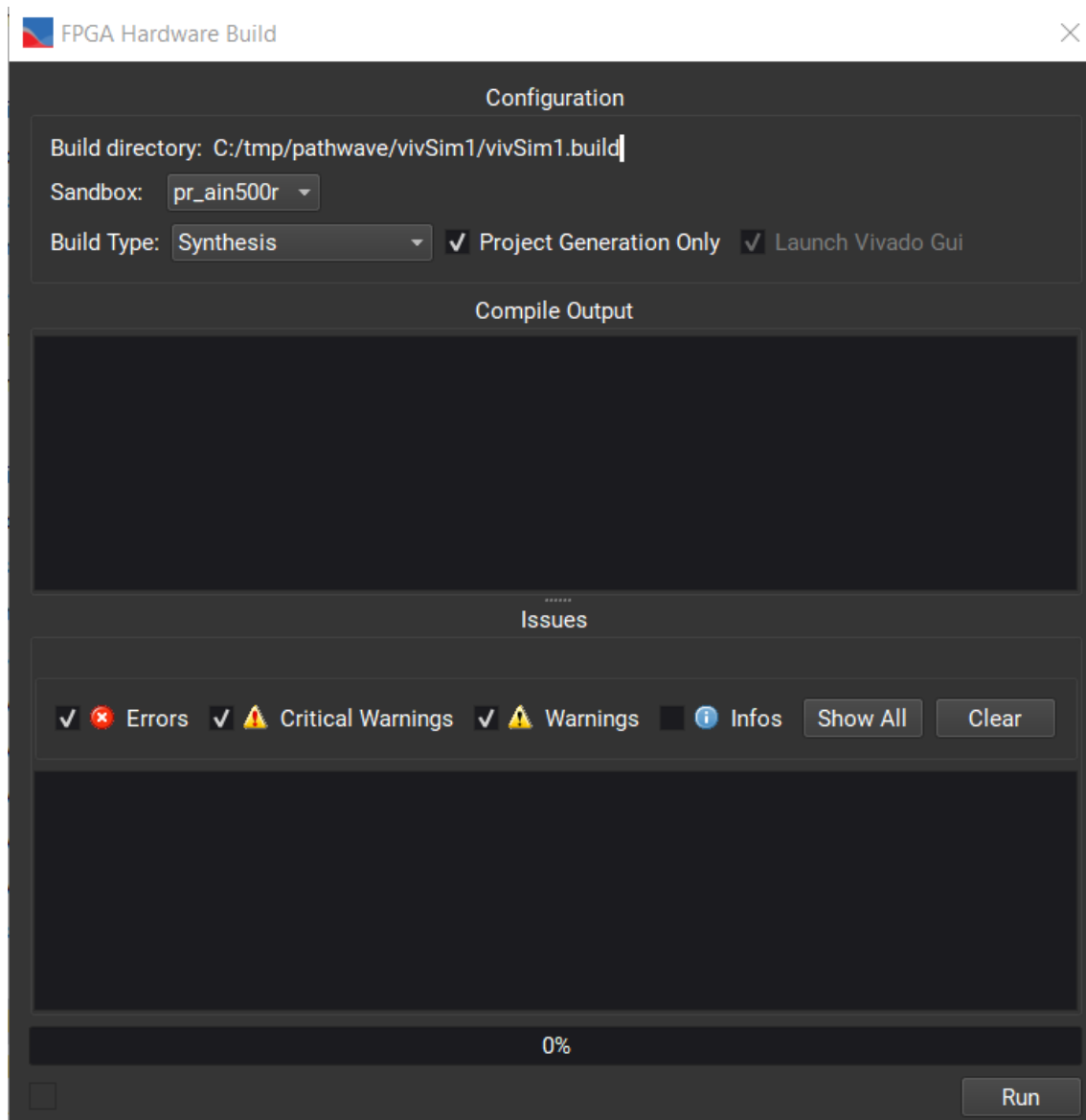
where <simulator> is the name of the simulator to be used, e.g. "questa". In that directory are scripts to compile the design as well as a README.txt describing how to use the files.

Using Vivado's Simulation Flow

Managing the files used in the sandbox design can be simplified by using Vivado's simulation flow. In this case the source files used within the sandbox are automatically sent to Vivado. The user still needs to manually add any source files needed for the testbench separately. The user still needs to write the top level testbench file that instantiates the sandbox design.

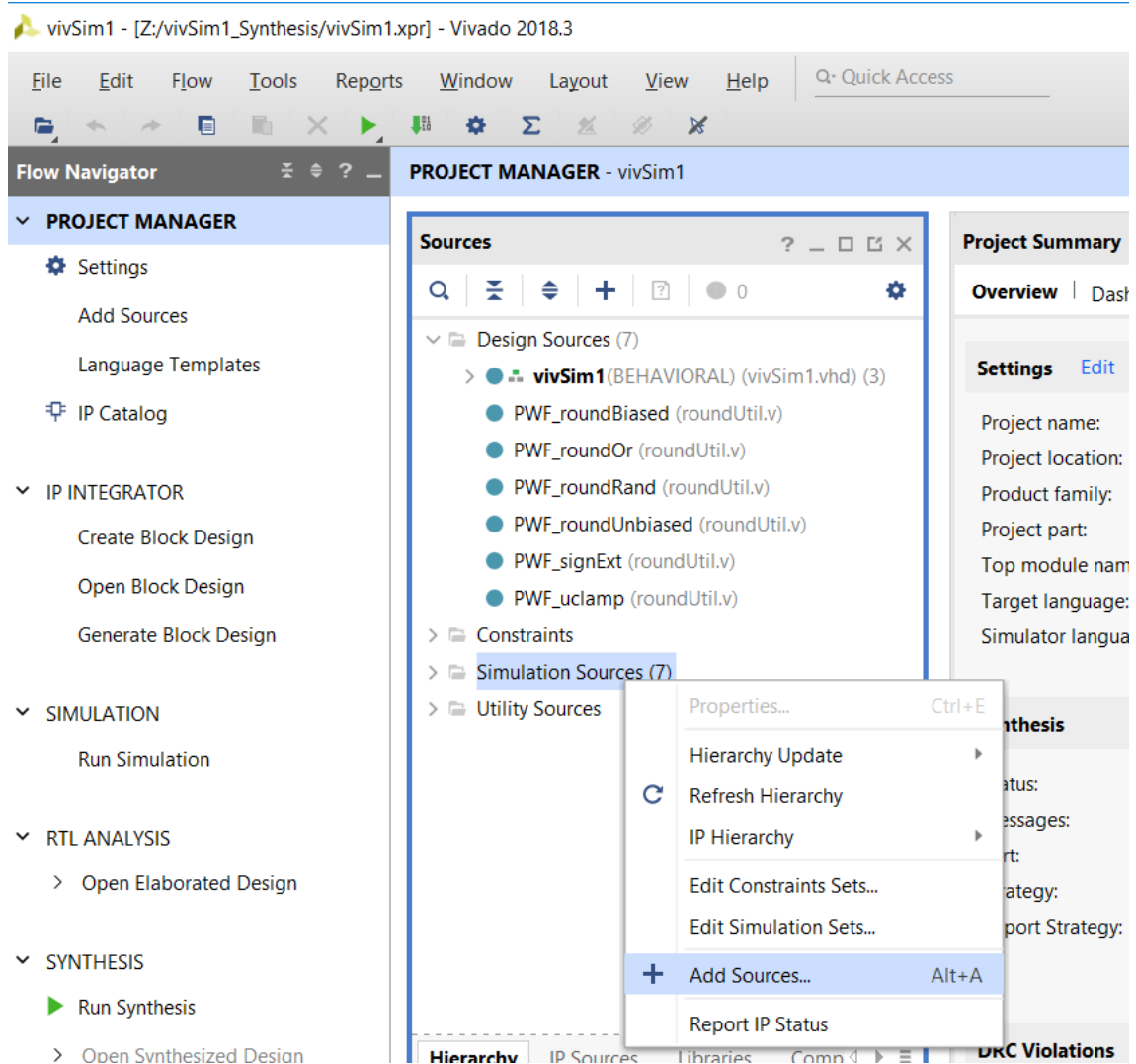
To use this flow, follow these steps:

In PathWave FPGA click on *Generate Bit File...* to bring up the FPGA Hardware Build dialog. Check the *Project Generation Only* (which will also automatically check the *Launch Vivado Gui* box) and hit run:

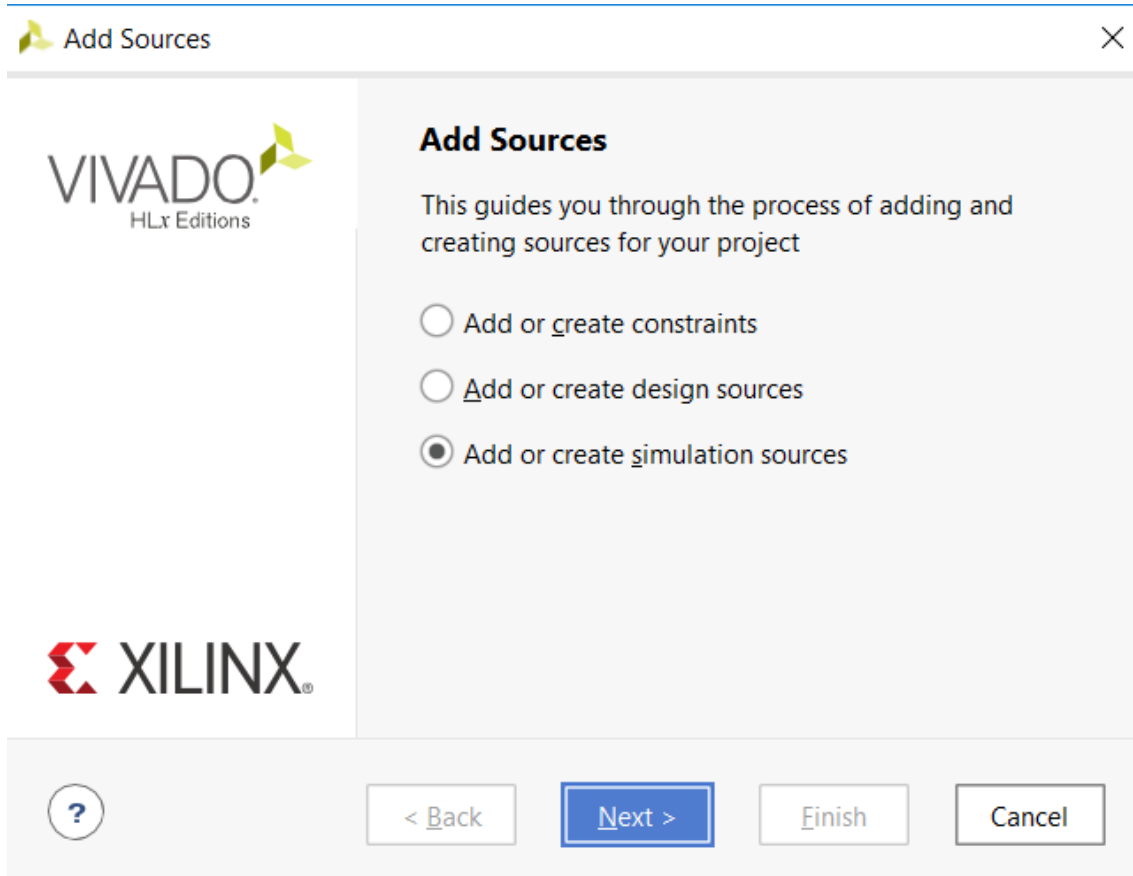


This will generate the sandbox project, create a Vivado Project for this design, and start the Vivado Gui.

Once Vivado is started, right-click on *Simulation Sources* and select *Add Sources...* to bring up the *Add Sources* dialog:

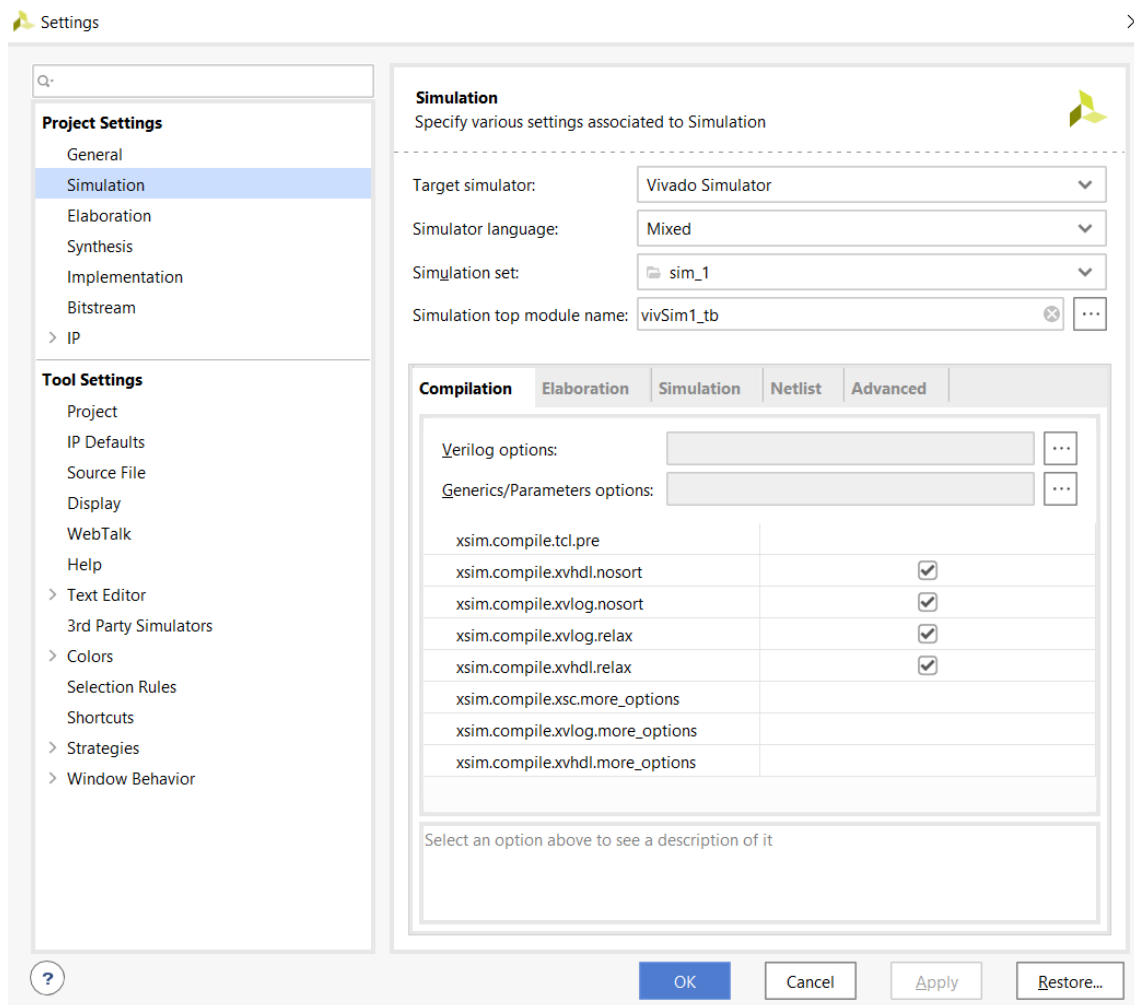


Select *Add or create simulation sources* and click *Next*.

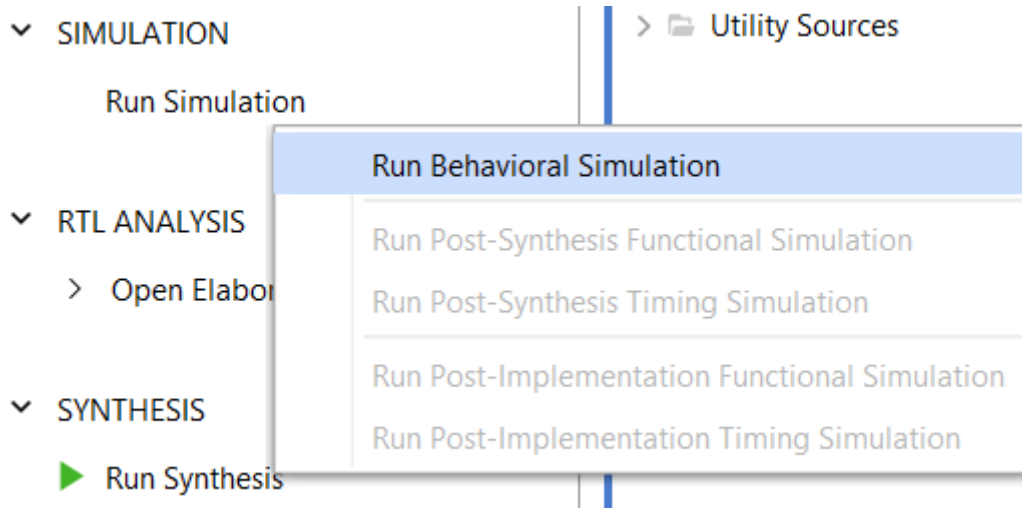


Add the necessary sources files for the testbench and click *Finish*. Note that every time PathWave FPGA does another build, it creates a new Vivado project. This new Vivado project will not have the simulation files you may have added to the previous Vivado project, so the simulation source files have to be added every time a new PathWave FPGA build is run.

Open the *Simulation Settings* dialog and verify that the correct *Simulation top module name:* is selected:



Click the *Simulation*→*Run Simulation*→ *Run Behavioral Simulation* to start the simulator:



Test Bench Address Mapping

During the FPGA build process, PathWave FPGA will automatically assign addresses for things like Memory Maps and Register Banks. When a design is modified, the assigned address for a particular entity may change. When writing test benches it is recommended to reference entities symbolically rather than with hard coded address values. Then if the address to an

entity changes due to a later design modification, the test bench will remain accurate whereas a hard coded address may have to be manually modified if the design changes.

To facilitate symbolic references to entities, PathWave FPGA generates a number of address mapping files during the FPGA build process. These files may be used by test benches to map symbolic names to actual addresses. These files are placed in the build directory in the sources sub-directory. PathWave FPGA creates a Verilog include file and a VHDL package file. The files are named as follows

```
<ProjectName>.build\<ProjectName>_<BuildDate>\sources\<ProjectName>_addressMapping.vh (for Verilog)
<ProjectName>.build\<ProjectName>_<BuildDate>\sources\<ProjectName>_addressMapping.vhd (for VHDL)
```

```
For a project called "simTest" these files might be:
simTest.build\simTest_synth_2019-12-19_11_53_45\sources\simTest_addressMapping.vh
simTest.build\simTest_synth_2019-12-19_11_53_45\sources\simTest_addressMapping.vhd
```

These files contain the same information as the AddressMapping.json file, but in a format suitable for HDL.

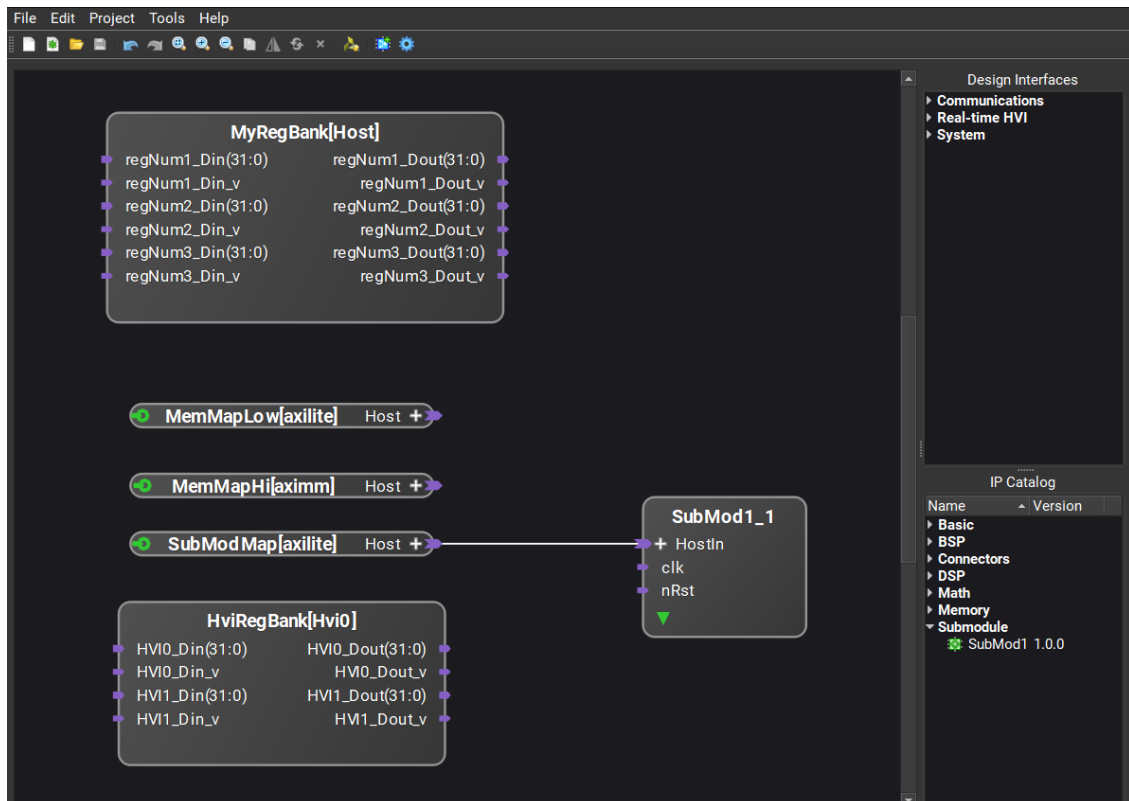
Contents of Address Mapping Files

The Verilog address mapping file is meant to be included into the Verilog test bench via a ``include` statement. It defines localparams for each entity. The VHDL file is meant to be compiled separately and defines a package consisting of integer constants for each entity.

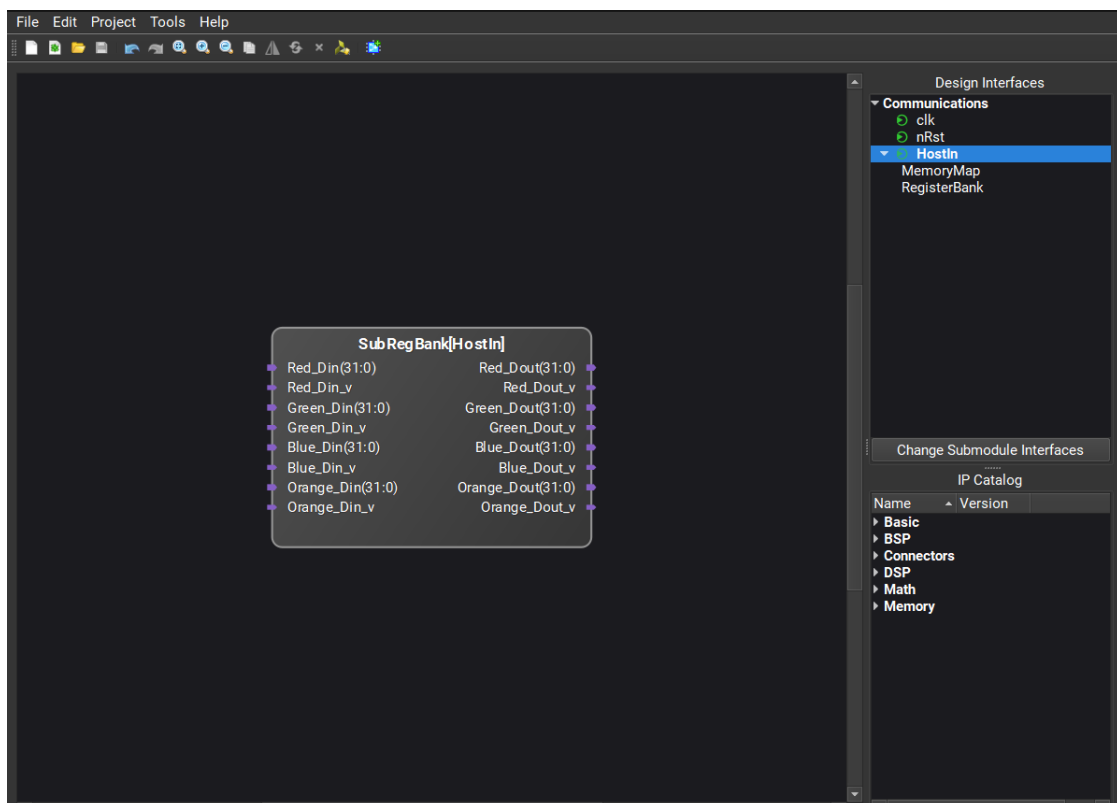
The name for each entry is formed from a hierarchical concatenation of names separated by the underscore ("_") character. For a MemoryMap, this is just the <MemoryMap name>. For a RegisterBank, this is the <RegisterBank name>_<Register name>. For entities in submodules, the name is <Submodule name>_<entity name within the submodule>.

Address Mapping Example

Consider the following design that includes RegisterBanks on both the Host and HVI interfaces, three MemoryMaps, and a submodule:



The submodule only contains another RegisterBank:



The resulting Verilog file is:

```

// This file was automatically generated by PathWave FPGA. PLEASE DO NOT
// EDIT IT.
//
// simTest_addressMapping.vh
`ifndef _simTest_addressMapping_vh_
`define _simTest_addressMapping_vh_

// Interface: Host
localparam MemMapLow = 'h0;
localparam SubModMap = 'h4000;
localparam MemMapHi = 'h6000;
localparam MyRegBank_regNum1 = 'h7000;
localparam MyRegBank_regNum2 = 'h7004;
localparam MyRegBank_regNum3 = 'h7008;

// Interface: Hvi0
localparam HviRegBank_HVI0 = 'h0;
localparam HviRegBank_HVI1 = 'h1;

// Interface: SubMod1_1.HostIn
localparam SubMod1_1_SubRegBank_Red = 'h4000;
localparam SubMod1_1_SubRegBank_Green = 'h4004;
localparam SubMod1_1_SubRegBank_Blue = 'h4008;
localparam SubMod1_1_SubRegBank_Orange = 'h400c;

`endif // _simTest_addressMapping_vh_

```

and the resulting VHDL file is:

```

-- This file was automatically generated by PathWave FPGA. PLEASE DO NOT
-- EDIT IT.

-- simTest_addressMapping.vhd
package simTest_addressMapping is

    -- Interface: Host
    constant MemMapLow : integer := 16#0#;
    constant SubModMap : integer := 16#4000#;
    constant MemMapHi : integer := 16#6000#;
    constant MyRegBank_regNum1 : integer := 16#7000#;
    constant MyRegBank_regNum2 : integer := 16#7004#;
    constant MyRegBank_regNum3 : integer := 16#7008#;

    -- Interface: Hvi0
    constant HviRegBank_HVI0 : integer := 16#0#;
    constant HviRegBank_HVI1 : integer := 16#1#;

    -- Interface: SubMod1_1.HostIn
    constant SubMod1_1_SubRegBank_Red : integer := 16#4000#;
    constant SubMod1_1_SubRegBank_Green : integer := 16#4004#;
    constant SubMod1_1_SubRegBank_Blue : integer := 16#4008#;
    constant SubMod1_1_SubRegBank_Orange : integer := 16#400c#;

end package simTest_addressMapping;

```

In both cases there are entries for each MemoryMap and each register within a RegisterBank.

Using the Address Mapping File

Verilog

To use the address mapping file, the file is included into the Verilog test bench with the ``include` statement. Then the various address mapping values can be used within the test bench. As an example:

```
module simTest_tb;
...
`include "simTest_addressMapping.vh"
...
    writeBus(MyRegBank_regNum2,32'h12345678); // write 0x12345678 to
register regNum2
...
task writeBus; // task to write a value to an Axi bus
input [31:0] address;
input [31:0] data;
...
endtask
endmodule
```

This example assumes there is a Verilog task called "writeBus" that will handle the necessary handshaking to write a value to the AXI bus. When calling writeBus, the symbolic name of the MyRegBank's regNum2 is used instead of the numeric address 0x7004.

VHDL

To use the address mapping file, the file is compiled into a library (the default being the library "work"). Then the constants can be used in other VHDL code. As an example:

```
use work.simTest_addressMapping.all;
...
architecture behavior of simTest_tb is
    signal addr_reg2 : std_logic_vector(31 downto 0);
...
    addr_reg2 <=
conv_std_logic_vector(work.simTest_addressMapping.MyRegBank_regNum2,
addr_reg2'length);
...
end architecture
```

In this partial example, a `std_logic_vector` for the address of MyRegBank's regNum2 is created and assigned and can be used wherever regNum2's address is needed.

Name Collisions

Since the entry names are created in a hierarchical manner, it is possible to get name collisions in certain cases. As an example, both a MemoryMap named "foo_bar" and a RegisterBank named "foo" with a register named "bar" would result in the same name "foo_bar". If a name collision is detected at build time, a Critical Warning is issued, and the duplicated name entry in the address mapping files is commented out along with a comment indicating the name collision.

If a name collision Critical Warning occurs, at least one of the names should be changed to prevent the name collision from occurring.

Advanced Features

- [Command Line Arguments](#)
- [Migrating a design to a new BSP](#)
- [Changing a Submodule Project Target Hardware](#)
- [Debugging in Hardware](#)
- [User Constraint Files](#)

Command Line Arguments

When PathWave FPGA is launched from a command line or script, there are a number of arguments to create or load projects, and control how the application operates.

```
Usage: PathWave_FPGA [--project/-p/<no_switch> <ProjectFile (*.kfdk)>]
[--bsp/-b <BspName>] [--version/-v <BspVersion>] [--template/-t
<TemplateName>] [-c <OptionName> <OptionValue>] [--retarget/-r
<ExistingProjectFile>] [--generate/-g <generationType>]
```

<no_switch> or -p [--project]	Path to project file to open or create (*.kfdk)
-b [--bsp]	Name of the BSP
-v [--version]	Version of the BSP
-t [--template]	Name of the BSP template to use
-r [--retarget]	Path to existing project (*.kfdk) to retarget to different BSP configuration
-c	Name/Value configuration option pairs for the specified BSP, separated by space
-g [--generate]	Type of generation: synthesis, implementation
-h [--help]	Print usage message

- For creating a new project, the <ProjectFile> and <BspName> arguments are required. The rest of the BSP options are needed only to distinguish different configurations of the same BSP.
- If there is no BSP matching the provided <BspName>, a list of available BSP names is displayed.
- If there are more than one configurations that match the provided arguments, or no configuration that matches them, a list of available configurations is displayed.
- If the '--generate' option is used, the application will close automatically after the completion of the generation build.

- The project path can be specified without any switch. However, in that case, it should not be specified after the '-c' switch arguments, as it will be translated, erroneously, as a configuration option
- The '--retarget' and '--template' switches cannot be used together

Examples

- Start GUI:

```
PathWave_FPGA
```

- Open project:

```
PathWave_FPGA path/to/myExistingProject.kfdk
```

- Open project and implement it (application will close automatically after the completion of the build):

```
PathWave_FPGA path/to/myExistingProject.kfdk -g implementation
```

- Create a new project from template and open it:

```
PathWave_FPGA path/to/newProject.kfdk --bsp M3202A -v 03.67.00 -c channels 2 -c fpga 7k325 -c clock Variable --template Default
```

- Create a new project from template and synthesize it (application will close automatically after the completion of the build):

```
PathWave_FPGA path/to/newProject.kfdk --bsp M3202A -c channels 2 -c fpga 7k325 -c clock Variable --template Default -g synthesis
```

- Retarget an existing project to different BSP configuration:

```
PathWave_FPGA path/to/newProject.kfdk --bsp M3202A -c channels 4 -c fpga 7k410 -c clock Variable --retarget path/to/existingPrj.kfdk
```

Migrating a design to a new BSP

This topic lists the steps to retarget an existing hardware project to a different BSP.

1. Select **File > Retarget Project**.
2. Select an existing PathWave FPGA Project File. Click **Next**.
 - a. If you begin retargeting while a project is open, the existing project will be selected.
3. Choose the **Board Support Package** for the target hardware module and click **Next**.
 - a. If multiple board options are available, select the configuration of the BSP you want to use.
4. A summary of the project details is displayed. Click **Finish**.
5. A dialog will appear informing you of a project version change.
 - a. A backup of your original file is created at this time.
6. The retargeted project will open, and any IP blocks that are now invalid with the retargeted project will have a red 'x'.

Command Line

You can also retarget your project using the command line, for more details see [Command Line Arguments](#).

Changing a Submodule Project Target Hardware

When a submodule is created, the target hardware for that submodule is inherited from the parent sandbox or submodule.

You may want to retarget a submodule to work with different hardware, or remove the targeted hardware altogether to make a generic submodule. A generic submodule can be shared with projects targeting different BSPs, but will not have access to the BSP IP.

Perform the following steps to change the submodule target hardware:

1. With the submodule project open in PathWave FPGA, select **Project > Properties...**
2. To change the **Target Hardware** to a new BSP, click **Change** and use the **Select BSP Configuration** wizard to choose a new BSP.
3. To remove the BSP and create a generic submodule, click **Clear**.
4. Click **Apply** to accept the changes.

Debugging in Hardware

PathWave FPGA supports the embedding of Vivado debug cores for use with the Vivado Logic Analyzer for debugging Sandbox designs in hardware, provided the following prerequisites are met:

- The targeted BSP must support hardware debugging.
- Must have a supported debug interface cable (JTAG download cable), or the BSP must support one of the "Virtual Cable" methods (PCIe, Ethernet).

Check the targeted BSP's documentation to confirm whether hardware debugging is supported and which connection methods may be used.

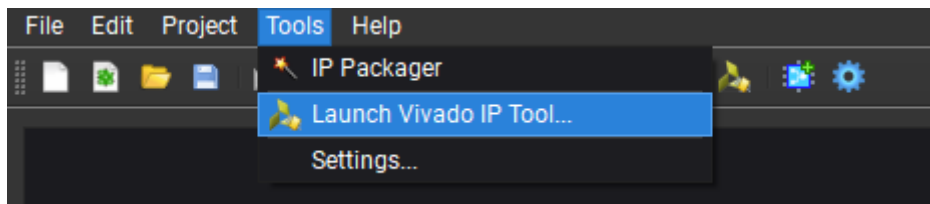
To debug a PathWave FPGA Sandbox design in hardware, simply use the PathWave FPGA Launch Vivado IP Tool feature to customize a Vivado IP debug core, instantiate the debug core in the Sandbox, make the necessary probe/trigger connections, and build the bit file. PathWave FPGA and the Vivado implementation tools take care of the rest. After the bit file is generated and loaded, the Vivado Logic Analyzer can be used for debugging Sandbox designs in hardware.

To provide an example of how to use the Vivado Logic Analyzer for a PathWave FPGA Sandbox design, consider a simple Sandbox design with an 8-bit counter.

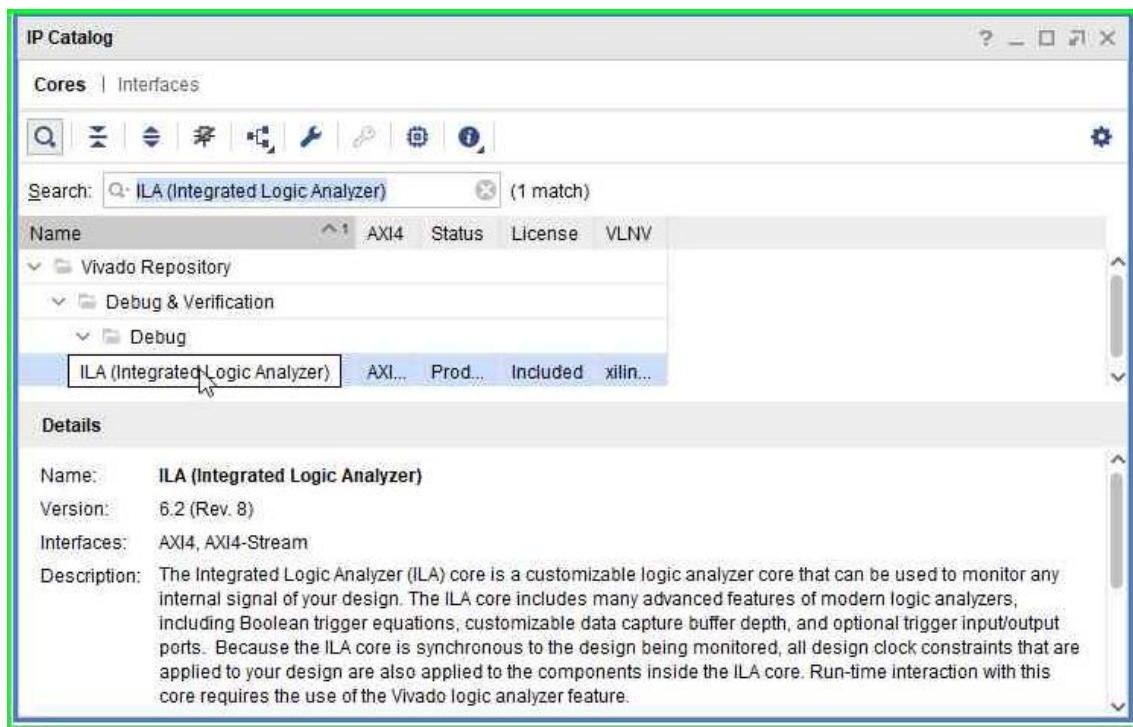


The counter was customized from the Vivado IP Catalog, imported into PathWave FPGA, instantiated in the design, and then connected to the Design Interface clock.

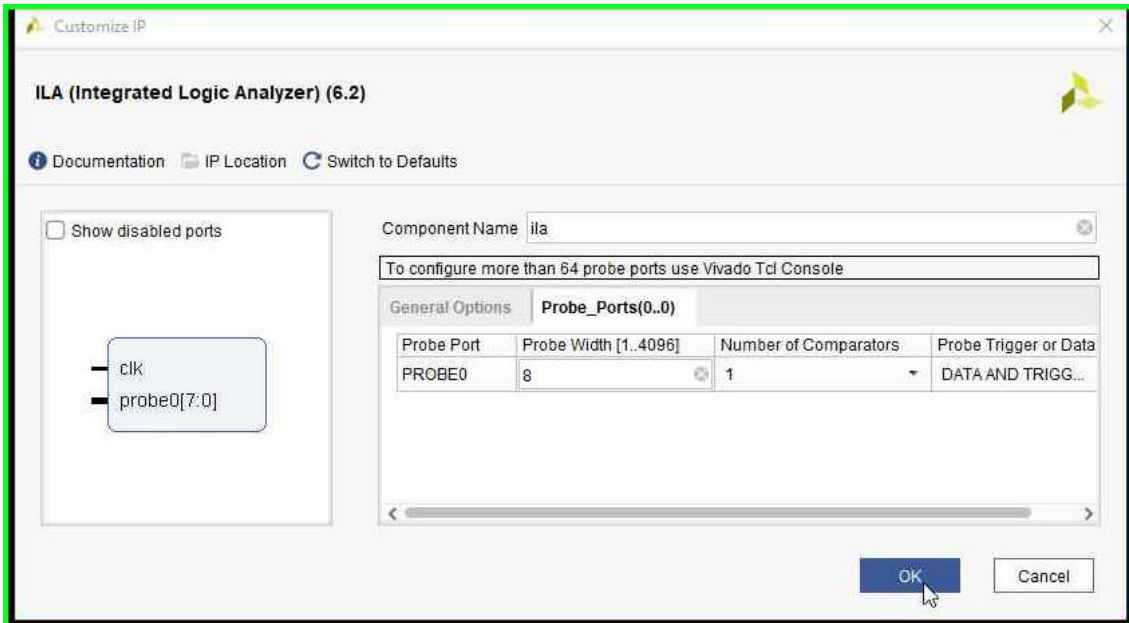
Next, click on the Launch Vivado IP Tool to customize an ILA debug core.



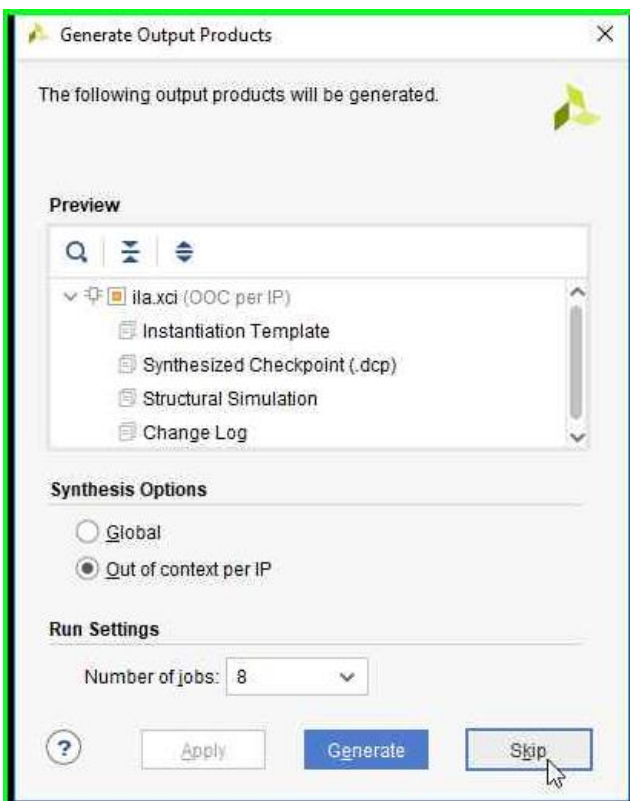
Vivado opens in "Manage IP" mode allowing management of the customized Vivado IP cores for the PathWave FPGA project. The counter which was customized earlier should already be visible. In the IP Catalog, enter "ILA (Integrated Logic Analyzer)" in the IP Catalog Search field to quickly find the ILA IP core.



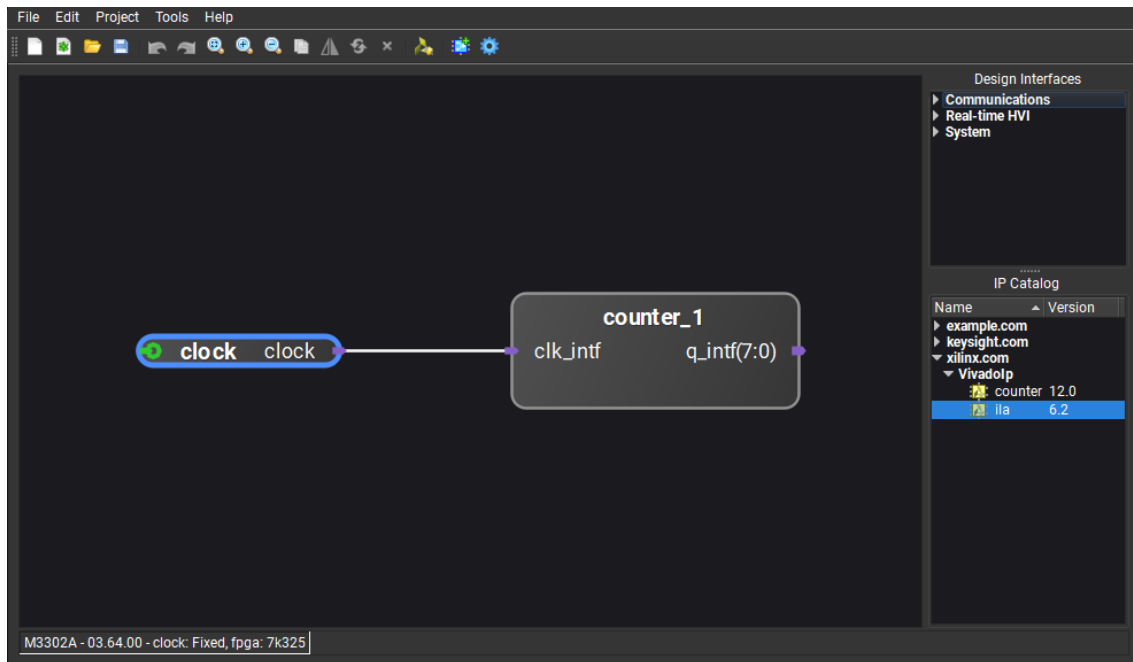
Double click to open the ILA IP customization dialog.



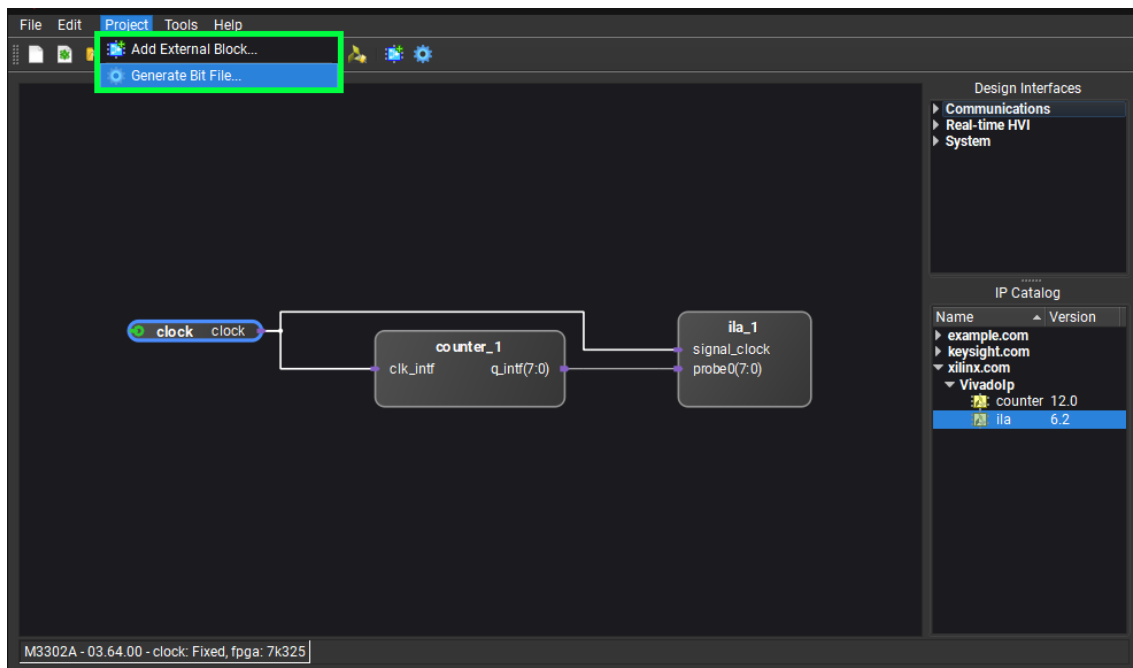
There is already 1 probe by default. Click on the Probe/Ports tab and set the probe width to 8 corresponding to the 8-bit counter in the design. Then, click OK. Next Vivado IP Manager will ask whether to generate the output products for the customized IP.



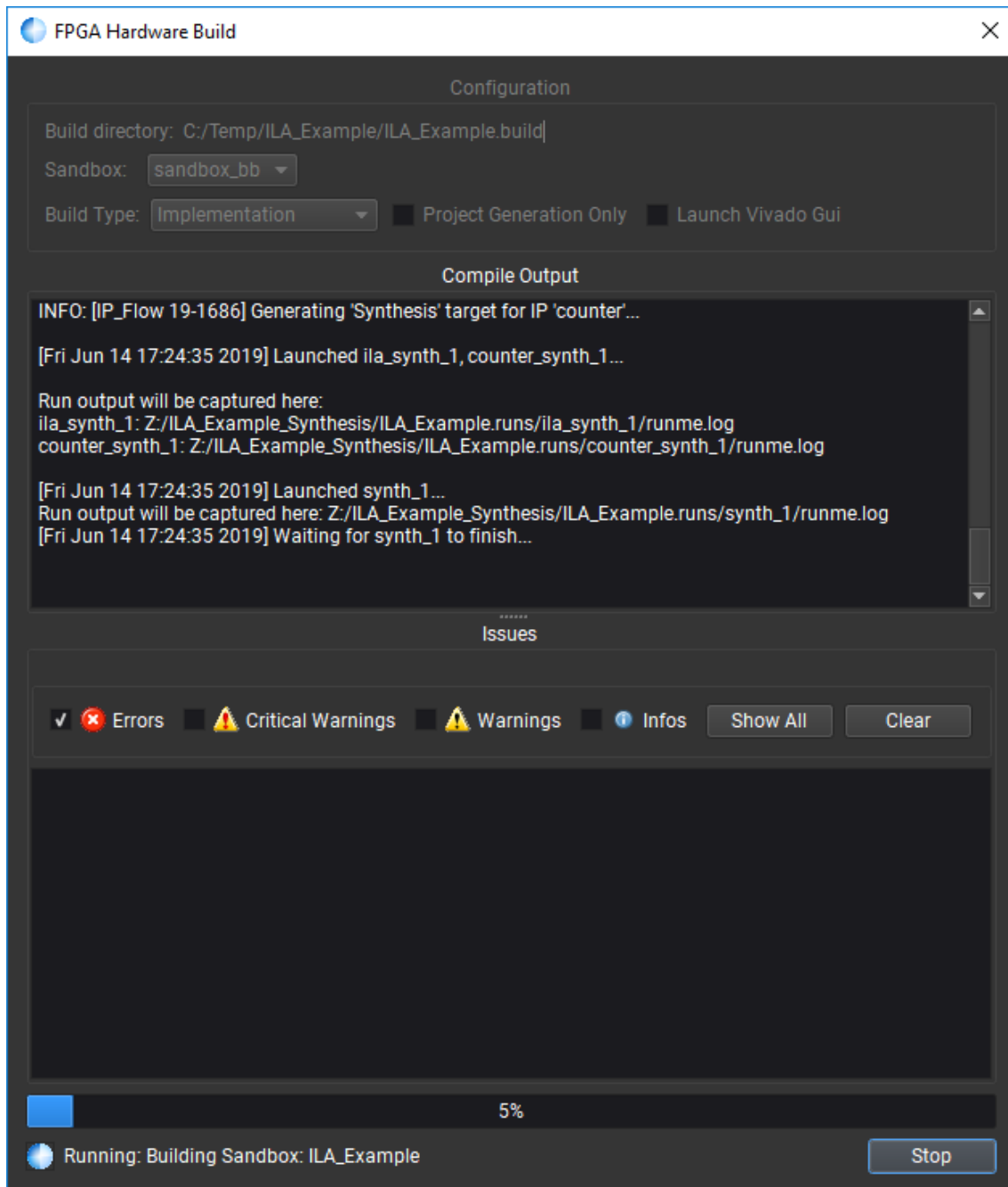
Click the Skip button to skip generation of the customized ILA IP core output products for now. The XCI file is all that is needed to import the component into PathWave FPGA and the output products for any customized IP cores are automatically generated later when building the PathWave FPGA Sandbox design. Then close/exit the Vivado IP Manager to return to the PathWave FPGA window.



In the Vivado XCI panel, double click on the customized ILA IP core to instantiate the ILA in the Sandbox design. Then make the necessary signal connections of the ILA core to the clock and counter output, as shown below

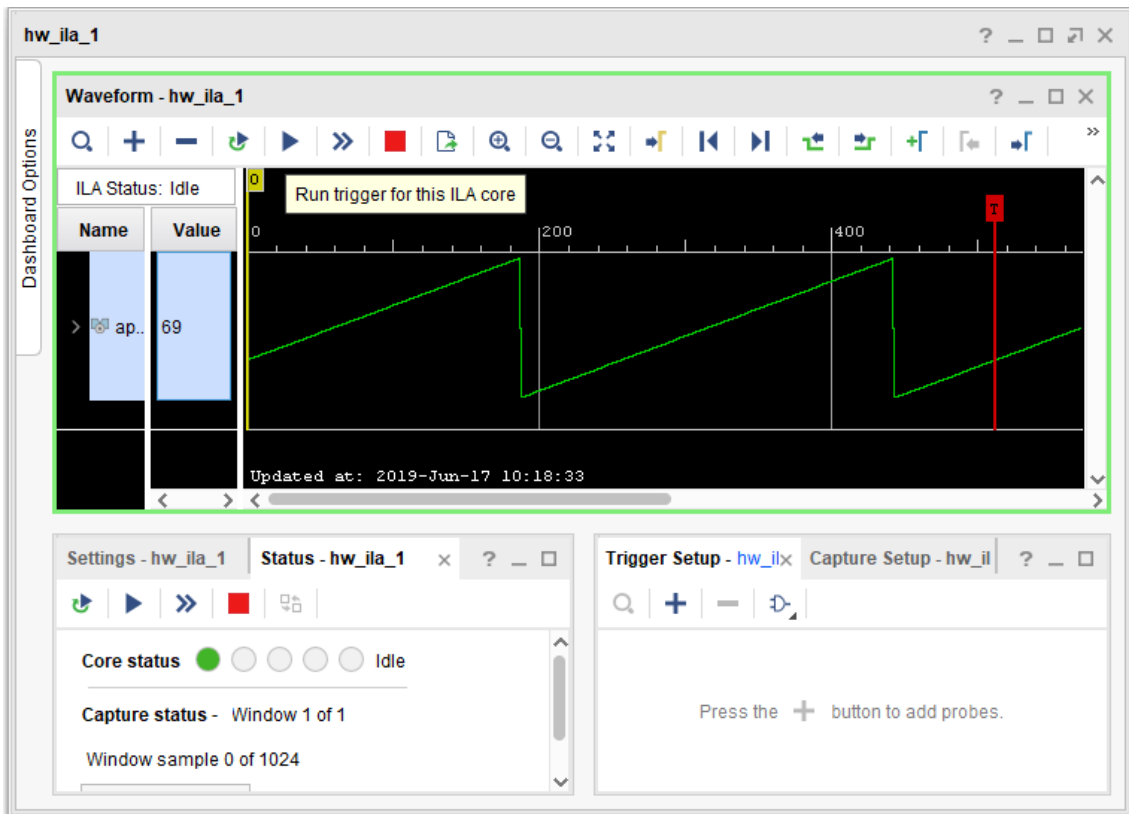


Next click on Generate Bit File and then click the Run button to run synthesis for the Pathwave FPGA project and to generate the bit file.



After the build has completed successfully, the generated bit file can be found in the PathWave FPGA project build results directory. Follow the BSP instructions on how to load the FPGA. The BSP documentation will specify the type of connection required for hardware debugging. With the debug cable connected, open the Vivado Hardware Manager and click on 'Open Target' to connect to the FPGA. After having connected successfully, any detected ILA cores will be displayed in the hardware panel. Click on the ILA to select it and to use the Vivado Logic Analyzer.

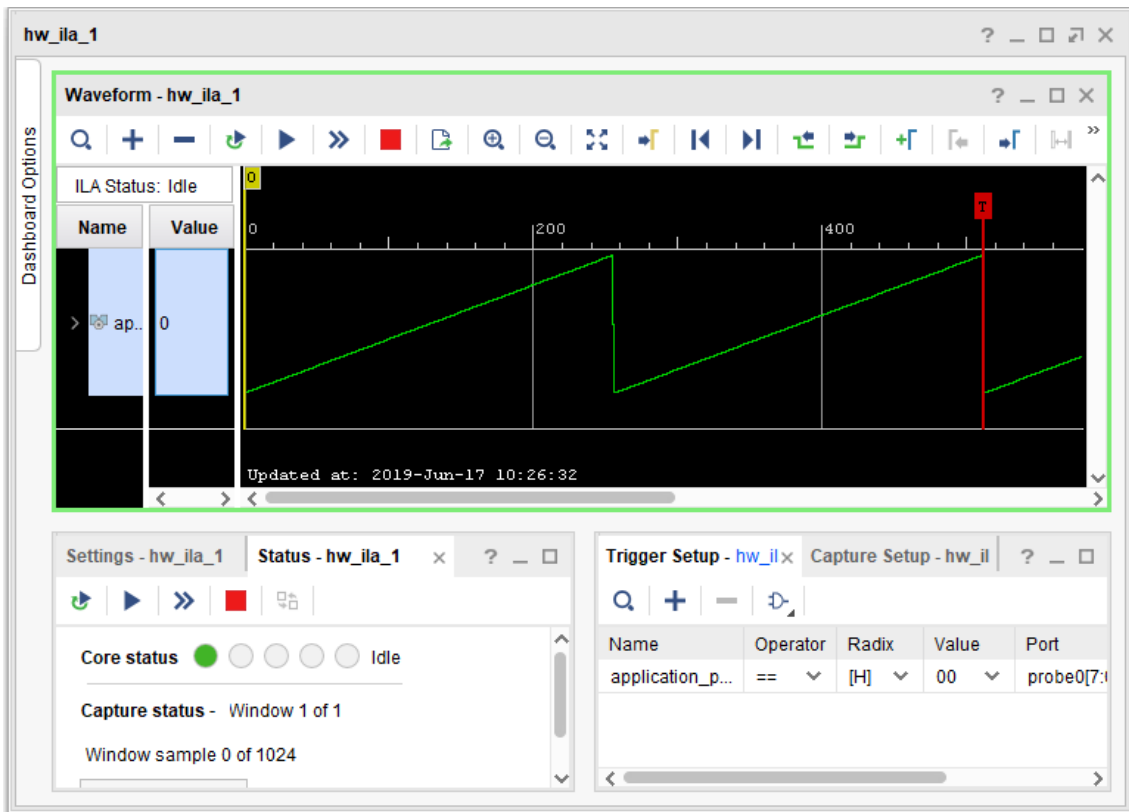
The waveform below shows the 8-bit counter over a few repetitions.



Without any trigger setup, the trigger position is random. However, the Xilinx ILA debug core supports advanced trigger setups. As an example, adding a trigger can stabilize the trigger position within the repetitious waveform.

Trigger Setup - hw_ila_1					
Name	Operator	Radix	Value	Port	Comparator Usage
application_p...	==	[H]	00	probe0[7:0]	1 of 1

With the trigger setup above, the trigger position is stable on counter value equals zero condition.



Multiple acquisitions now produce exactly the same acquisition data, with the trigger position stable on counter value equals zero condition.

The other Xilinx debug cores such as Virtual Input/Output (VIO), Integrated Bit Error Ratio Test (IBERT), JTAG-to-AXI, Memory IP, and System ILA may be used similarly to the ILA debug core. Note that the System ILA is an IP Integrator block and thus is only applicable to IP Integrator designs. Thus the System ILA probe connections would not be visible from the PathWave FPGA design as they would be hidden within the IP Integrator block.

Additional information on debugging using Vivado and using and customizing the ILA may be found in the following Xilinx documents:

- UG908 Vivado Design Suite Programming and Debugging User Guide
- PG172 ILA (Integrated Logic Analyzer) LogiCORE IP Product Guide

User Constraint Files

In some high performance designs, it may be necessary to add constraints to aid in the build process for the sandbox. PathWave FPGA supports the addition of user specified constraint files and tcl files provided that the targeted BSP supports user constraint files. Check the targeted BSP's documentation to confirm whether user constraint files are supported.

PathWave FPGA supports both the inclusion of Xilinx constraint files (*.xdc) and tcl files (*.tcl). These may be things like the addition of timing constraints or location constraints to assist in routability and timing closer. These files may need to be included at certain stages of the build process. The step at which these files are processed is determined by the name of the file used. Multiple files may be used to define different constraints to be used at different stages of the build process.

User constraint and tcl files should be placed in the same directory as the PathWave FPGA project file. This is the *.kfdk file for the project. The constraint or tcl file should have the same name as the project file, followed by a suffix to denote at what step to use the file, followed by either .xdc or .tcl.

Briefly, the build process goes through the following stages:

- Synthesis
- Optimize design (opt_design)
- Place design (place_design)
- Physical optimization (phys_opt_design)
- Route design (route_design)

Files can be specified to run before or after each of these stages by using one of the following suffixes:

- _pre_synth, _post_synth
- _pre_opt, _post_opt
- _pre_place, _post_place
- _pre_phopt, _post_phopt
- _pre_route, _post_route

Any file(s) to be processed should be in the same directory as the project file with the filename <designName>_<suffix>.<fileType>.

For example, if the PathWave FPGA project file was named mydesign.kfdk, then some possible file names are mydesign_pre_opt.tcl or mydesign_post_phopt.xdc.

Glossary

Term	Definition
Bit file	File built from the user design containing the bits to download to the FPGA sandbox.
Block	An HDL IP block that is placed on the PathWave FPGA design schematic.
Board support package (BSP)	A package containing all of the necessary content to target a Keysight Open FPGA. These are installed separately from PathWave FPGA.
Design Canvas	The main part of the PathWave FPGA window where the user develops a schematic.
Design Interfaces	Blocks which communicate between the user design and the outside.
FPGA support package (FSP)	The portion of the BSP that allows you to build a bit file for the target FPGA.
Instrument driver	Provides a C/C++/Python API that you can use to download and control an FPGA bit image.
Interface	A set of ports for a block that can be connected to another compatible interface. Alternatively, an interface can be expanded and the individual ports can be connected to other compatible ports.
IP Repository	IP repositories are libraries of blocks that are loaded into PathWave FPGA. PathWave FPGA has a builtin IP repository, each BSP may come with its own IP repository, and the user may define custom IP repositories. Blocks in these IP Repositories are available in the panes on the right side of the PathWave FPGA window.

Term	Definition
Module	Either a top level module or submodule that is currently the top level module for simulation purposes
Port	An input or output signal of a block.
Program archive	An archive file (.k7z) containing one or more bit files and associated metadata.
Sandbox	The user-configurable region in the FPGA.
Static region	The region of the FPGA that is <i>not</i> user-configurable. This region is implemented by the BSP.
Submodule	Hierarchical schematic design that can be instantiated in either a top level module or another submodule
Top level module	Top of the user design, defines the IO of the sandbox.

IP Developers Guide

PathWave FPGA allows a range of file formats (e.g. VHDL, Verilog, IP-XACT, etc.) for importing IP for usage within a project. Among those formats, the recommended one, that optimizes the support of IP within the software, is IP-XACT. By the usage of this format, PathWave FPGA allows a set of features and conveniences to be applied which include, among others, packing ports to interfaces, simplifying components connectivity, documenting IP usage, allowing specification of dependencies (e.g. libraries, constraints, documentation, simulation files), increasing validation on aspects like hardware compatibility. In this guide, instructions on how an IP-XACT file should be created for an IP, in order to be successfully imported in PathWave FPGA, are provided.

- [Generation of IP-XACT file](#)
- [IP Repositories](#)

Generation of IP-XACT file

IP-XACT, also referred to as [IEEE 1685-2014](#), is a standard which defines a set of xml schemas to describe IP. For more information on IP-XACT, please consult the [IEEE 1685-2014](#) standard. PathWave FPGA uses IP-XACT to define IP blocks to use in a user's design. PathWave FPGA supports a subset of the elements defined in the IP-XACT standard along with custom defined elements.

Since the process of creating an IP-XACT file can be tedious and error-prone, PathWave FPGA includes a tool, IP Packager, that allows IP developers to quickly and effectively create IP-XACT files for their IP. See [IP Packager](#) for a detailed description of this tool.

IP Repositories

IP repositories are directories that contain all the artifacts required to describe an IP. For an IP to be discovered by PathWave FPGA, an IP-XACT file (of the [IEEE 1685-2014](#) standard) is required. To load an IP repository, use the [Settings Dialog](#).

To have your IP repository documentation added to the PathWave FPGA, specify your document location in the [IP Repository Manifest](#).

IP Packager

The recommended format for IP import in *PathWave FPGA* is IP-XACT. *PathWave FPGA* offers a set of features and conveniences enabled by using IP-XACT which include packing ports to interfaces, simplifying component connectivity, documenting IP usage, and allowing specification of dependencies (e.g. libraries, constraints, documentation, simulation files). Since the process of manually creating an IP-XACT file can be tedious and error-prone, *PathWave FPGA* includes **IP Packager**, a tool that allows IP developers to quickly and effectively create IP-XACT files for their IP.

- [Start IP Packager](#)
- [Welcome Page](#)
- [Main Page](#)
 - [Tabs Section](#)
 - [General Tab](#)

- [Interfaces Tab](#)
- [Port Mapping Tab](#)
- [Physical Ports Tab](#)
- [Parameters Tab](#)
- [Enumerations Tab](#)
- [Files Tab](#)

Start IP Packager

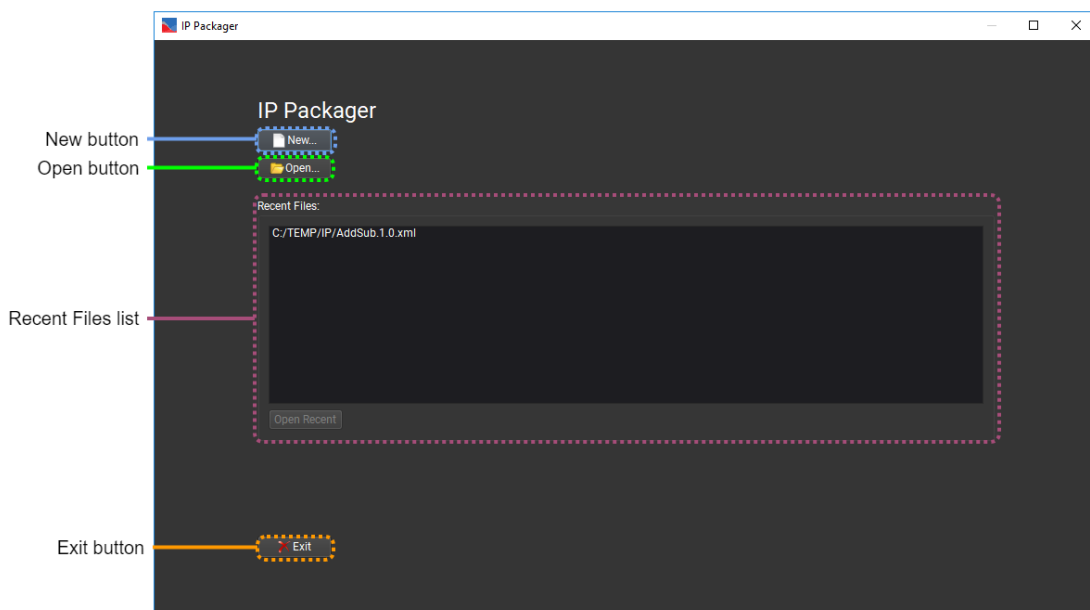
To open up *IP Packager* GUI, start *PathWave FPGA*, go to the *Tools* menu and click *IP Packager*. This will bring up the *IP Packager* GUI.

Import to project

IP-XACT files created by *IP Packager* can be imported into *PathWave FPGA* using one of the methods for importing IP-XACT files described in [Adding Blocks](#).

If a project is loaded in *PathWave FPGA*, and *IP Packager* is used to create new IP, the user will be asked after closing *IP Packager* if any valid IP-XACT files that were created should be imported into the open project.

Welcome Page



New Button

The **New** button will create a new IP-XACT file. Browse to the directory where the new file should be saved, and enter a file name.


Open Button

The **Open** button lets you load an existing IP-XACT file for editing.

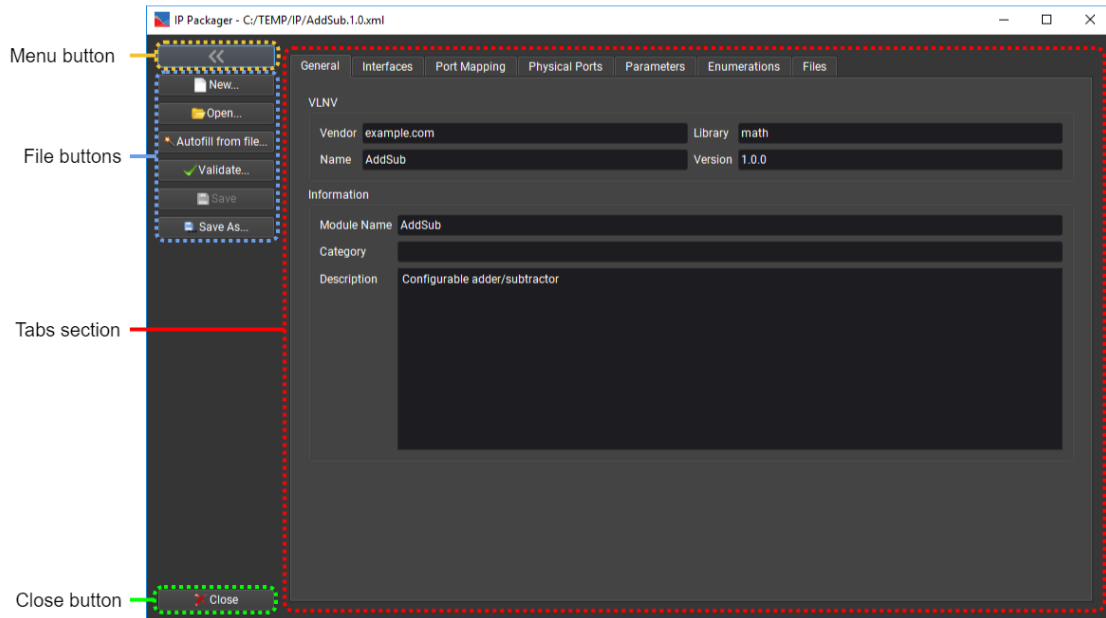
Recent Files List

This will display a list of up to 10 files that were previously processed by the tool, with the most recent first in the list. Select a file and click **Open Recent**, or double-click a file to open it immediately.

Exit Button

The  **Exit** button, exits the IP Packager.


Main Page




Menu button


This button is a toggle switch used to shrink all the menu buttons down to their icon. Click it again to expand them to their normal size.


File Buttons


The  **New** button will create a new IP-XACT file. Browse to the directory where the new file should be saved, and enter a file name. The shortcut is Ctrl-N.

The  **Open** button lets you load an existing IP-XACT file for editing. The shortcut is Ctrl-O.

The  **Autofill from file** button is used to load information from a design file (such as VHDL, Verilog, XCI, or IP-XACT). For example, loading a VHDL or Verilog file will fill the name, physical ports, interfaces, parameters, and will add the file to the Files tab. Interfaces may be inferred from the physical ports by their port names, see [Infer Interface Reference](#). The default for the checkbox controlling interface inference is set in the [PathWave FPGA Configuration](#) dialog. The shortcut is Ctrl-Shift-O.

The  **Validate** button checks whether the current information is valid and sufficient to describe the IP. The shortcut is Ctrl-W.

The  **Save** button saves the current state of the IP to the path selected during the creation of a New file or the path of the file opened. Before saving, it validates the IP and reports any issues. The shortcut is Ctrl-S.

The  **Save As** button allows you to save a new copy of the IP in a different directory or file name. The shortcut is Ctrl-Shift-S.

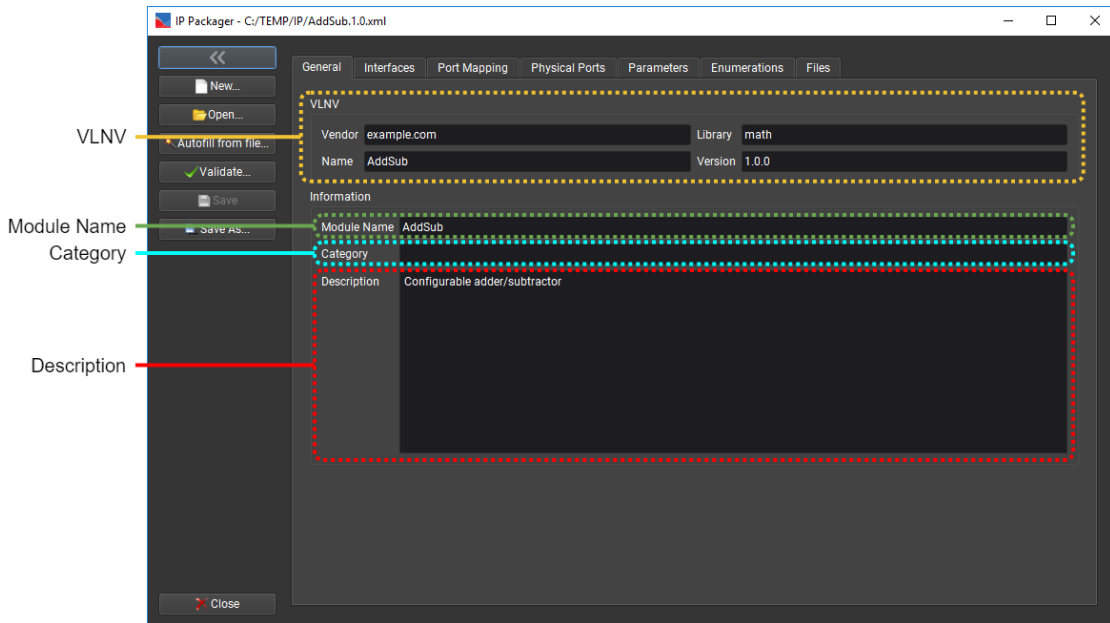
The  **Close** button will close the currently loaded file. If there are unsaved changes, you will be prompted to save them.

To exit *IP Packager*, use the Close button on the window title bar, or press the Escape key. If a project was loaded in *PathWave FPGA* while starting the *IP Packager*, you will have the option to import the IP-XACT files you created into the open project.

Tabs Section

General Tab

This tab contains identification and other relevant information about the IP.



VLNV

VLNV stands for *Vendor-Library-Name-Version* and is a concept introduced by IP-XACT. The VLN of an IP is defined in the first four fields of an IP-XACT component.

PathWave FPGA uses the VLN value to uniquely identify IP, hence each IP must have a unique combination of *Vendor*, *Library*, *Name*, and *Version*. The library field is used to categorize IP in the IP Catalog.

Module Name

This field must match the module name (for Verilog and SystemVerilog) or entity name (for VHDL) of the top-level module represented by this IP. By default, this will be the same as the Name field.

Category

This is an optional field. It is used by PathWave FPGA to further categorize the IP inside the IP Catalog. The library field will label the first level of the tree path, any entries in the Category field will label intermediate levels in the tree path, and the component *Name* will label the leaf.

For example, if an IP has the VLN *keysight.com::Algorithms::StreamAdder::1.0* and the category *Math*, it will be available in PathWave FPGA library under the tree path:

- ALGORITHMS
 - MATH
 - StreamAdder

Categories can be nested with the slash character (forward or backward). For the example above, but with the category *Math/Adders*, tree path would be:

- ALGORITHMS

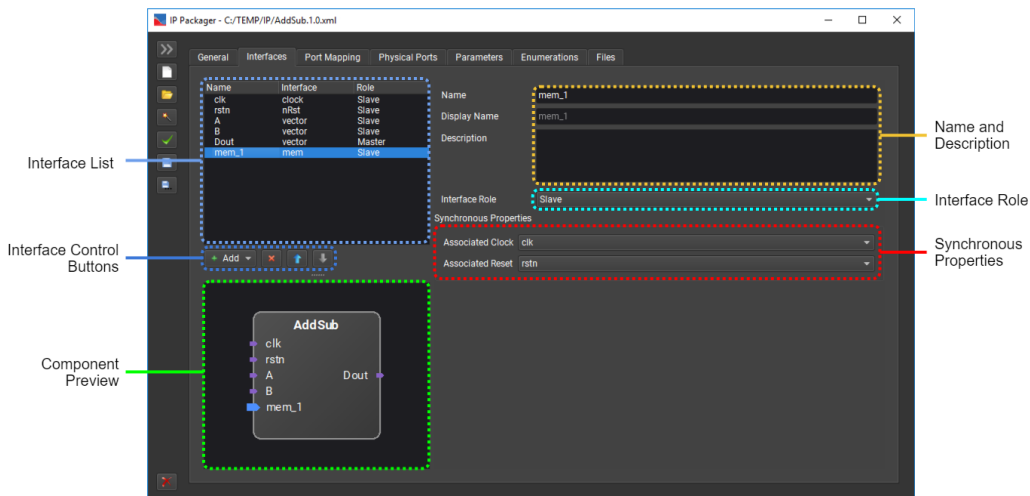
- MATH
 - ADDERS
 - StreamAdder

Description

It provides a text section for entering a description about the IP being created. PathWave FPGA displays the IP description when a component is added into the design canvas, and also in the component's Properties dialog. This is an optional field.

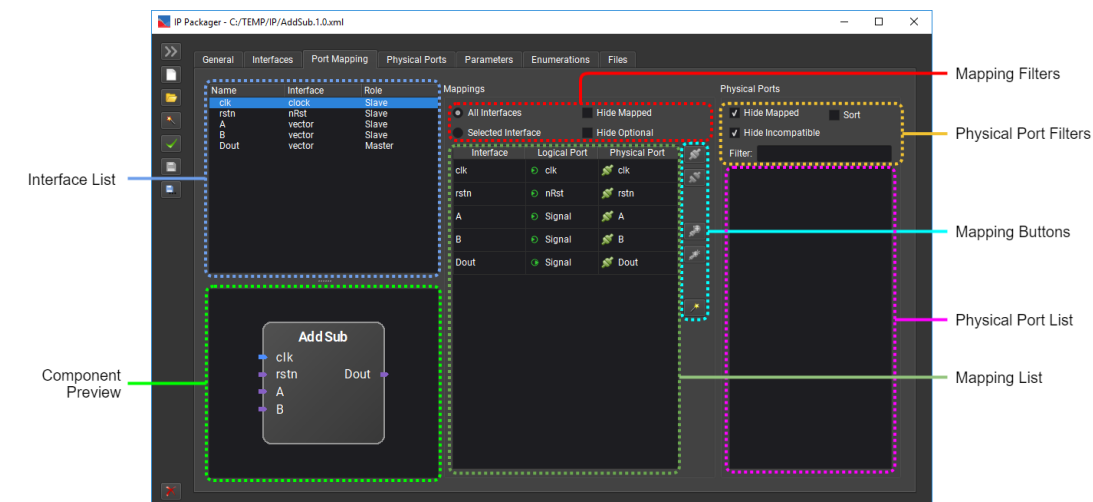
Interfaces Tab

Use this tab to configure the standard interfaces in the IP definition. The usage of this tab is similar to the one defined in [Configuring Submodule Interfaces](#). Please consult that page for usage instructions.



Port Mapping Tab

Each interface added in the **Interfaces** tab has one or more logical ports. These need to be mapped to the physical ports of the IP design.



Interface List

This list shows the interfaces that are defined in the **Interfaces** tab. Select an interface to show the logical ports for that interface.

Component Preview

This preview shows how the IP component will be displayed in a *PathWave FPGA* canvas. When an interface is selected in the **Interface List**, that interface will be highlighted in the preview.

Mapping Filters

All Interfaces radio button: When selected, the **Mapping List** will show the logical ports for all interfaces. A new column will appear to show which interface the logical port is from.

Selected Interface radio button: When selected, the **Mapping List** will show only the logical ports for the interface selected in the **Interface List**.

Hide Mapped: The **Mapping List** will only show logical ports that have not been mapped. Use this to focus on mapping the unmapped ports.


Hide Optional: The **Mapping List** will only show logical ports that are required by the interface. Any unmapped optional ports will be disabled when the IP is saved.


Mapping List


A table that displays the mappings between logical ports of interfaces to physical ports of the IP. It contains three columns:


- **Interface**: (Only visible when the **All Interfaces** radio button of the Mapping Filters is selected) This shows the name of the interface to which the logical port belongs.
- **Logical Port**: This shows the name of the logical port. For a specific row, it shows the name of the logical port that takes part in the mapping. An icon with the direction of the logical port is displayed on the left side.
- **Physical Port**: This shows the name of the physical port that the logical port is mapped to. If the logical port is not mapped to a physical port, this will show the red open mapping icon 🛑 if the logical port is required, or the yellow open mapping icon 🟡 if it is optional. The green connected mapping icon 🟢 indicates that the port is mapped.


Mapping buttons

 **Map** button: This will map the logical port selected in the **Mapping List** to the physical port selected in the **Physical Ports List**. You can also double-click the physical port to map it to the selected logical port.

 **Unmap** button: This will remove the mapping of the selected logical port.

 **Map to new** button: This will create a new physical port and map it to the selected logical port. The name of the physical port is `<interface name>_<logical port name>`.

 **Map all to new** button: This will create new physical ports for all unmapped logical ports in the **Mapping List**. It behaves the same as the **Map to new** button.

 **Infer interfaces** button: This will infer interfaces from physical ports by their port names. For full inference rules, see the [Infer Interface Reference](#). Any newly inferred interfaces will appear in the **Interfaces List** and the logical ports of those interfaces will be mapped to their physical ports. The interface names and graphical order may be changed, and interface descriptions may be entered, in the **Interfaces Tab**.

Physical Port Filters

Hide Mapped check box: When checked, the **Physical Ports List** will not show any physical ports that are mapped to a logical port other than the one selected in the **Mapping List**.

Hide Incompatible check box: When checked, the **Physical Ports List** will not show any physical ports that are incompatible with the logical port selected in the **Mapping List**.

Sort check box: When checked, the **Physical Ports List** will show the ports in alphabetical order. Otherwise the ports will be in the order that they appear on the **Physical Ports Tab**.

Filter: The **Physical Ports List** will only show physical ports that contain the text in their name. The filtering is case-insensitive.

Physical Port List

A list of the physical ports for the IP. Use the **Physical Port Filters** to show only a subset of the physical ports. If a logical port is selected in the **Mapping List**, you can double-click a physical port to create a mapping between the two.



icon and red text color is used for incompatible physical ports



icon is used for unmapped compatible physical ports



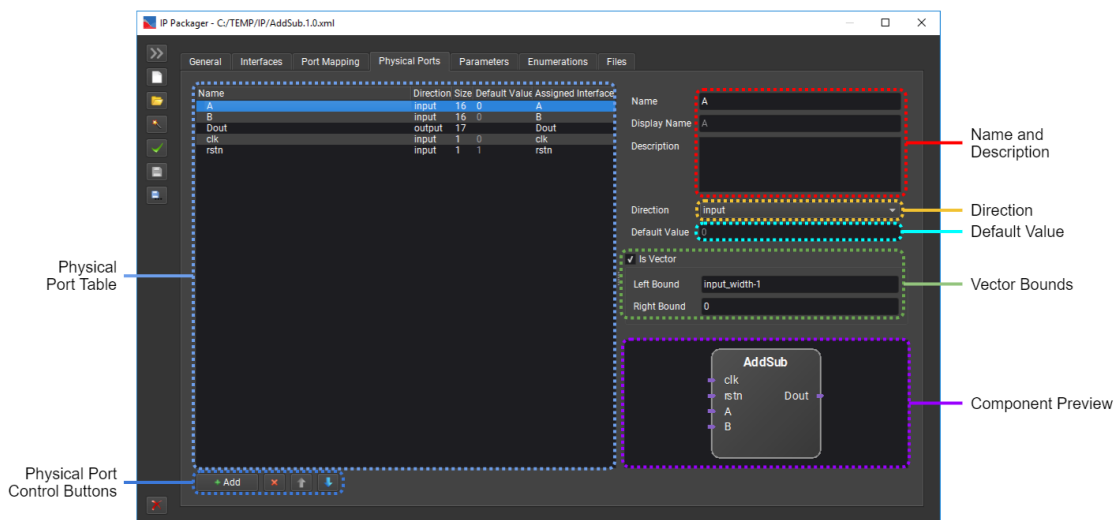
icon is used for mapped compatible physical ports



icon is used for the physical port that is actually mapped to the selected logical port

Physical Ports Tab

The physical ports are the ports presented by the IP top-level implementation file. Usually they are loaded from a file, but you may create or modify them manually if needed.



Physical Port Table

Contains a list of the physical ports for the IP. Each port is displayed in columns showing its name, its direction and size, default value if appropriate, and its assigned interface.

Physical Port Control Buttons

+ Add button: Creates a new physical port with a unique name.

✗ Remove button: Removes the selected physical port.

↑ Up button: Moves the selected physical port up.

↓ Down button: Moves the selected physical port down.

Name and Description

The Display Name is what will appear in *PathWave FPGA*. The Description can be viewed by double-clicking the port on an instance in the canvas.

Default Value

This value is used when the port is not connected. When the field is empty, it will show the default value for the logical port of the mapped interface in gray text. The default value can only be set for input ports.

Vector Bounds

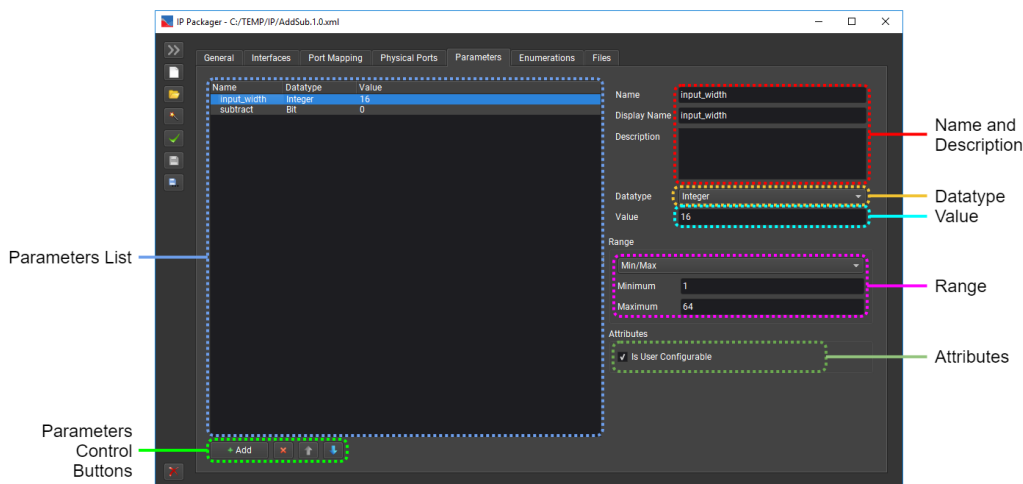
Configure the left and right bounds of a vector to set the width. Either the left or right bound must be 0.

Component Preview

This preview shows how the IP component will be displayed in a *PathWave FPGA* canvas. When an unmapped physical port is selected in the **Physical Port Table**, that port will be highlighted in the preview.

Parameters Tab

A model might use parameters for controlling the port widths or any other configurable feature of the model. The Parameters Tab allows you to add, modify, and remove parameters.



Parameters List

Contains the list of parameters of the IP. Each entry is split in three columns:

- **Name:** displays the name of the parameter. In case this is a module parameter, it should match the name in the actual design file.
- **Datatype:** displays the acceptable datatype of the value.
- **Value:** displays the default value of the parameter.

Parameters Control Buttons

+ Add button: Creates a new parameter with a unique name.

✗ Remove button: Removes the selected parameter.

↑ Up button: Moves the selected parameter up.

↓ Down button: Moves the selected parameter down.

Name and Description

The fields of this group describe the parameter:

- **Name:** the name of the parameter. In case this is a module parameter, it should match the name in the actual design file.
- **Display Name:** a user friendly name for this parameter. This name will be shown in PathWave FPGA.
- **Description:** a description for this parameter. The description will be available in PathWave FPGA.

Datatype

PathWave FPGA supports the following data types for parameters:

- *Bit*: represents 1-bit value
- *Byte*: represents an integer value of 8-bits
- *Short Integer*: represents an integer value of 16-bits
- *Integer*: represents an integer value of 32-bits
- *String*: represents a string of characters

Value

The value or expression to be used by default for this parameter. The possible value is restricted by the selected datatype and the specified Range.

Range

Allows three different range validations for the value of the parameter:

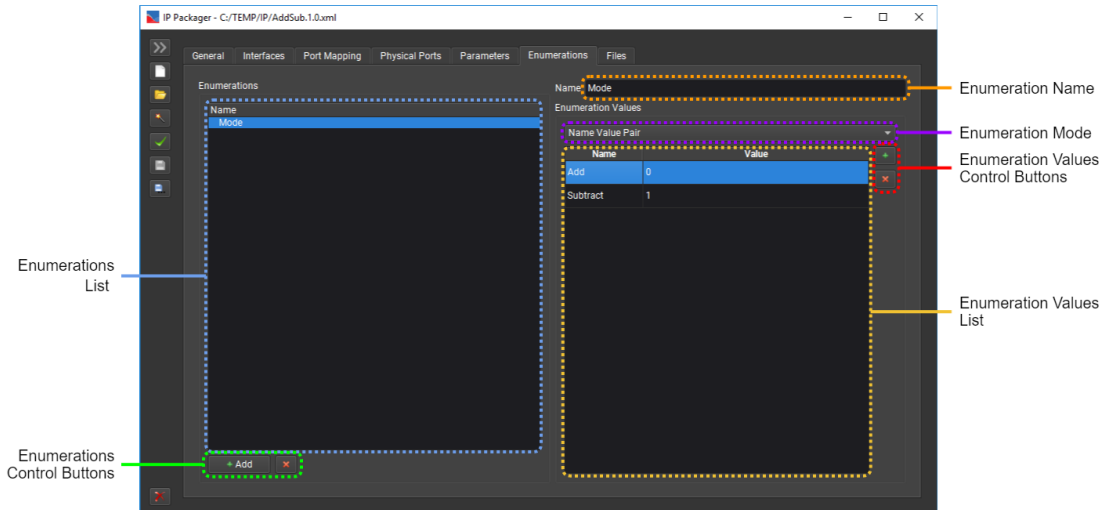
- **No Range**: If this is selected, the value of the parameter is only limited by the available range of the selected datatype.
- **Min/Max**: If this is selected, two extra fields are displayed to define the continuous value range for the parameter. This selection has no effect if the selected datatype is *string* or *bit*.
 - *Minimum*: The minimum value the parameter can take. The value should have the same datatype as the one selected for the parameter and should be no larger than *Maximum*. If left empty, minimum is the $-\infty$.
 - *Maximum*: The maximum value the parameter can take. The value should have the same datatype as the one selected for the parameter and should be no smaller than *Minimum*. If left empty, maximum is the $+\infty$.
- **Enumeration**: If this is selected, a menu with the available valid enumerations is displayed. If nothing is displayed, go to the Enumerations tab to add a new enumeration or fix an invalid one. The value of the parameter is restricted by the allowed values of the selected enumeration.

Attributes

Is User Configurable checkbox: If this is checked, each instance can give this parameter a different value than the default. User Configurable parameters will be displayed to the *PathWave FPGA* users in the component dialog of this IP. This box should be unchecked for parameters that are not directly controlled by the user. This would be the case if a parameter is an expression of other parameters and hence can be calculated from these other values.

Enumerations Tab

Some parameters of the model may be restricted to specific discrete values. The Enumerations Tab allows the user to specify enumerations that can be used as range validators inside parameter definitions.



Enumerations List

This is the list of enumerations that are defined in the context of the IP and can be referenced by parameters.

The names of the enumerations should be unique and should start with a letter, colon (:) or underscore (_) character and can be followed by any number of letter, numeric, colon (:), underscore (_), dot (.) or hyphen (-) characters.

If an enumeration is invalid (in case of invalid name structure or because of insufficient number of defined elements), it is displayed with red text color and a tooltip is available that describes the issue.

Enumerations Control Buttons

+ Add Enumeration button: Creates a new enumeration and adds it to the list, giving it a unique name.

✖ Remove Enumeration button: Removes the currently selected enumeration from the enumerations list. If the enumeration selected is being used by any parameter of the model, the user will be given the option to abort the remove action.

Enumeration Name

Name of the currently selected enumeration. Can be edited to change the name. If an invalid name is entered, the name will turn red and the enumeration list will not be updated until the name is changed to a valid value.

Enumeration Values List

This is the list of values that a selected enumeration can take.

The definition of values can take two formats:

- *list of name/value pairs*: in this case, the names of the list should be unique
- *list of values*: in this case, the values should be unique

Enumeration Values mode

Value Mode combo box: Changes between the two element value types: **Name Value Pair** or **Value Only**.

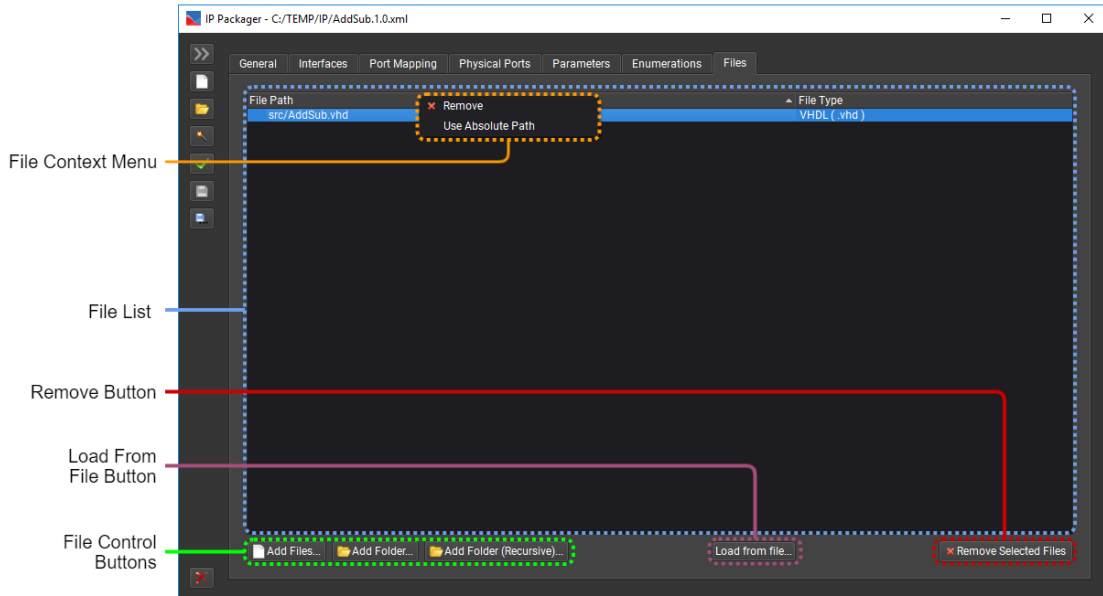
Enumeration Values Control Buttons

+ Add Enumeration Value button: Creates a new enumeration name/value pair (or just value, if **Enumeration Values** mode is **Value Only**).

✖ Remove Enumeration Value button: Removes the selected enumeration value from the list.

Files Tab

An IP-XACT file describes IP defined in one or more other files, such as VHDL or Verilog files. This tab defines the files used for the IP during the build process, as well as documentation files in PDF format.





File List


Displays the list of files that will be used during the synthesis and implementation of the IP, as well as documentation files in PDF format.

- There must be at least one implementation file defined for each IP-XACT file.
- Files are represented either by their absolute path or by the relative path from the parent directory of the IP-XACT file. By default, all the files are represented by their relative path. Right-click a file to change the path to absolute or back to relative. Double-click a file to manually modify the path.
- A file will be highlighted in red if any errors are detected with that file. Hovering over the bad file will show a tooltip describing the error.
- A documentation file can be provided in the form of a PDF file. This file will then be displayed to the user when he asks for help for this IP.

File Control Buttons

 **Add Files** button: Browse to the implementation files for this IP and add them to the **File List**. You may select multiple files at once.

 **Add Folder** button: Add all the implementation files in a directory to the **File List**. This will not include files in subdirectories.

 **Add Folder (Recursive)** button: Add all the implementation files in a directory to the **File List**. This will search all subdirectories recursively.

Load from File button: This button will load the physical ports and parameters from the selected file. Any existing physical ports and parameters are replaced with the ports and parameters loaded from the file. The port mappings will be restored to compatible physical ports with the same name. If an existing parameter is also in the file, the value and data type will be updated while all other properties remain unchanged. If an existing parameter is not in the file, it will be removed. Interface names and descriptions are restored if possible. Interfaces may be inferred from the physical ports by their port names. For full infer rules see, [Infer Interface](#)

Reference. The default for the checkbox controlling interface inference is set in the [PathWave FPGA Configuration](#) dialog.

✖ **Remove Selected Files** button: Remove the selected files from the **File List**.

File Context Menu

✖ **Remove:** Removes the selected files from the **File List**.

Use Absolute Path: Converts the selected file path from relative to absolute. This will only appear if one or more selected files are in relative form.

Use Relative Path: Converts the selected file path from absolute to relative. This will only appear if one or more selected files are in absolute form.

IP Repository Manifest

The IP repository manifest is a JSON file that describes various aspects of an IP repository. Currently, the primary use for this manifest is to define a location for the repository documentation. If the documentation location is defined, PathWave FPGA will add a link to it in the Help → IP Repositories menu.

The manifest file must be named "manifest.json" and must be located in the root of the IP repository path.

Manifest Format

Below is an example of an IP repository manifest:

```
{
  "manifest": {
    "version": "1.0",
    "type": "IP Repository Manifest"
  },
  "repository": {
    "name": "PathWave FPGA Example IP",
    "version": "1.0.0",
    "description": "Example IP description",
    "doc": {
      "path": "doc/ExampleDoc.pdf",
      "url": "www.example.com/ExampleDoc.pdf"
    }
  }
}
```

If **path** or **url** is not empty, PathWave FPGA will add a menu item named "<name> - <version>" to the Help → IP Repositories menu. If both **path** and **url** are defined, path takes precedence.

The **description** element is not currently used by PathWave FPGA.

Tutorials

- [IP Packager Tutorial](#)
- [HVI Tutorial](#)
- [Import Vivado High-Level Synthesis \(HLS\) generated IP](#)
- [Power of Two Decimation Tutorial](#)
- [Xilinx System Generator for DSP™ Tutorial](#)

IP Packager Tutorial

While PathWave FPGA can directly import simple HDL files directly, more complex designs need to be "Packaged" to be fully supported in PathWave FPGA. *Packaging* means creating an IP-XACT 2014 ([IEEE 1685-2014](#)) file to describe the IP. An IP-XACT file contains information about IP that describes things like the ports, the interfaces, the files needed to build it, descriptive text, and customization parameters among other things. Packaging IP is needed if the IP needs more than one source file to be built (e.g. if the design is hierarchical), if the design wants to use interfaces but the port names don't match the PathWave naming convention, or if the design uses complex expressions in its parameterization.

A big reason to package IP is to utilize logical interfaces. An HDL IP block has physical ports which are the input and output signals for the IP block. One or more of these ports can be combined into a logical interface which describes how the signals interact and connect with other signals. An interface may consist of a single port or even a signal wire. An example of this is a clock interface. Other interfaces, such as the AXI-MM interface, may have dozens of potential ports. By describing which ports constitute a particular interface and which role each port has, the IP-XACT description eases connecting interfaces together. Two AXI interfaces can connect together with only one graphical connection even though a considerable number of individual ports will be connected in the hardware.

While the IP-XACT file is text and can be manually created in any text editor, it is simpler and easier to use the PathWave FPGA [IP Packager](#).

These tutorials will show how to package (create IP-XACT) for simple and not so simple IP blocks using the PathWave FPGA IP Packager.

The example files used in these tutorials can be found in the PathWave FPGA install directory under `examples\IpPackagerTutorial`.

There are four tutorials. The first tutorial, [Simple HDL done automatically](#), shows the easiest way to package a design. In this tutorial, most of the information is autofilled from the source HDL file.

In the second tutorial, [Simple HDL done manually](#), the same HDL file is packaged manually rather than using the autofill from file function. This tutorial shows how to manually modify ports and interfaces.

The third tutorial, [Parameterized HDL](#), shows how to generalize an IP block using simple parameters. This includes different types of parameters such as integer, bit, or enumerated.

The fourth tutorial, [Advanced IP Packaging](#), shows more advanced techniques such as using non-user configurable parameters, parameter expressions, and design hierarchy.

Simple HDL done manually

Instead of automatically filling in the ports and interfaces from the IP's HDL source file, as described in the previous tutorial, the information may be added manually. This might be useful if the HDL for the IP doesn't yet exist or if the IP already has IP-XACT but the IP has changed, e.g. a new port was added. In that case it might be easiest to manually make the changes.

This tutorial will package the same module used in the previous tutorial:

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

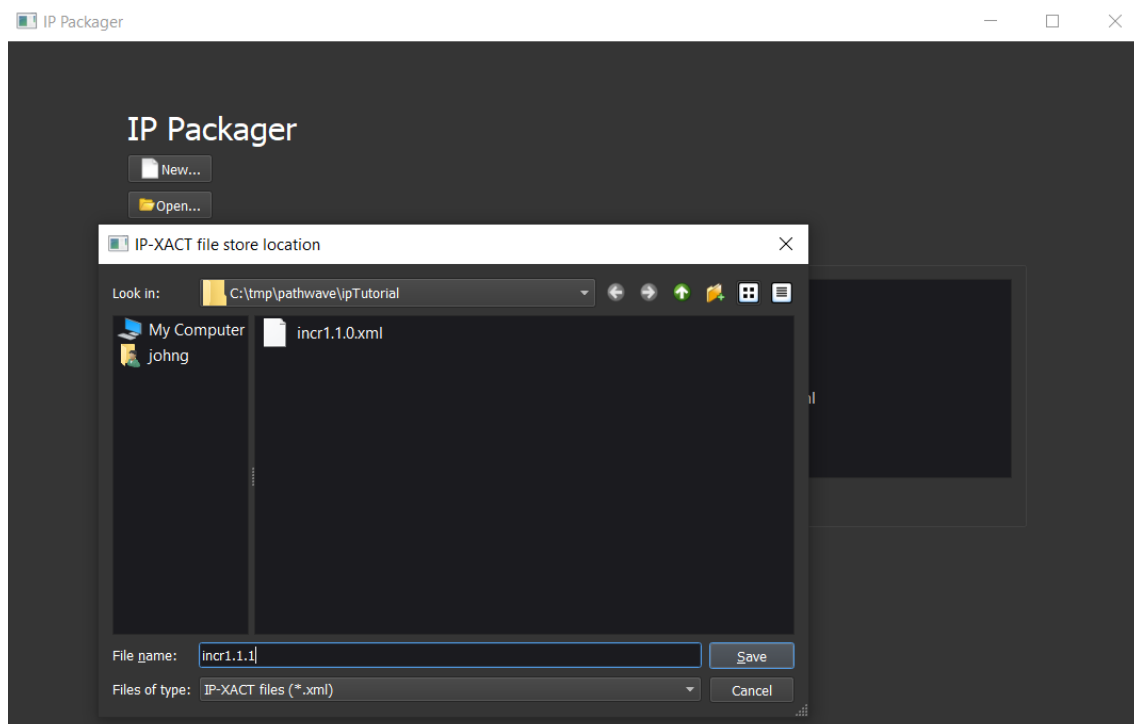
entity incr1 is
  port (
    clk    : in  std_logic;
    nrst   : in  std_logic;           -- Active low reset
    incr   : in  std_logic_vector(7 downto 0);
    count_tdata : out std_logic_vector(7 downto 0);
    count_tvalid : out std_logic
  );
end incr1;

architecture Behavioral of incr1 is
  signal count : std_logic_vector(7 downto 0);
begin -- Behavioral
  count_tdata <= count;
  count_tvalid <= '1' when (incr /= 0) else '0';
  process(clk)
  begin
    if (nrst = '0') then
      count <= (others => '0');
    else
      count <= count + incr;
    end if;
  end process;
end Behavioral;

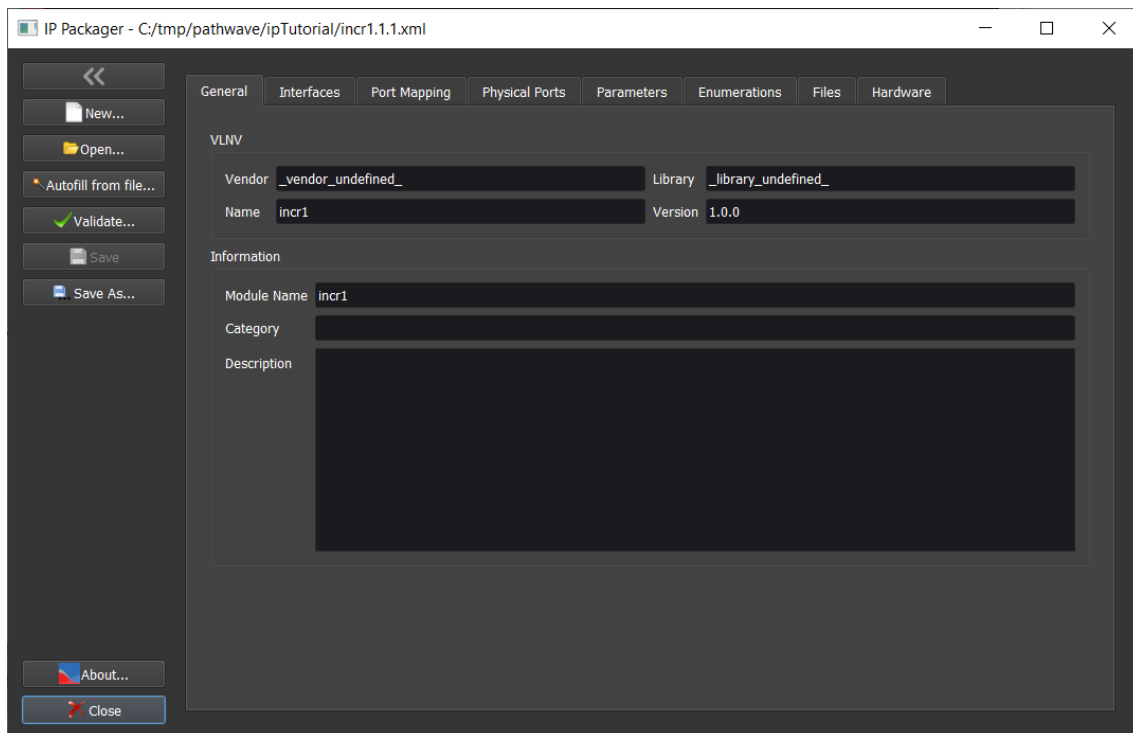
```

This block will increment an internal counter based on its incr input and output the counter value on an AXI-streaming count interface. It also has a clock input and an active low nrst input.

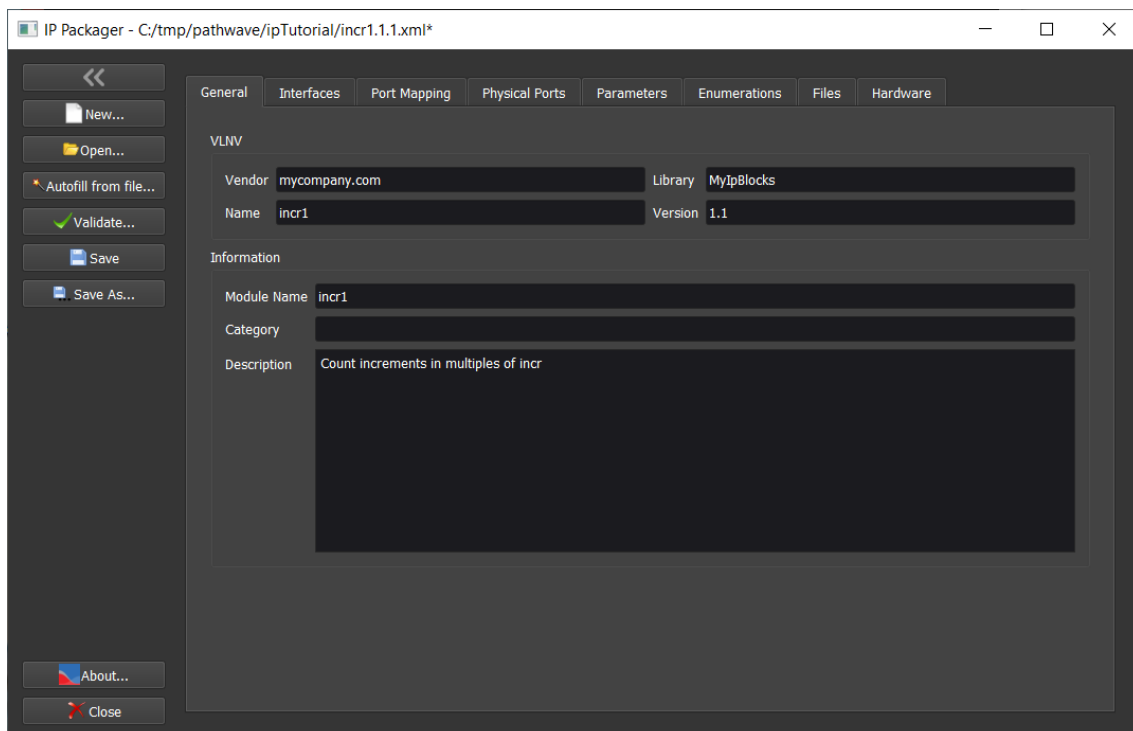
To package this IP, first start the IP Packager which is located in the Tools menu inside PathWave FPGA. The manual process starts the same as in the previous tutorial. We'll call this one version 1.1 to differentiate it from the previous tutorial. Click the "New..." button and navigate to the desired location for the IP-XACT file, enter "incr1.1.1" for the name for the file, and click "Save":



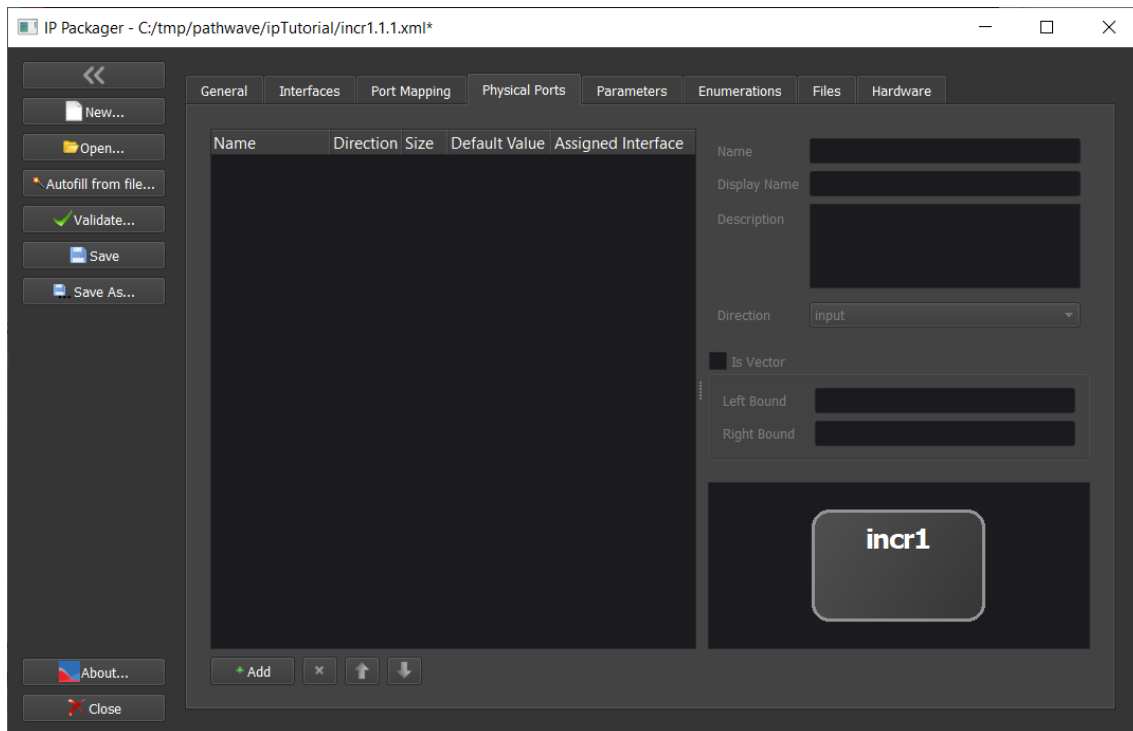
You now have a mostly empty description of the IP block:



This time, we won't click the *Autofill from file....* Instead we'll enter the information manually. Fill out the fields in the *General* tab as shown below. This time we'll use *Version 1.1*:



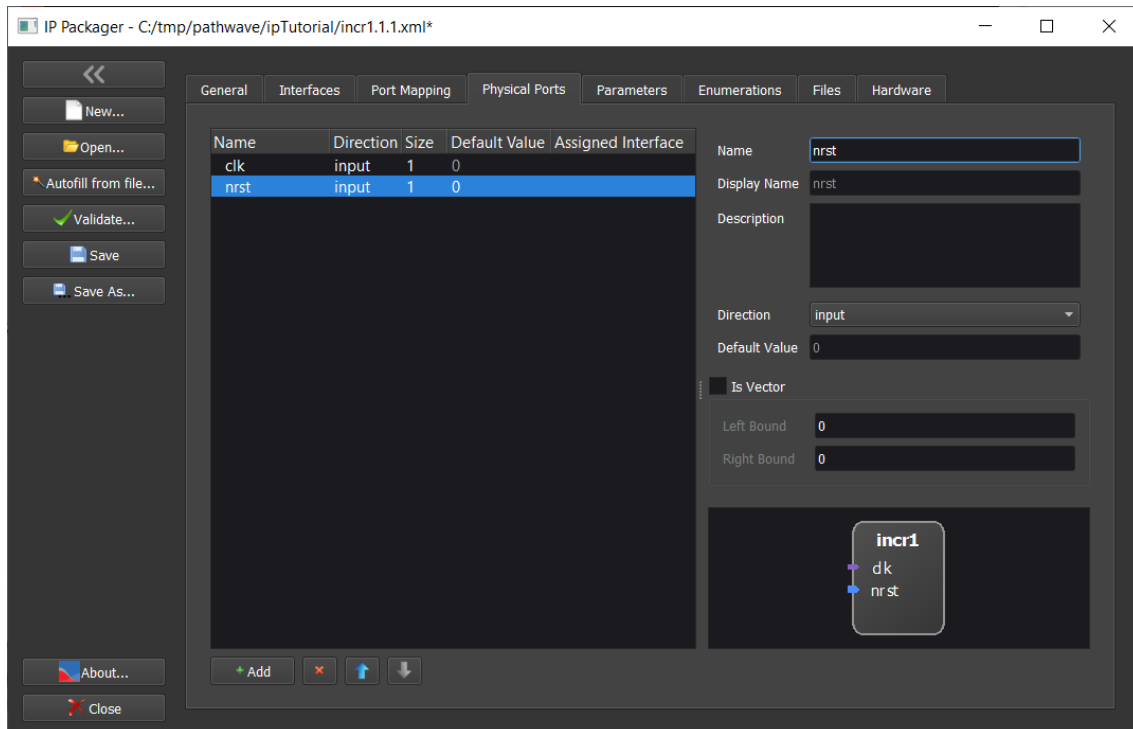
Now it is time to enter the physical ports in the IP block using the *Physical Ports* tab:



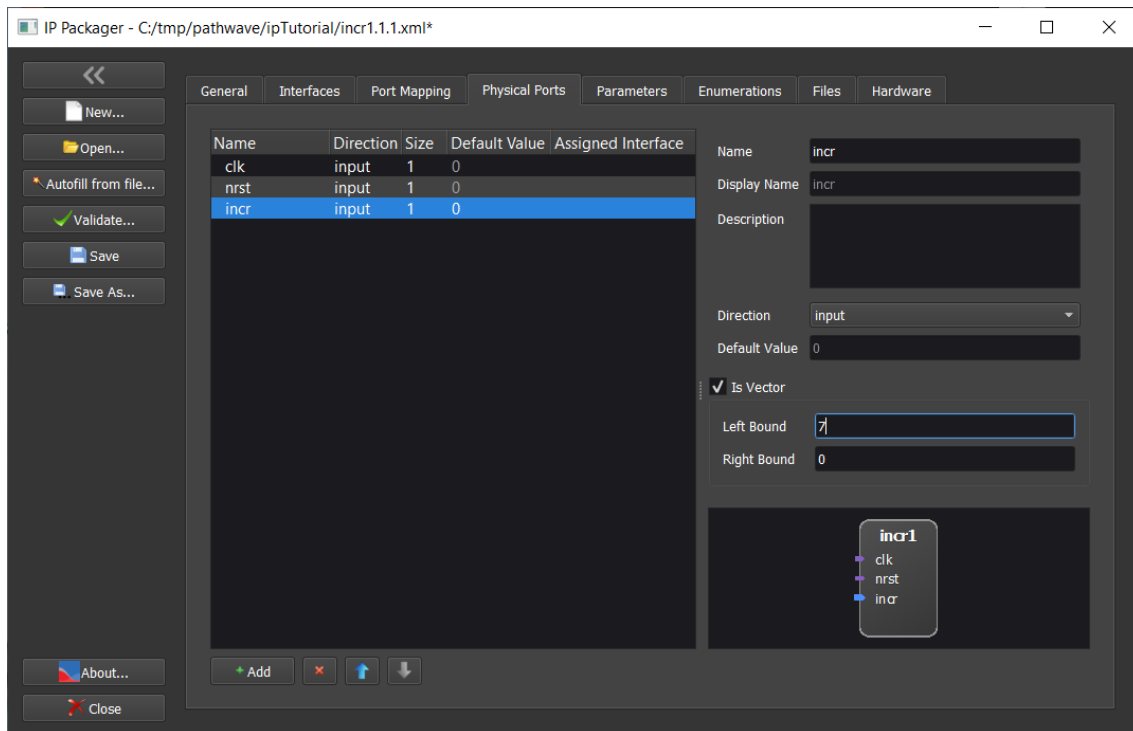
Initially the list of ports is empty. To add ports, we first create a new port with the **+Add** button, change the name to the actual port's name, set the *Direction* to either *Input* or *Output*, and if the port is a vector, specify the bounds.

First we'll add the *clk* port. Click the **+Add** button to add a port. Change the port name from the default *port_0* to *clk*. The default *Direction* is *Input*, so this does not need to be changed.

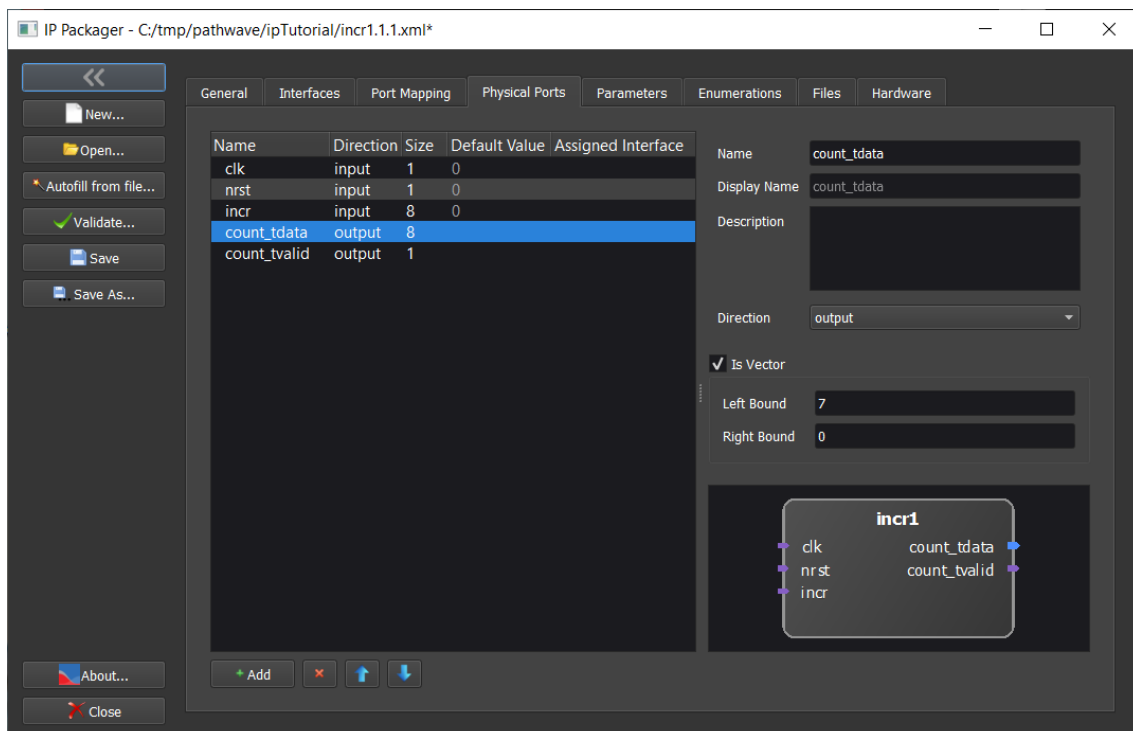
Next we'll add the *nrst* port in the same way:



We then add the *incr* port. This one is a vector, so the *Is Vector* box is checked, and the *Left Bound* is changed to 7:

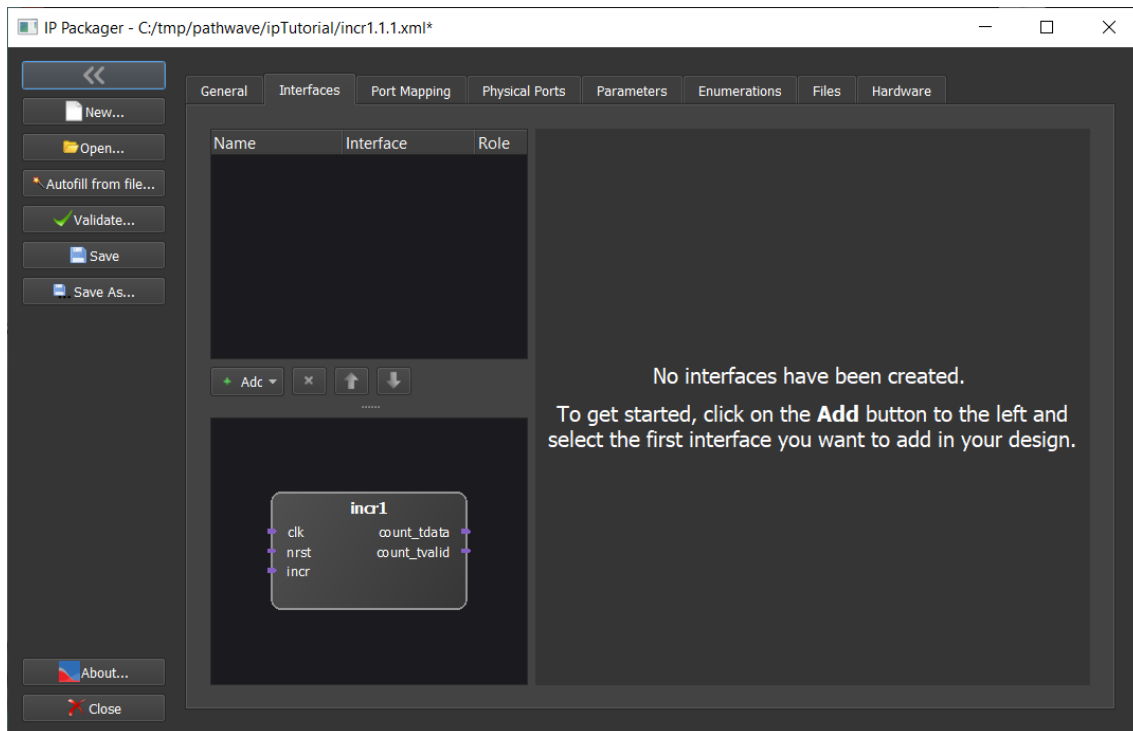


We then add the ports *count_tdata* and *count_tvalid* in a similar way. These will need the *Direction* changed to *Output*.

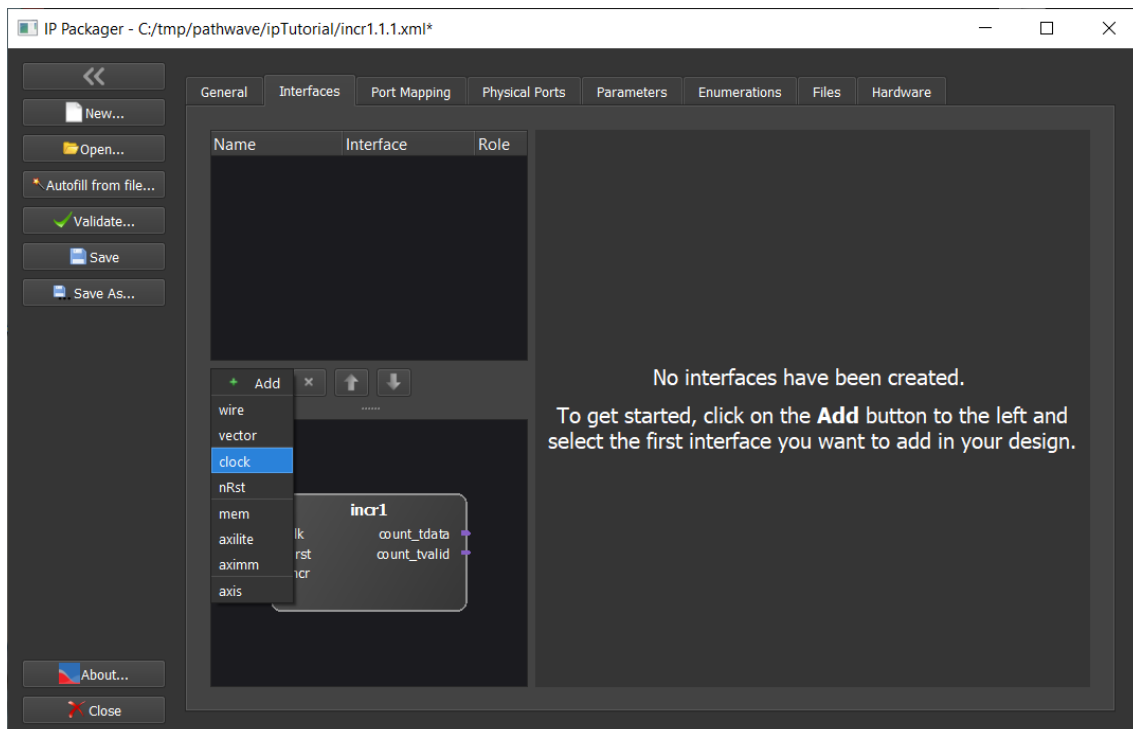


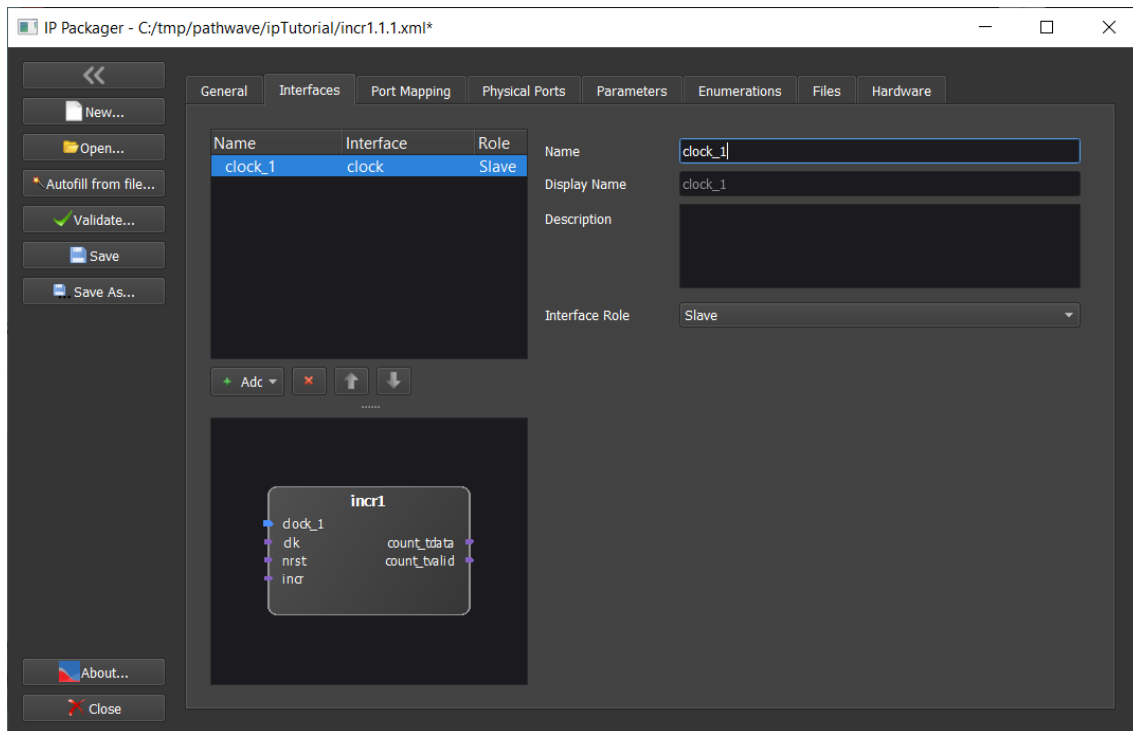
Since we haven't assigned or even defined any interfaces yet, the *Assigned Interface* column is blank.

Next we'll define the interfaces for this block. First, select the *Interfaces* tab:

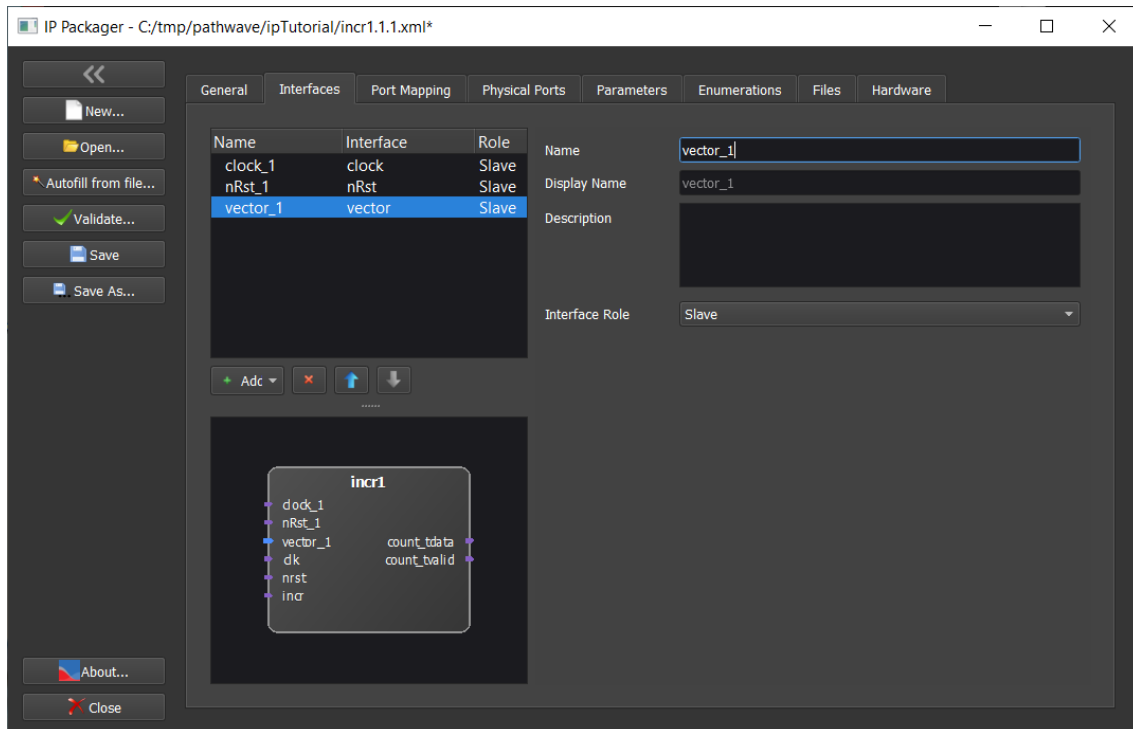


This is initially blank. We create interfaces in a similar way as we created ports, except we have to specify what kind of interface we're creating. We'll start with the clock interface. Click the **+Add** button and select *Clock* from the pull down menu to create the clock interface. You can't have an interface with the same name as an unassigned port, so we can't change the name of the interface to *clk* yet. For the moment, leave the name as *clock_1*. The *Interface Role* is somewhat analogous to *Direction* was for ports. Per the IP-XACT specification, a *Master* is the side of an interconnect that instigates transactions, while the *Slave* is the side of the interconnect that responds to transactions. For a simple interface such as a clock, an *Output* port would be the *Master*, and an *Input* port would be the *Slave*. For more complex interfaces, there may be multiple ports in the interface, and some may be *Output* ports and some may be *Input* ports. In this case, the clock interface would be a *Slave* interface, so we can leave the default *Interface Role* alone.

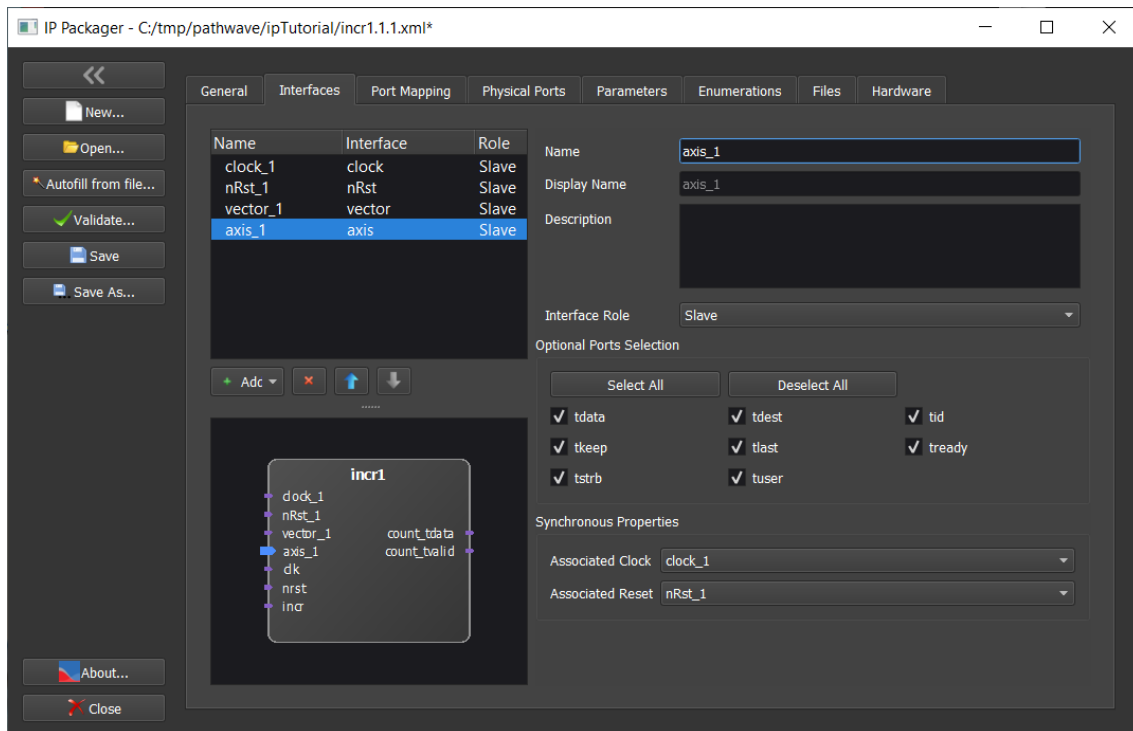




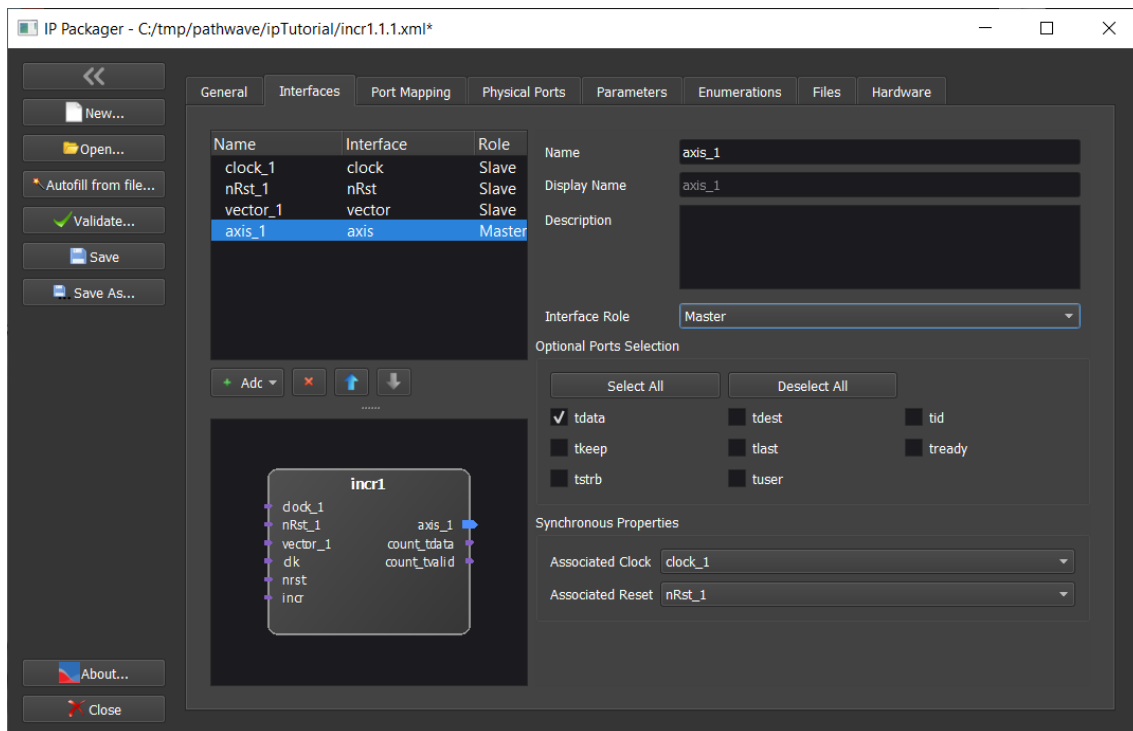
In the same way, create a *nRst* interface and a *vector* interface:



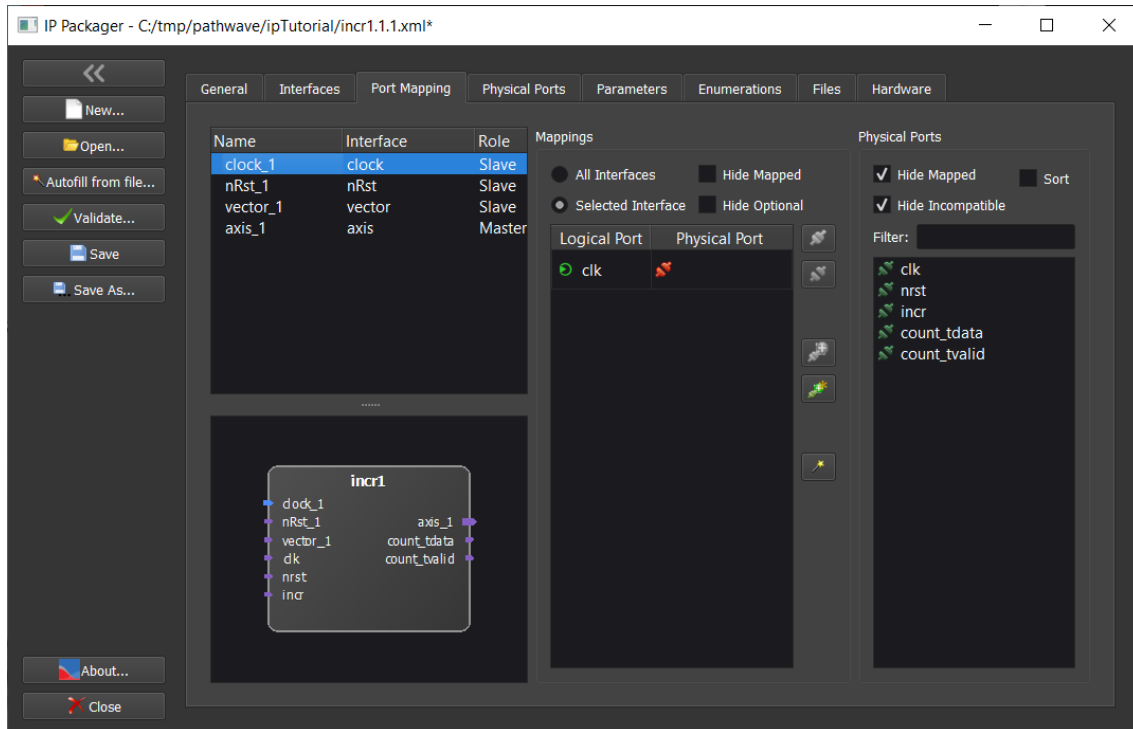
Now we'll create the Axi4-streaming interface for the *count* signals. Click the *+Add* button and select *axis* for the interface type. This interface has more options than the clock or vector interfaces had:



The Axi4-streaming specification has some required signals, such as the *tvalid* signal. It also has optional signals which may or may not be in that interface. These are specified in the *Optional Port Selection* pane. This interface only has the optional *tdata* signal. Click the *Deselect All* to clear all the check boxes and then check the *tdata* box. In this case, the IP block instigates transactions on the *count* interface, so change the *Interface Role* to *Master*. The Axi4-streaming specification requires a clock and reset to be associated with the interface. These are specified in the *Synchronous Properties* pane. Since there is only one clock and one reset in this design, the default values are fine. If there were more than one clock, you would pick which clock to use for this interface. Note: this means you should specify a clock and a reset interface prior to creating an Axi interface:

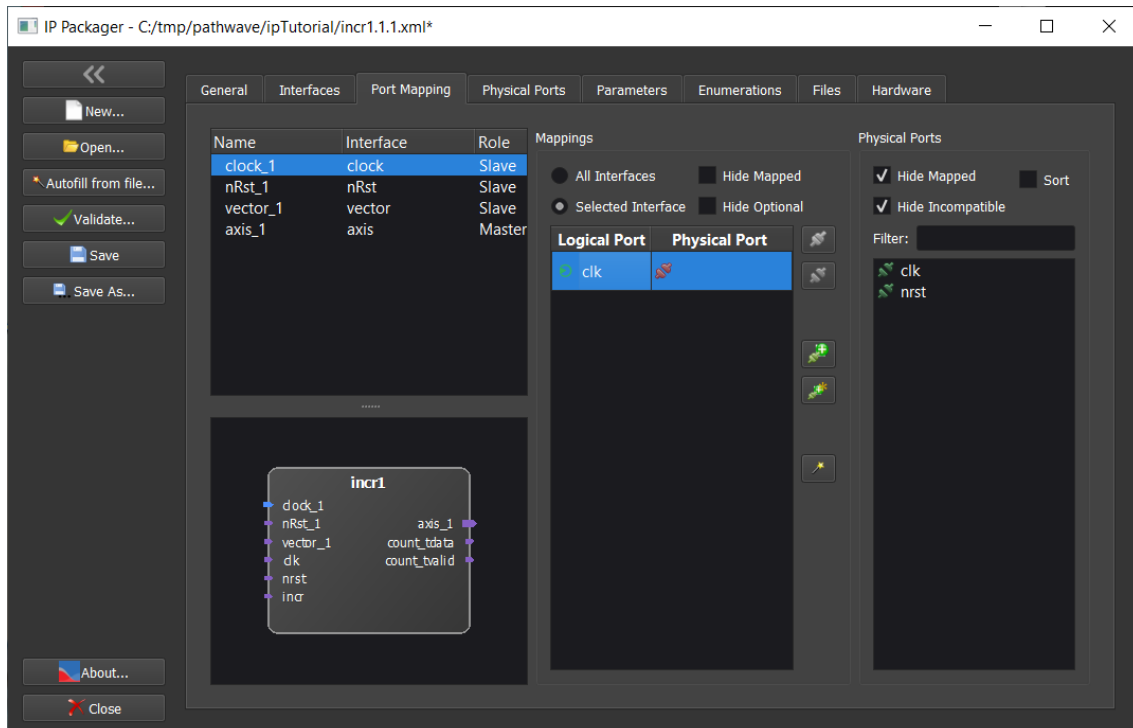


Now that all the ports and interfaces have been created, we need to map the logical ports of each interface to its physical port. Select the *Port Mapping* tab. We'll select each interface in turn and map its ports. First select the *clock_1* interface:

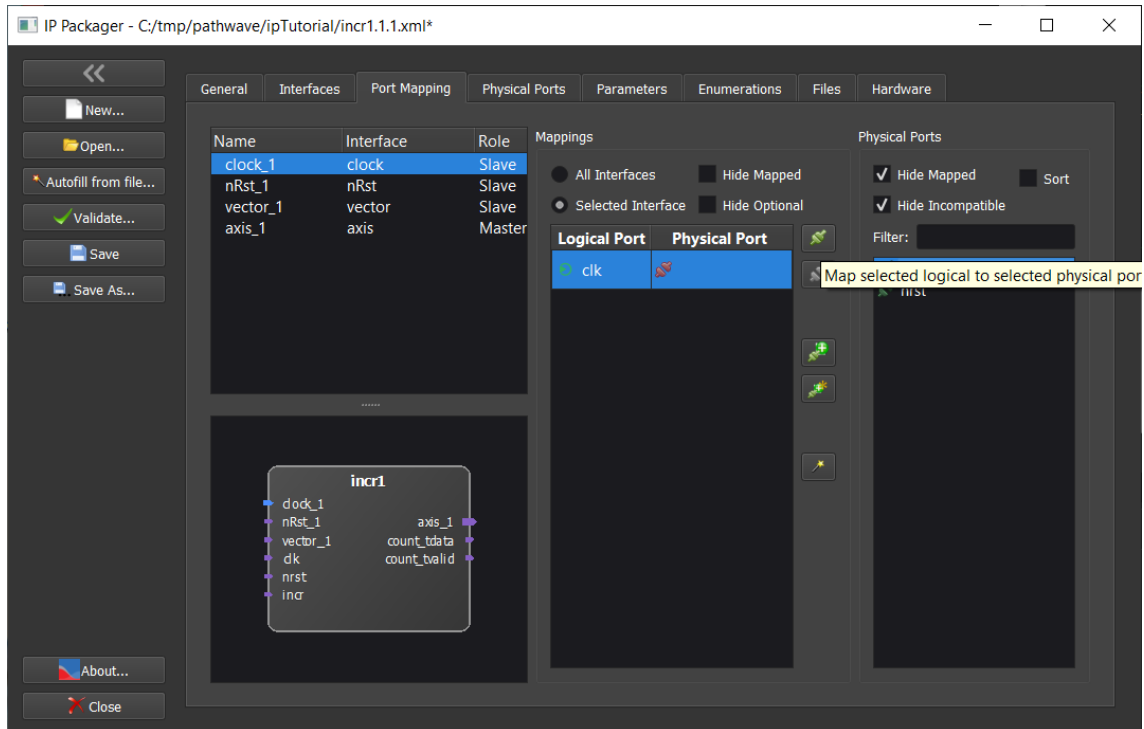


The middle pane shows the logical ports for that interface along with any mapping (since we haven't mapped it yet, the *Physical Port* column is blank). The right pane shows a list of possible physical ports.

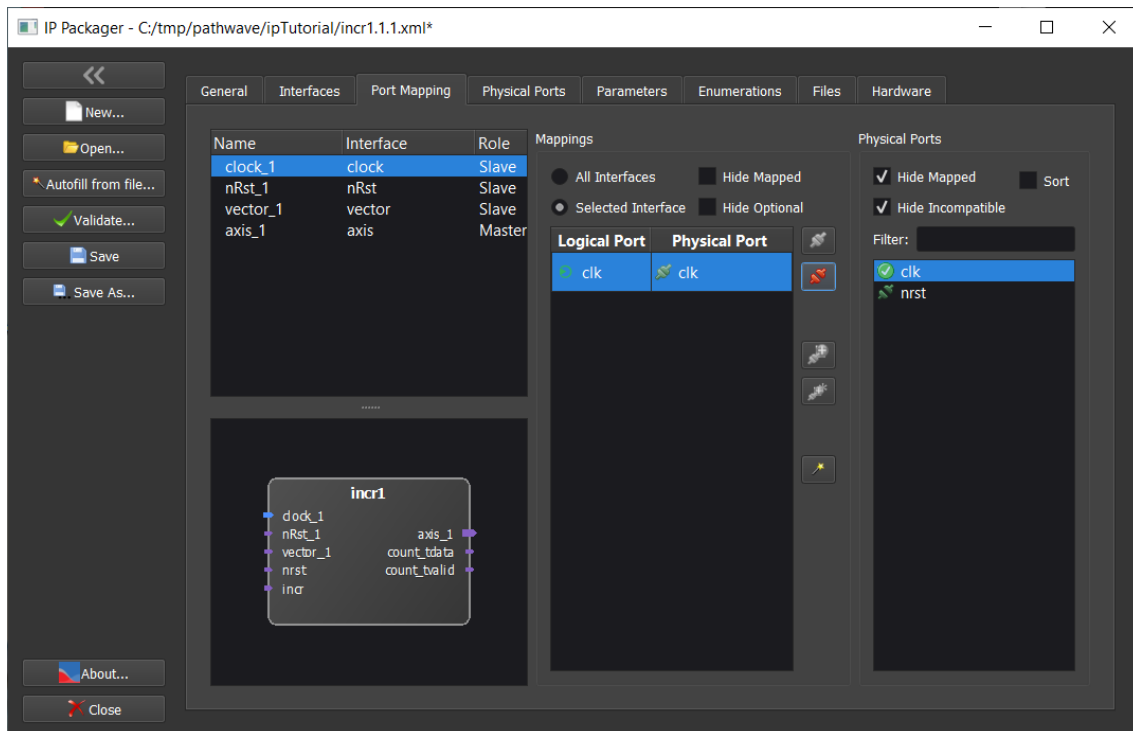
If you click on the *clk* logical port in the middle pane, you'll see the list of possible physical ports has been limited to compatible and unmapped ports. By default it won't show ports that aren't compatible or have already been used. If desired you can change these settings:



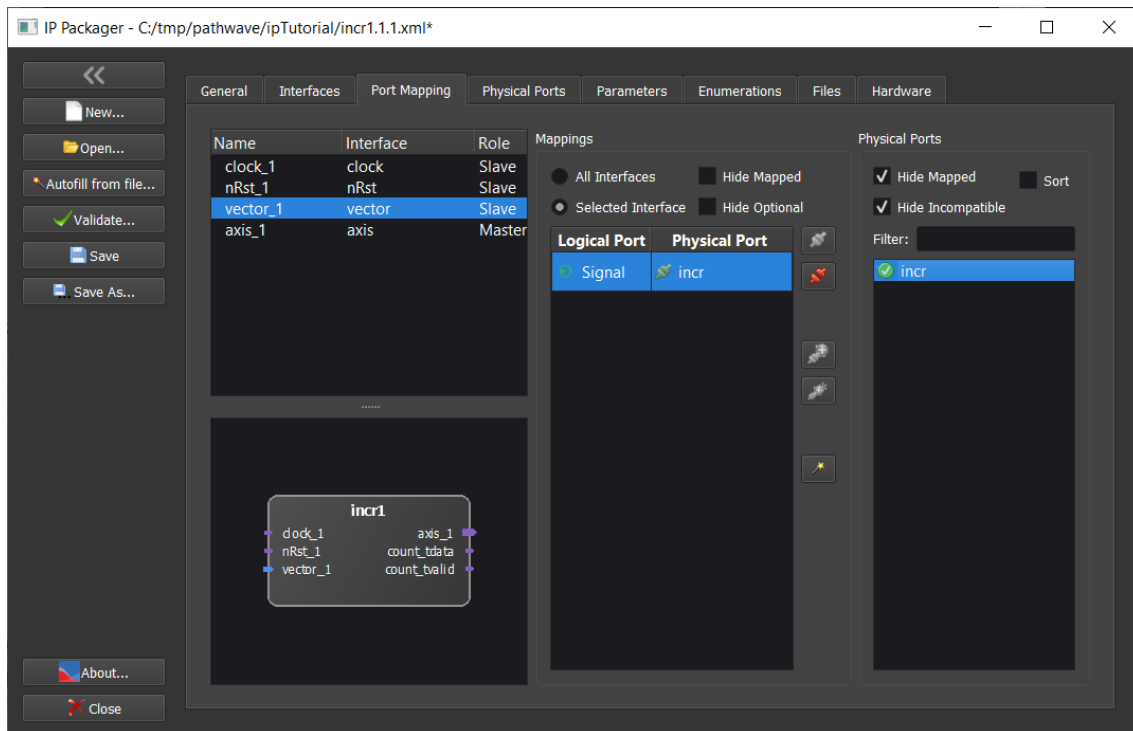
Select the *clk* physical port, then click the *Map* button. This is the green connection button that is the top button between the *Mappings* and *Physical Ports* panes:



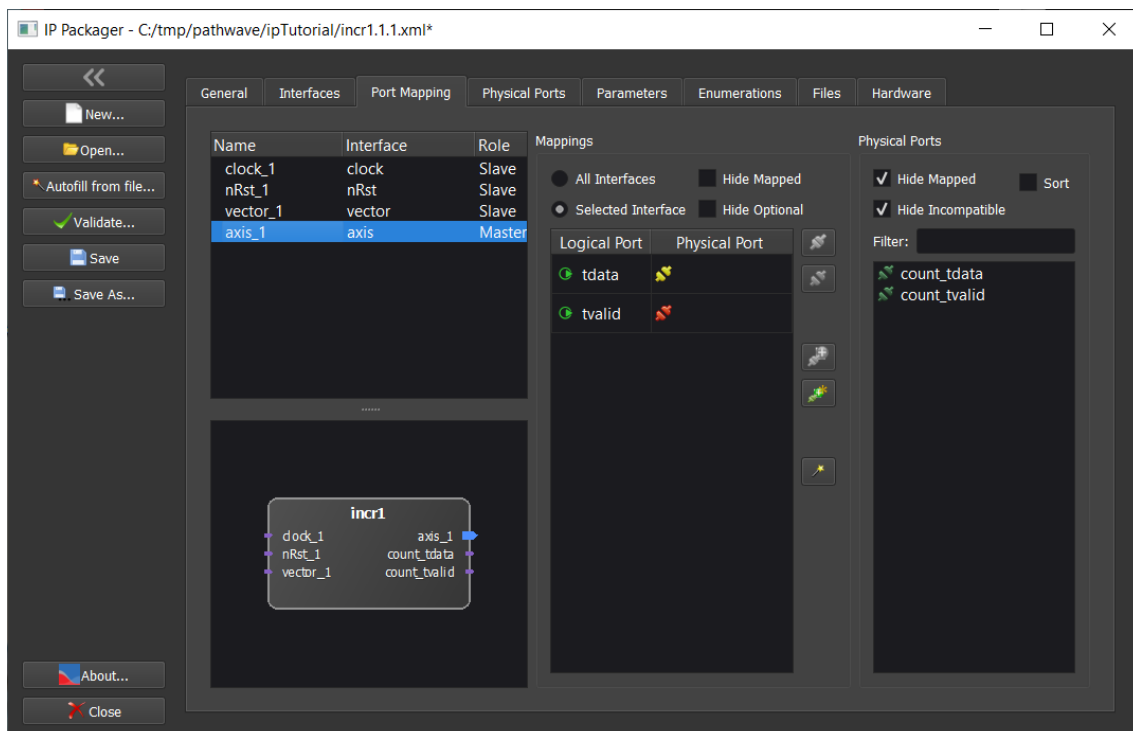
After mapping the port, you'll see it connected:



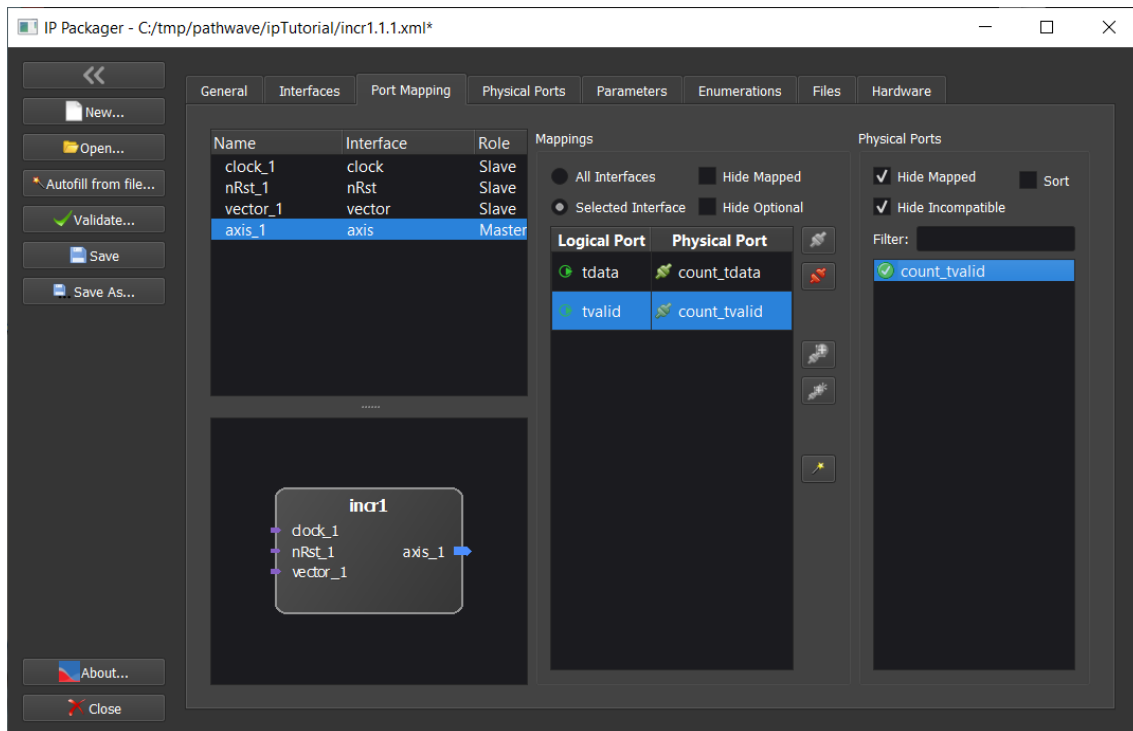
Select the *nRst_1* interface and map it to the *nrst* physical port, and likewise map *vector_1* to the *incr* port. Instead of using the *Map* button, you can alternately click on the *Logical Port* and then double click on the desired *Physical Port*.



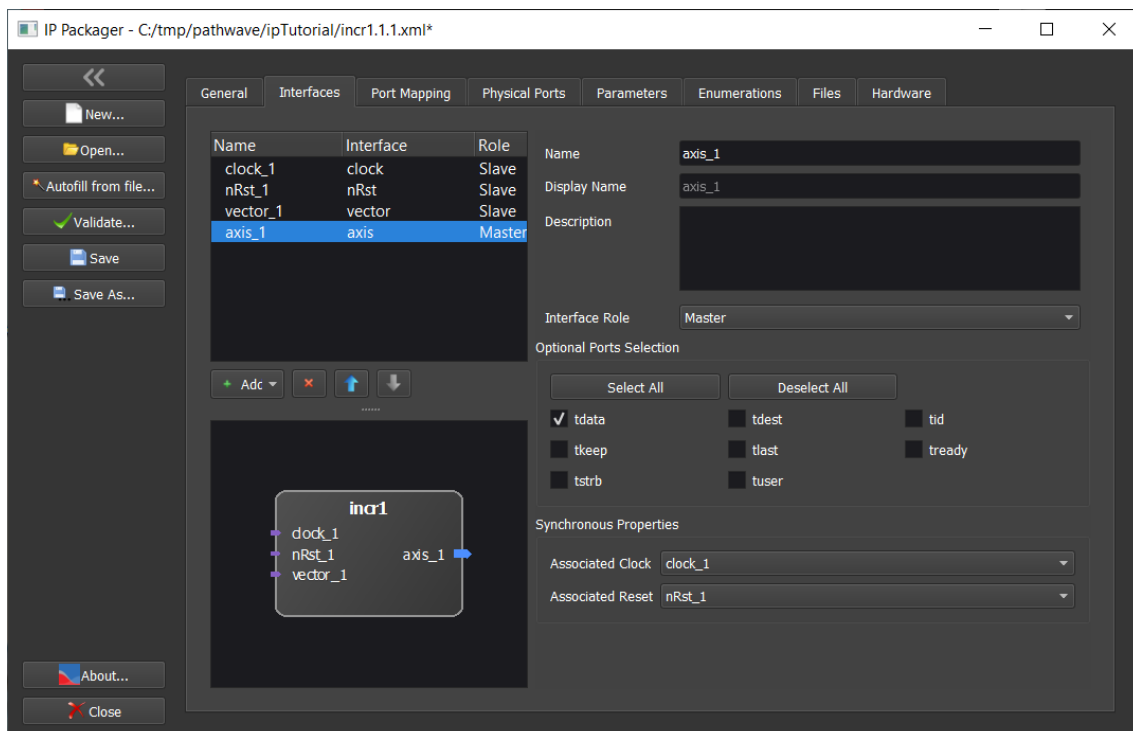
Now we want to map the *axis_1* interface. This one is a little different because it has two ports associated with the interface:



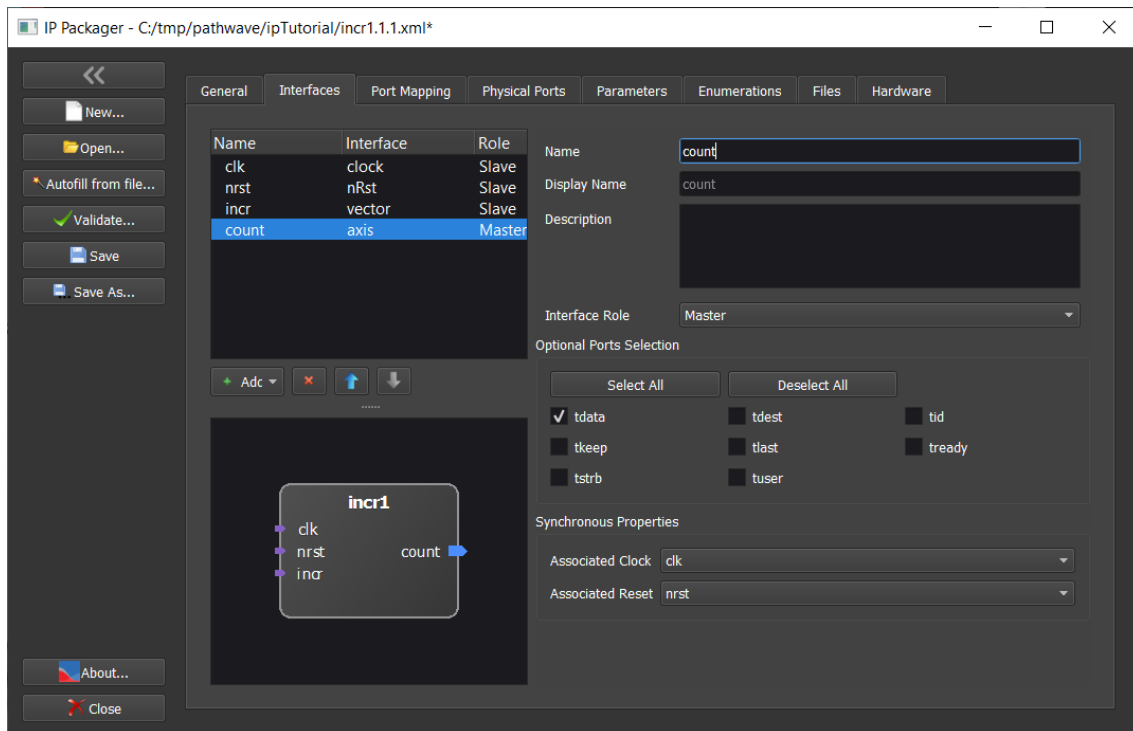
Mapping is done the same way, select the *tdata* logical port and map it to *count_tdata*, then select the *tvalid* logical port and map it to *count_tvalid*:



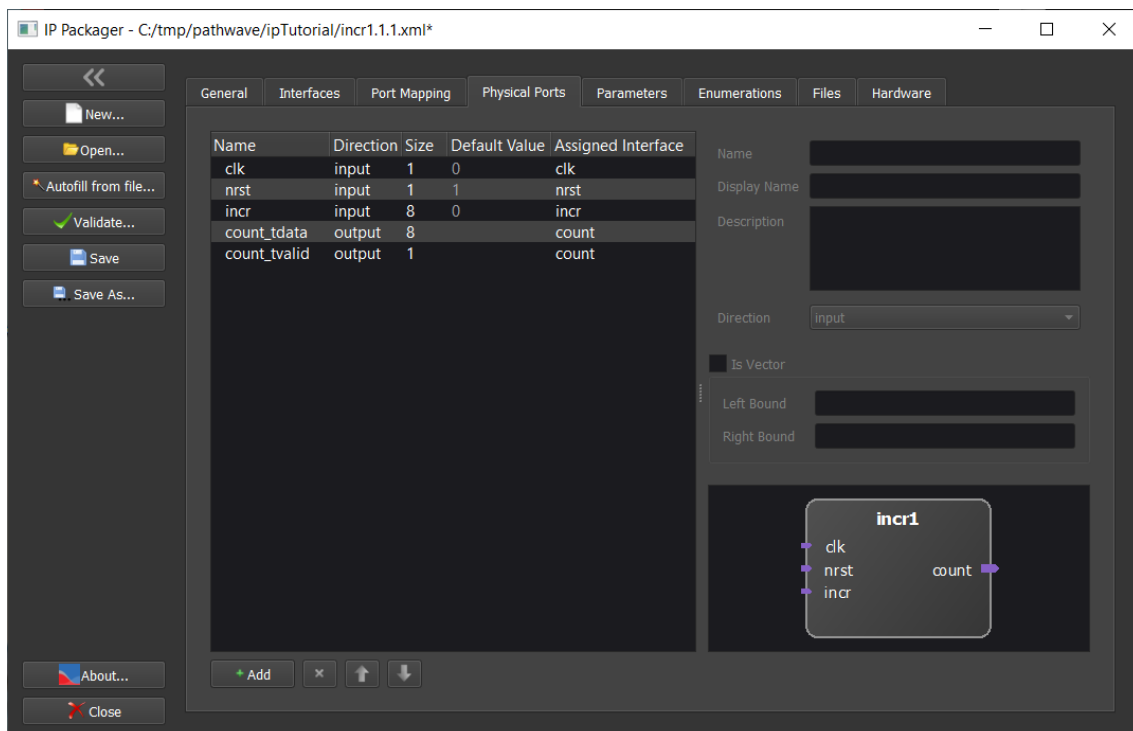
Now that all the ports have been mapped to interfaces, we can go back and rename the interfaces to our desired names. Select the *Interfaces* tab again:



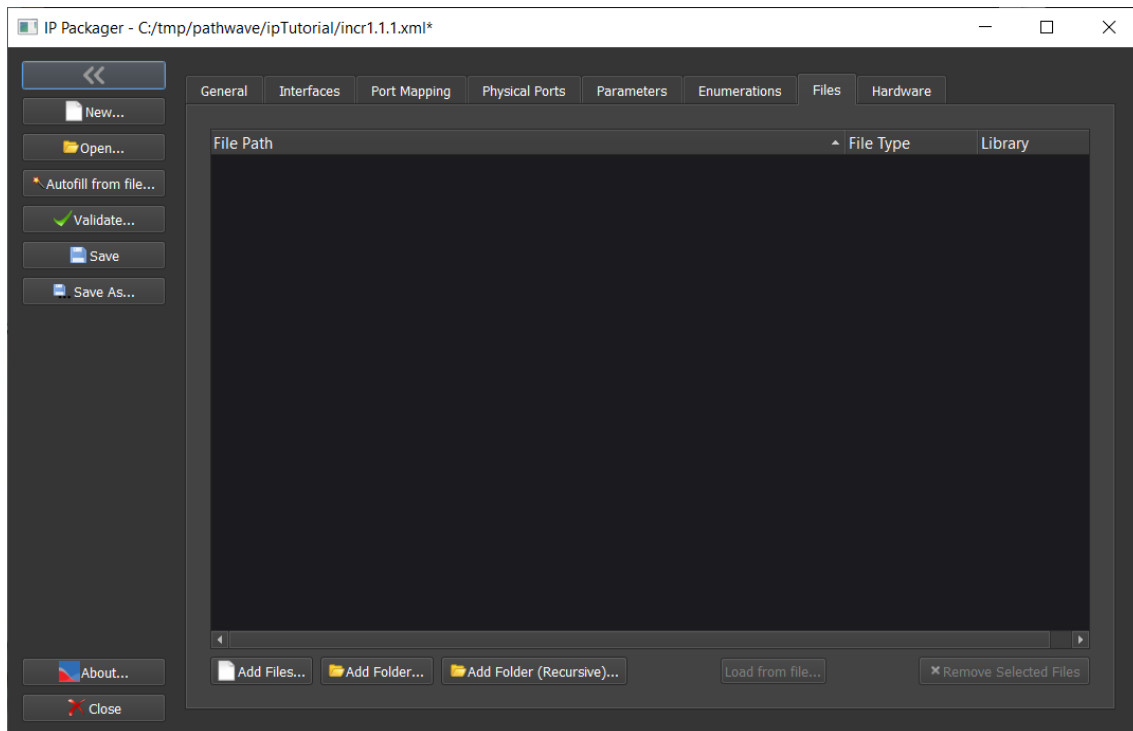
Select the *clock_1* interface and change its name to *clk*. Likewise change the other three interface names to *nrst*, *incr*, and *count*:



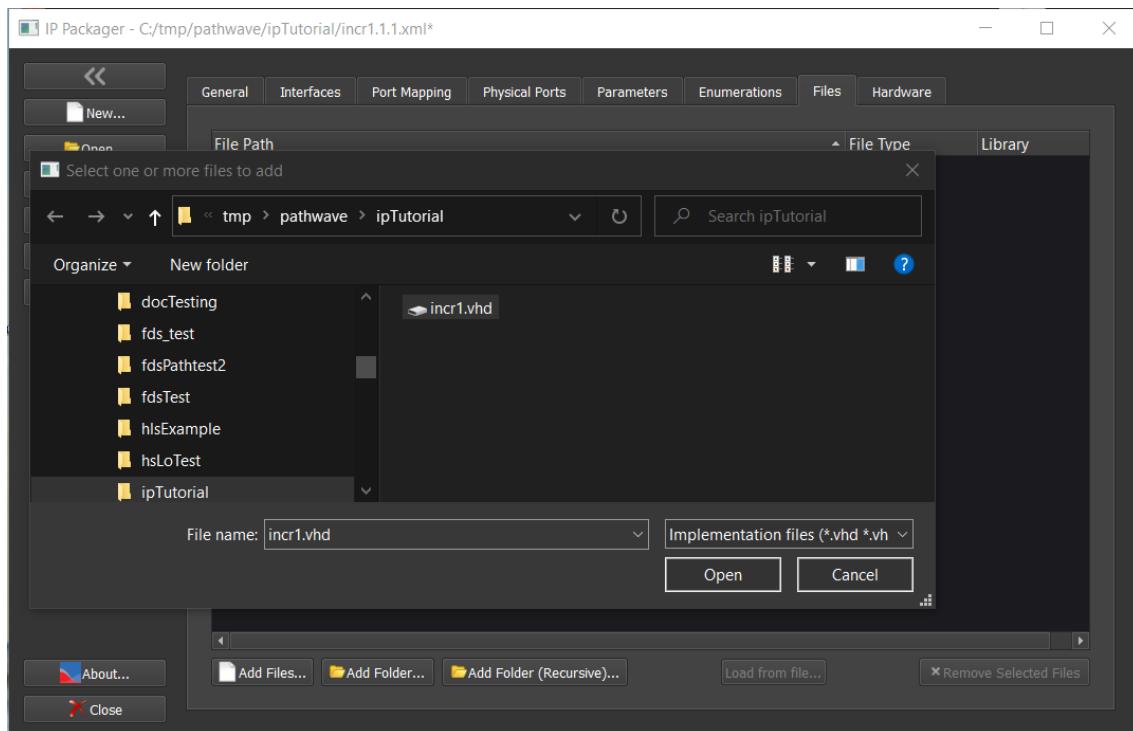
If you now look at the *Physical Ports* tab, you'll see that each port is now assigned to an interface:



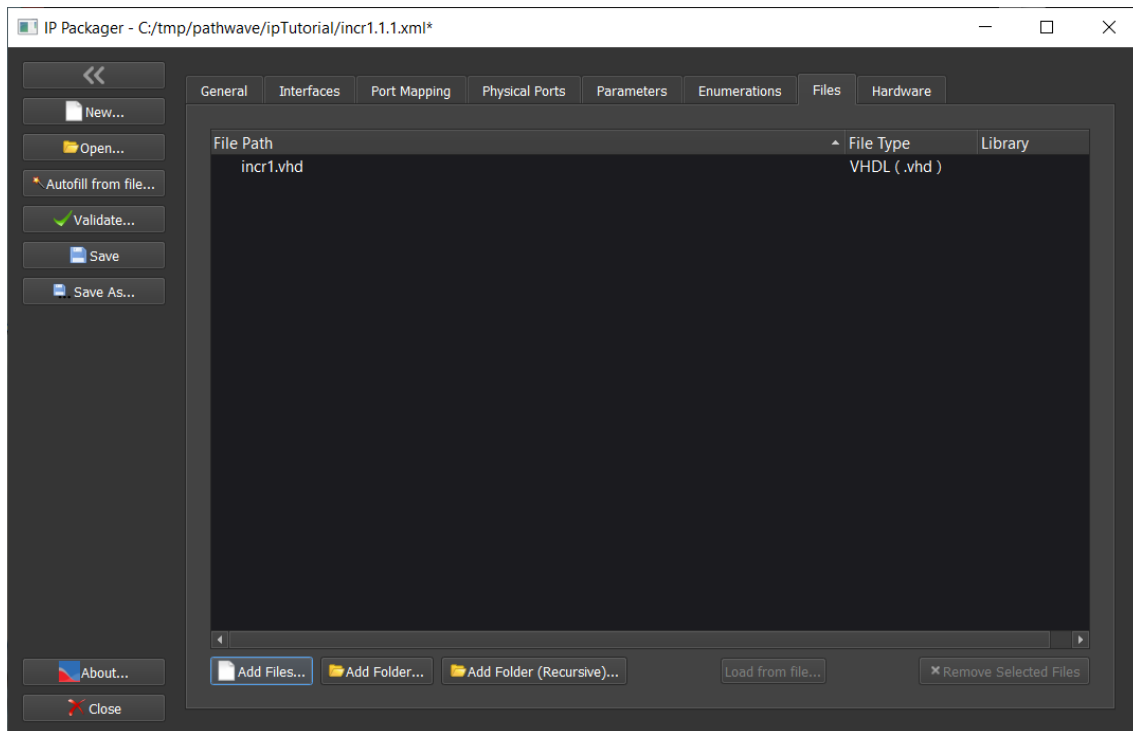
Finally, we have to define the files needed to build this IP block. Select the *Files* tab. Initially, no files are associated with this design yet:



Select *Add Files...* and navigate to, and select the *incr1.vhd* file:



After clicking *Open* we see:



For this design, this is all the files we need to add. For more complex hierarchical designs, we might need to add more files.

We are done for this module. Click *Save* and then *Close* to finish.

Simple HDL done automatically

It is very easy to package simple IP using the PathWave FPGA IP Packager. While all the information needed to package an IP block may be entered manually, and this will be shown in the following tutorial, it is far easier to let the IP Packager learn about the IP block by scanning its HDL source file. This will automatically fill in things like the module name, the I/O ports, the source file name, and if the port names follow the PathWave FPGA naming convention ([Infer Interface Reference](#)) the ports will automatically be mapped to the appropriate interfaces.

This tutorial will package the following simple block:

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

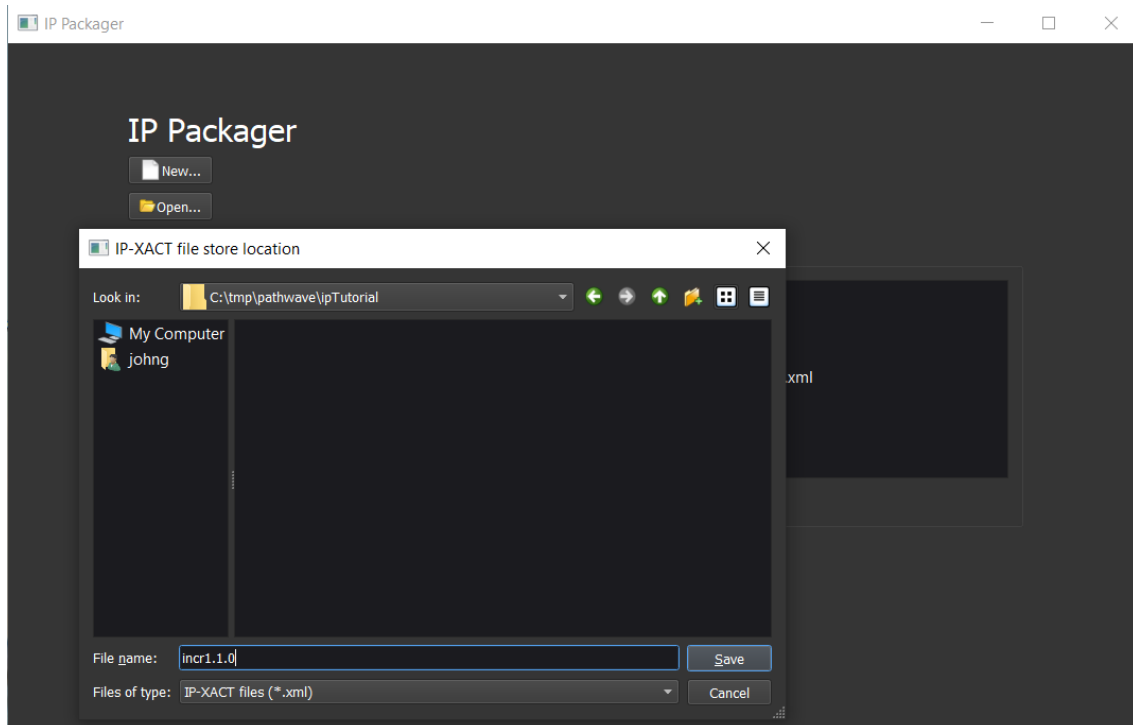
entity incr1 is
  port (
    clk    : in  std_logic;
    nrst   : in  std_logic;           -- Active low reset
    incr   : in  std_logic_vector(7 downto 0);
    count_tdata : out std_logic_vector(7 downto 0);
    count_tvalid : out std_logic
  );
end incr1;

architecture Behavioral of incr1 is
  signal count : std_logic_vector(7 downto 0);
begin -- Behavioral
  count_tdata <= count;
  count_tvalid <= '1' when (incr /= 0) else '0';
  process(clk)
  begin
    if (nrst = '0') then
      count <= (others => '0');
    else
      count <= count + incr;
    end if;
  end process;
end Behavioral;

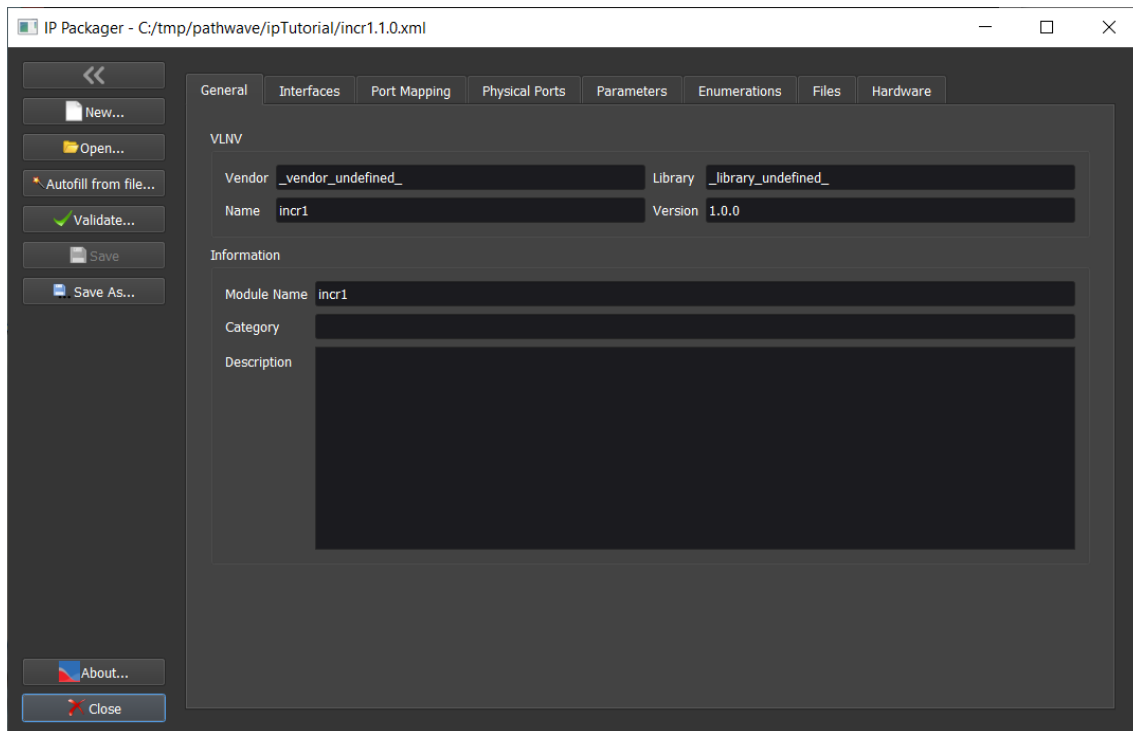
```

This block will increment an internal counter based on its incr input and output the counter value on an AXI-streaming count interface. It also has a clock input and an active low nrst input.

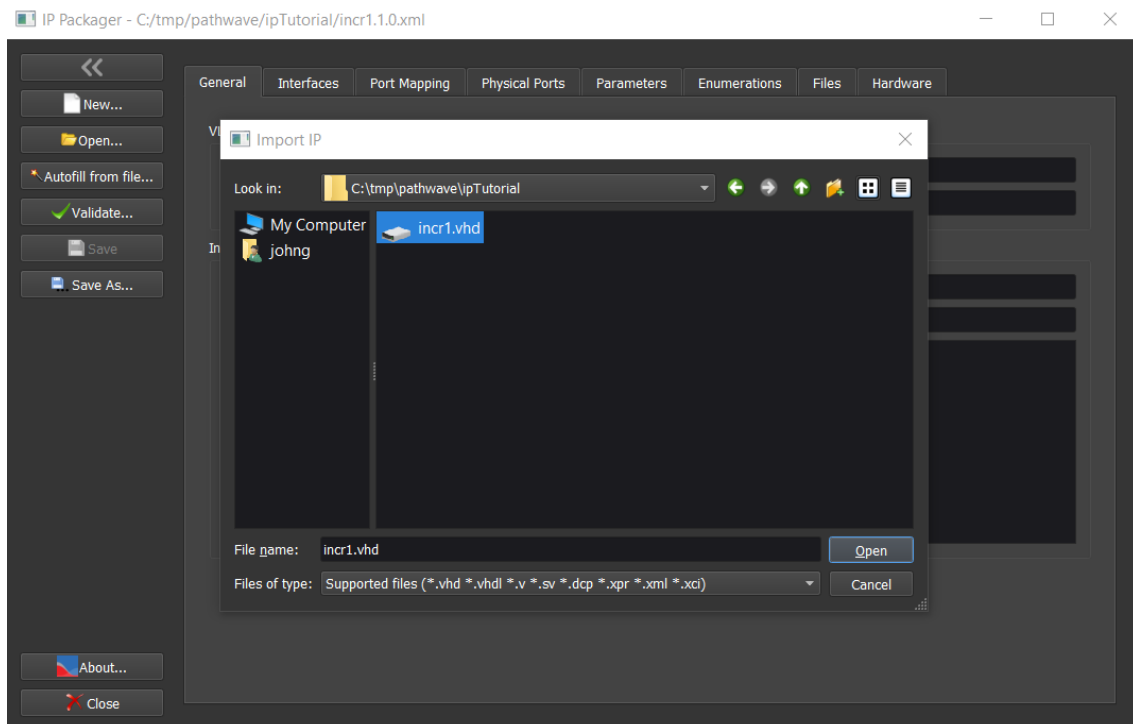
To package this IP, first start the IP Packager which is located in the Tools menu inside PathWave FPGA. Click the *New...* button and navigate to the desired location for the IP-XACT file, enter *incr1.1.0* for the name for the file, and click *Save*:



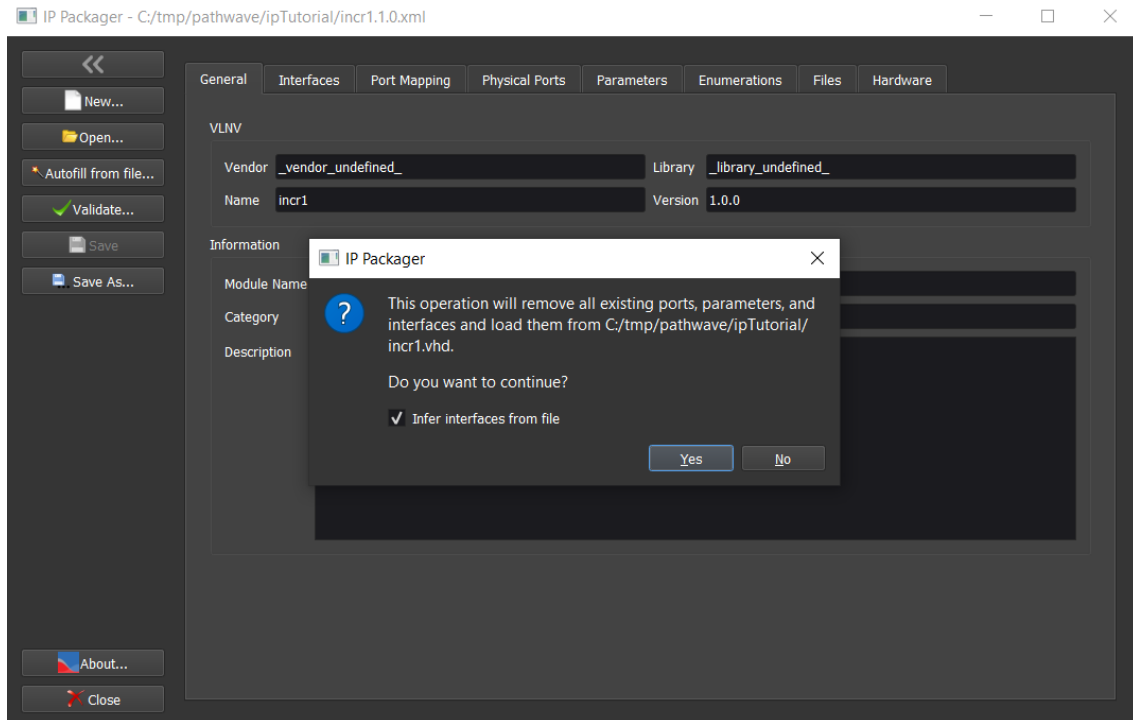
You now have a mostly empty description of the IP block:



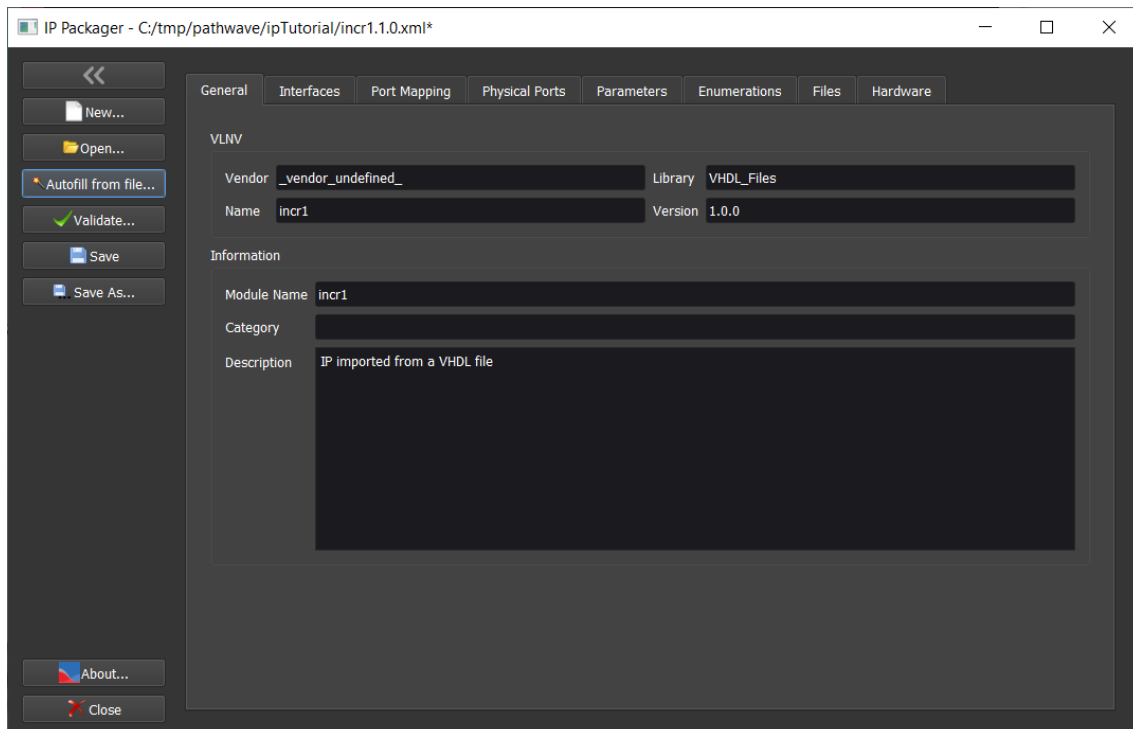
At this point you could start entering information manually (which is shown in the next tutorial), but the simpler way is to use the HDL source file to pre-fill out much of information. To do this, click the *Autofill from file...* button, navigate to IP's source HDL file, select it, and click *Open*:



The IP Packager will ask you to confirm the operation. Make sure that "Infer interfaces from file" is checked, and click "Yes":

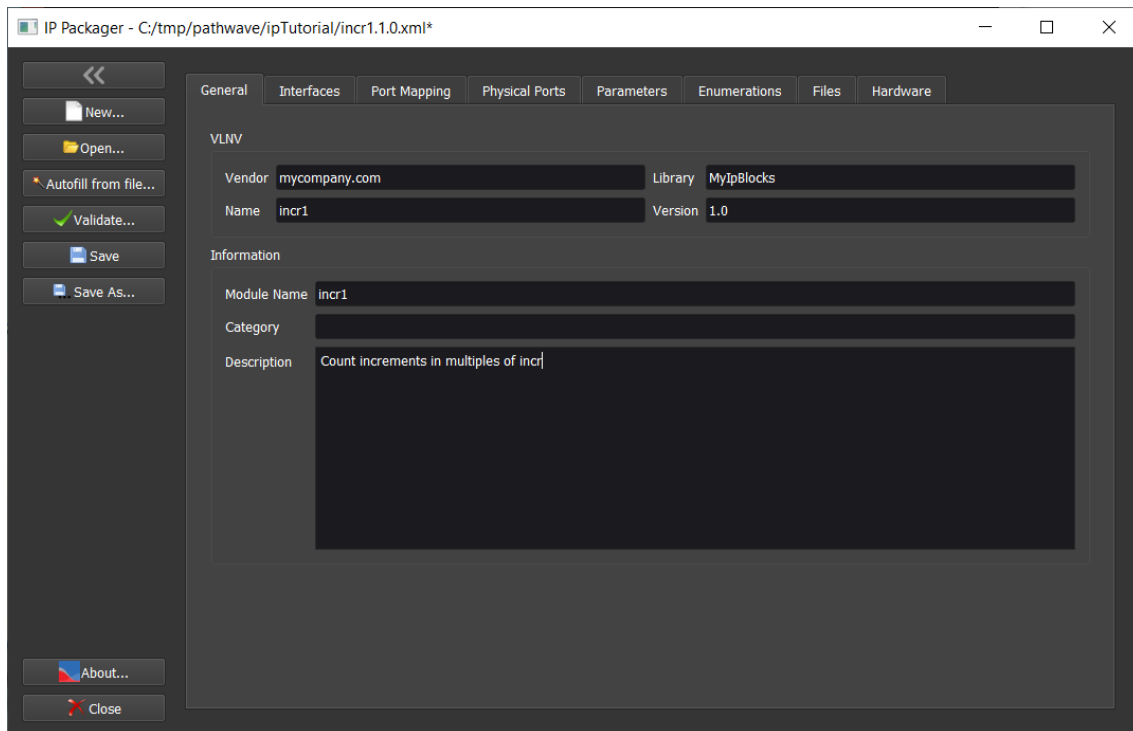


The IP Packager will read incr1.vhd and determine the ports and interfaces. This first tab looks much the same as it did before, but the other tabs now have been filled out:

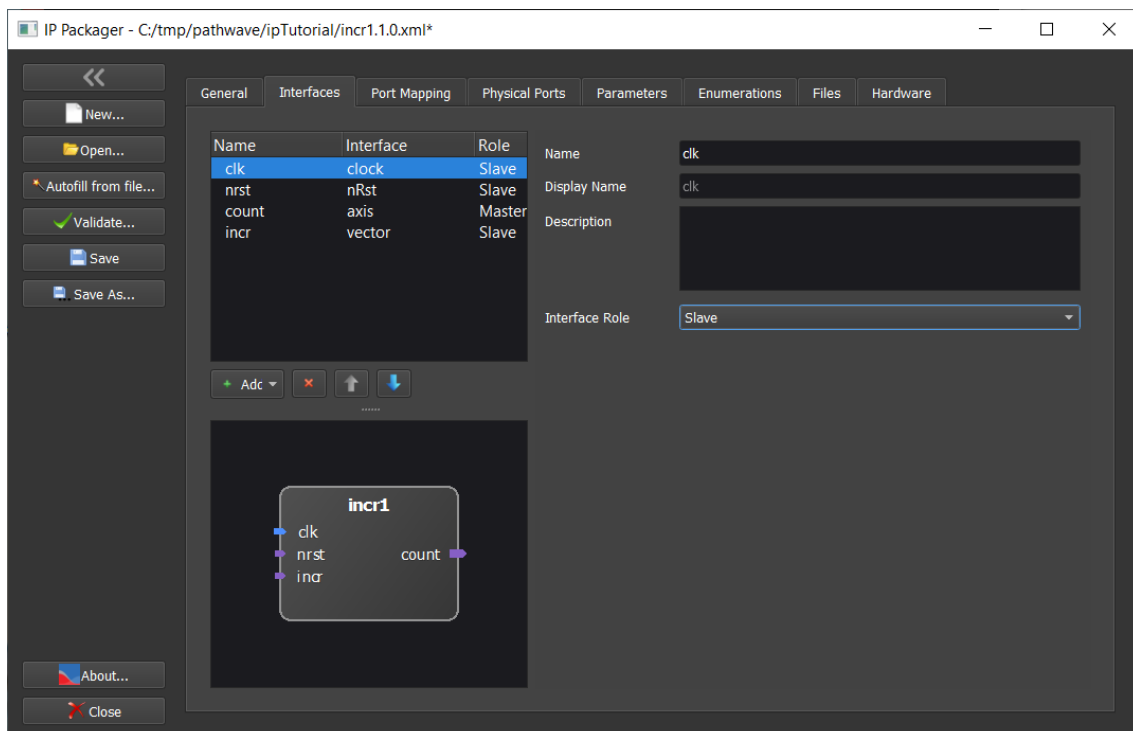


The VLNv (Vendor/Library/Name/Version) is used to identify the IP block and serves as a unique identifier. You can only have one block with the same combination of VLNv. The *Vendor* is typically the domain name of your company. The *Library* is how you want the IP organized. The *Name* is the name of the IP block (which does not have to exactly match the module name in the HDL). And the *Version* denotes the version for the IP block.

The *Module Name* must match the module's name in the HDL source code. The *Category* is an optional field if you want to subcategorize your IP finer than just with the *Library* field. The *Description* is shown to users when they add your IP block to their design and should give a brief summary of the block. Fill out these fields as shown below:



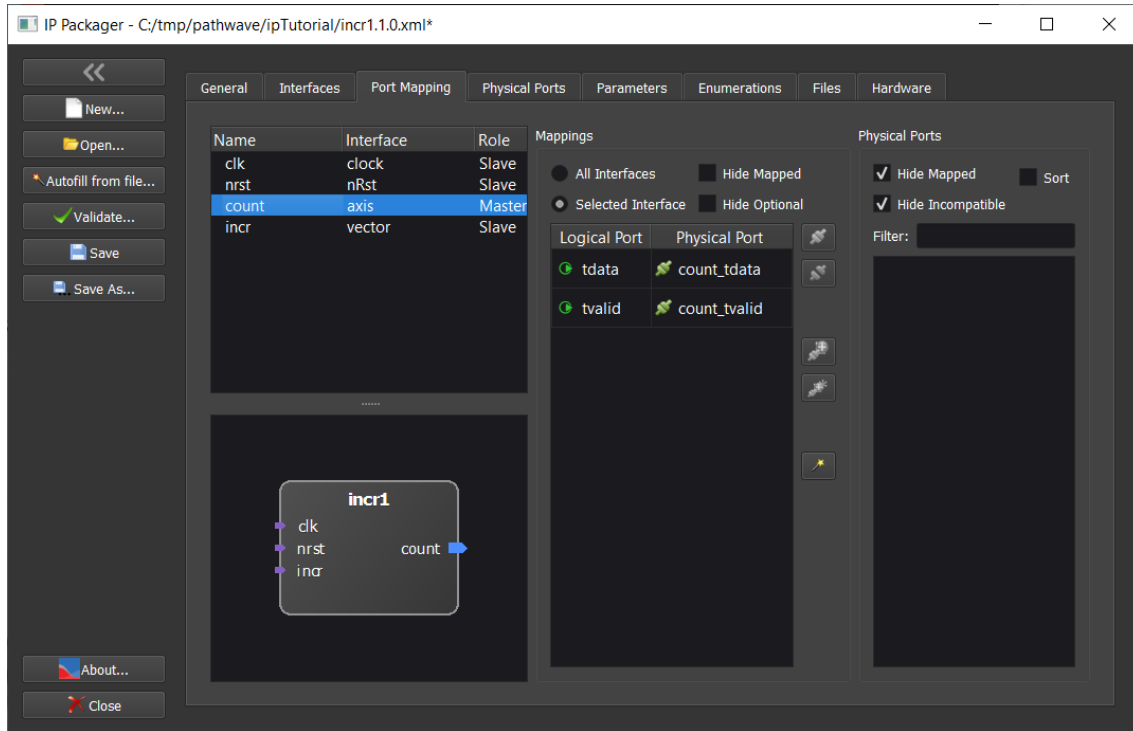
For this simple block, that's all you need to do (aside from saving it). Before we do that, let's explore some of the other tabs. The *Interfaces* tab shows the module's logical interfaces:



In this case, the tool found four interfaces, a clock interface, a reset interface, an Axi4-streaming interface, and a vector interface.

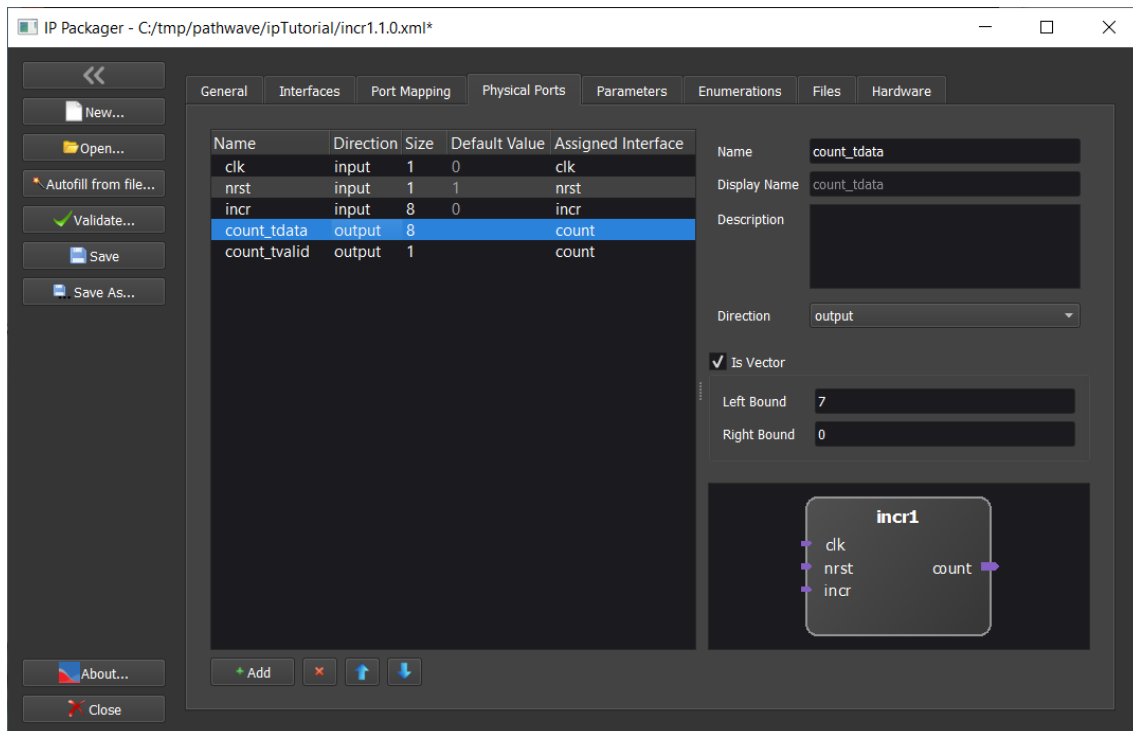
If desired, you can change the *Display Name* to something other than the interface name to make it more descriptive. You may also enter a *Description* for the interface to help explain to users what the interface is used for.

By clicking on the *Port Mapping* tab and selecting one of the interfaces, you can see how the logical ports of the interface map to the physical ports in the hardware:



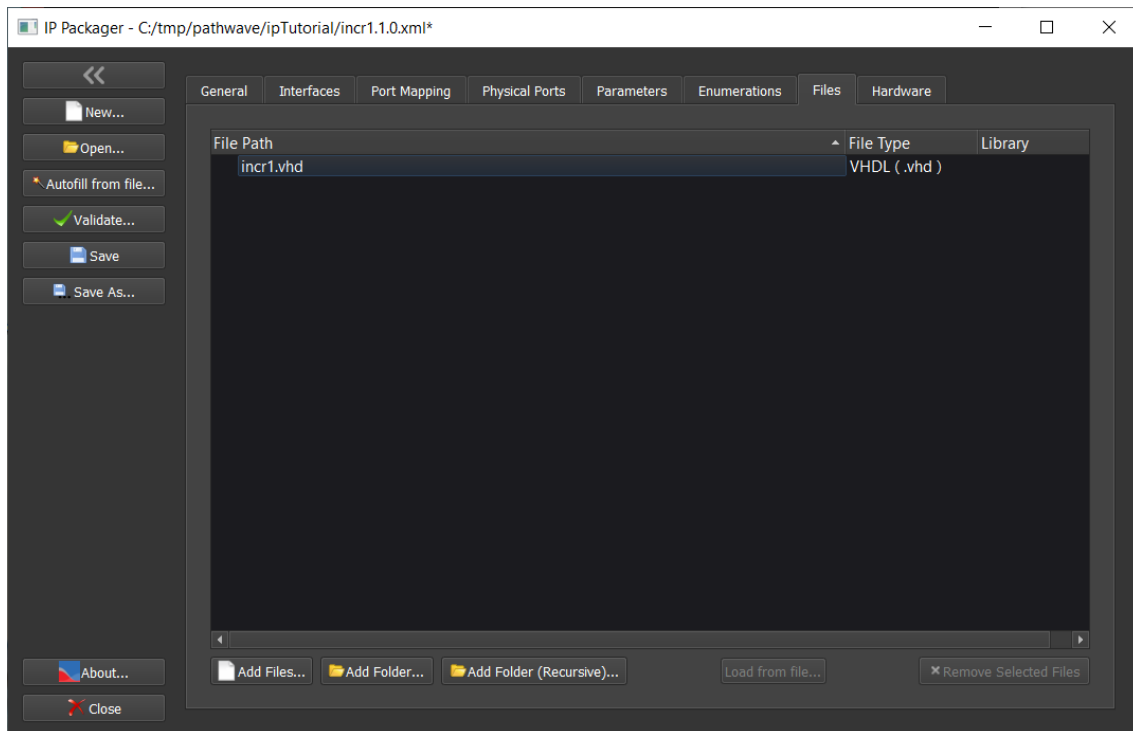
This port mapping was able to be inferred from the HDL since the port names followed the PathWave FPGA naming convention.

Clicking on the *Physical Ports* tab shows the physical ports for the module:



Note that the reason the IP Packager could figure out the Axi4-streaming interface *count* is because the ports associated with this interface followed the convention of `<interfaceName>_tdata` and `<interfaceName>_tvalid`. If desired, the *Display Name* can be changed and a *Description* added just as in the *Interfaces* tab.

This design doesn't have any parameters, so that tab will be blank. The *Files* tab will only have the one source file `incr1.vhd`:



Click **Save** to save the IP-XACT file, then close the module and exit the IP Packager. That's all there was to packaging this simple example.

Parameterized HDL

IP blocks can be made more generalized and easier to be used through the use of *parameters* (called *generics* in VHDL). These are values that are specified when the IP block is instantiated and can be used to customize the block. This allows one IP block to fill more needs than a single non-parameterized block would. For example, instead of requiring multiple IP blocks to support adders of different sizes, one adder block can be parameterized so that the size of the adder can be specified when the block is used.

This tutorial uses a block similar to that which was used in the earlier tutorials [Simple HDL done automatically](#) with the difference being that this block uses two parameters, *width* which specifies the bit width of the block, and *dir* which specifies whether the block increments or decrements. The process for creating the IP-XACT file is very similar to the case for non-parameterized IP blocks with a few steps added towards the end.

This tutorial will package the following block:

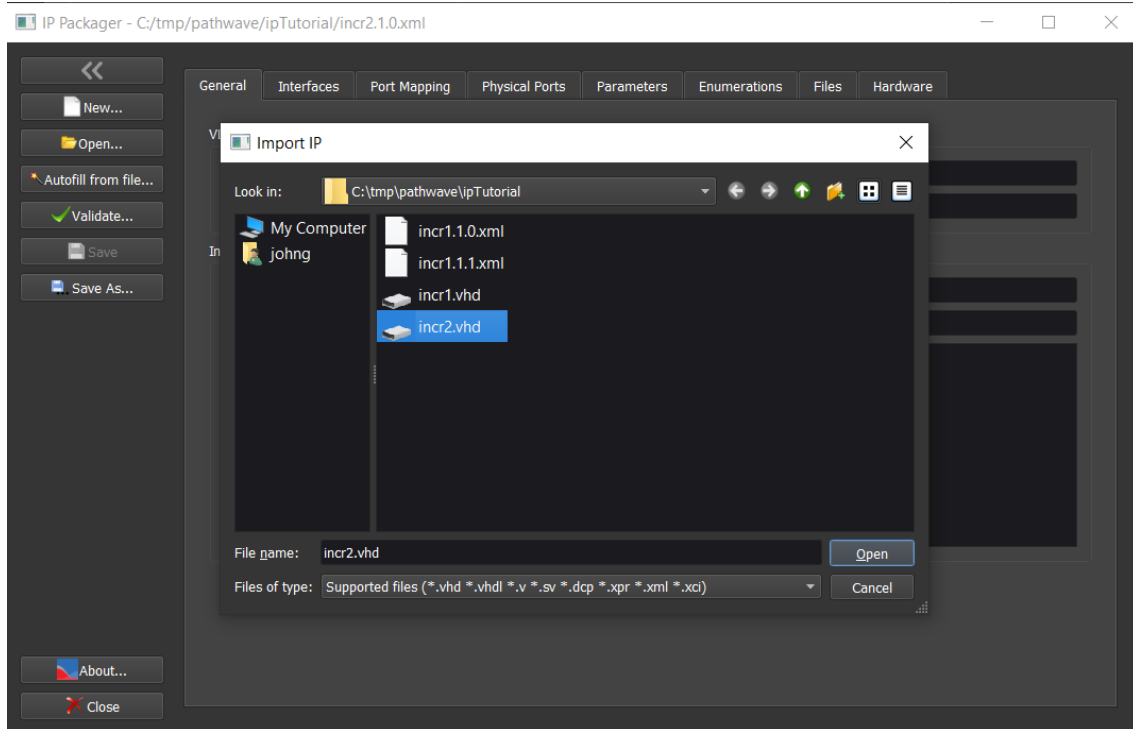
```

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
entity incr2 is
  generic (
    width : integer := 8;
    dir : integer := 0); -- Direction : 0 = up, 1 = down
  port (
    clk : in std_logic;
    nrst : in std_logic; -- Active low reset
    incr : in std_logic_vector(width-1 downto 0);
    count_tdata : out std_logic_vector(width-1 downto 0);
    count_tvalid : out std_logic
  );
end incr2;
architecture Behavioral of incr2 is
  signal count : std_logic_vector(width-1 downto 0);
begin -- Behavioral
  count_tdata <= count;
  count_tvalid <= '1' when (incr /= 0) else '0';
  process(clk)
  begin
    if (nrst = '0') then
      count <= (others => '0');
    else
      if (incr = 1) then
        count <= count + incr;
      else
        count <= count - incr;
      end if;
    end if;
  end process;
end Behavioral;

```

To package this IP, first start the IP Packager which is located in the Tools menu inside PathWave FPGA. Click the *New...* button and navigate to the desired location for the IP-XACT file, enter *incr2.1.0* for the name for the file, and click *Save*.

At this point you could start entering information manually (which is shown in the next tutorial), but the simpler way is to use the HDL source file to pre-fill out much of information. To do this, click the *Autofill from file...* button, navigate to the IP's source HDL file, select it, and click *Open*:

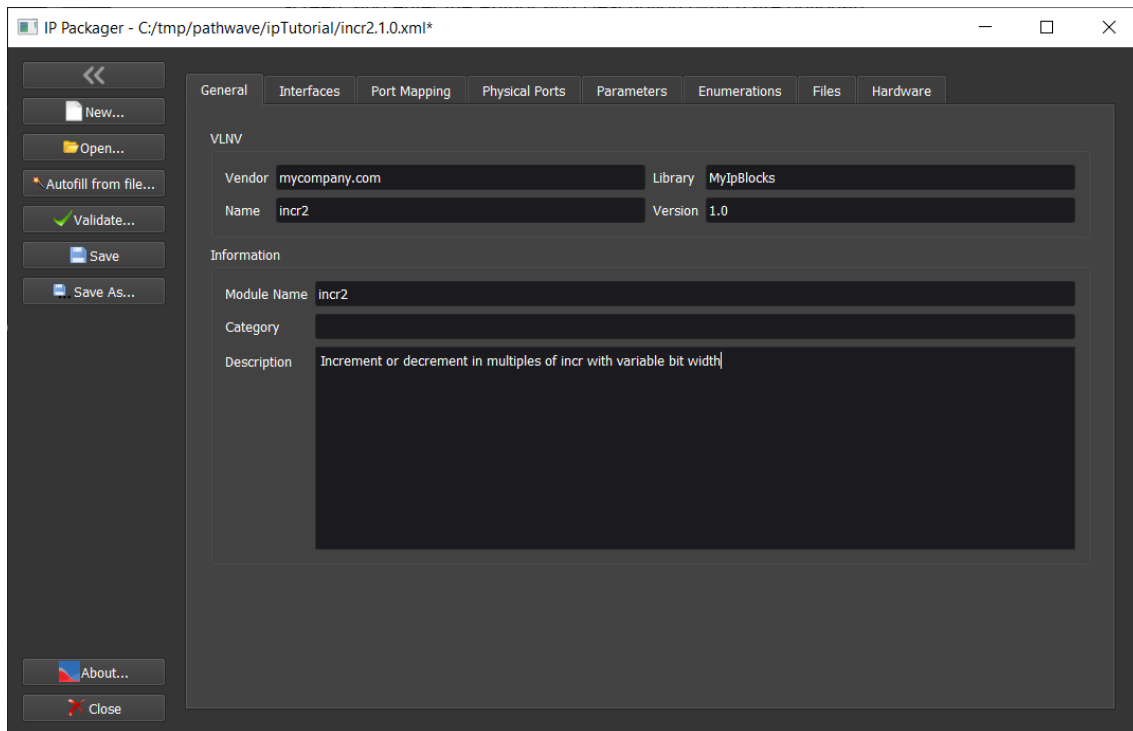


The IP Packager will ask you to confirm the operation. Make sure that "Infer interfaces from file" is checked, and click "Yes".

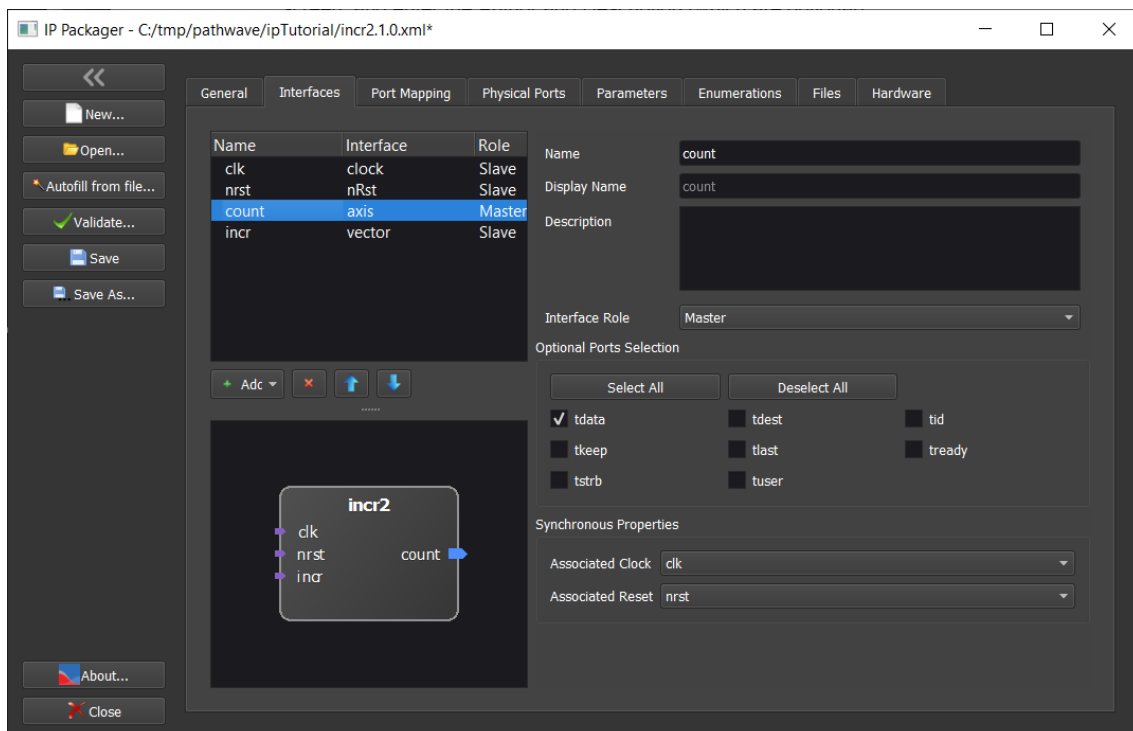
The IP Packager will read `incr2.vhd` and determine the ports and interfaces. First fill out the *General* tab with the identity of the IP.

The VLVN (Vendor/Library/Name/Version) is used to identify the IP block and serves as a unique identifier. You can only have one block with the same combination of VLVN. The *Vendor* is typically the domain name of your company. The *Library* is how you want the IP organized. The *Name* is the name of the IP block (which does not have to exactly match the module name in the HDL). And the *Version* denotes the version for the IP block.

The *Module Name* must match the module's name in the HDL source code. The *Category* is an optional field if you want to subcategorize your IP finer than just with the *Library* field. The *Description* is shown to users when they add your IP block to their design and should give a brief summary of the block. Fill out these fields as shown below:



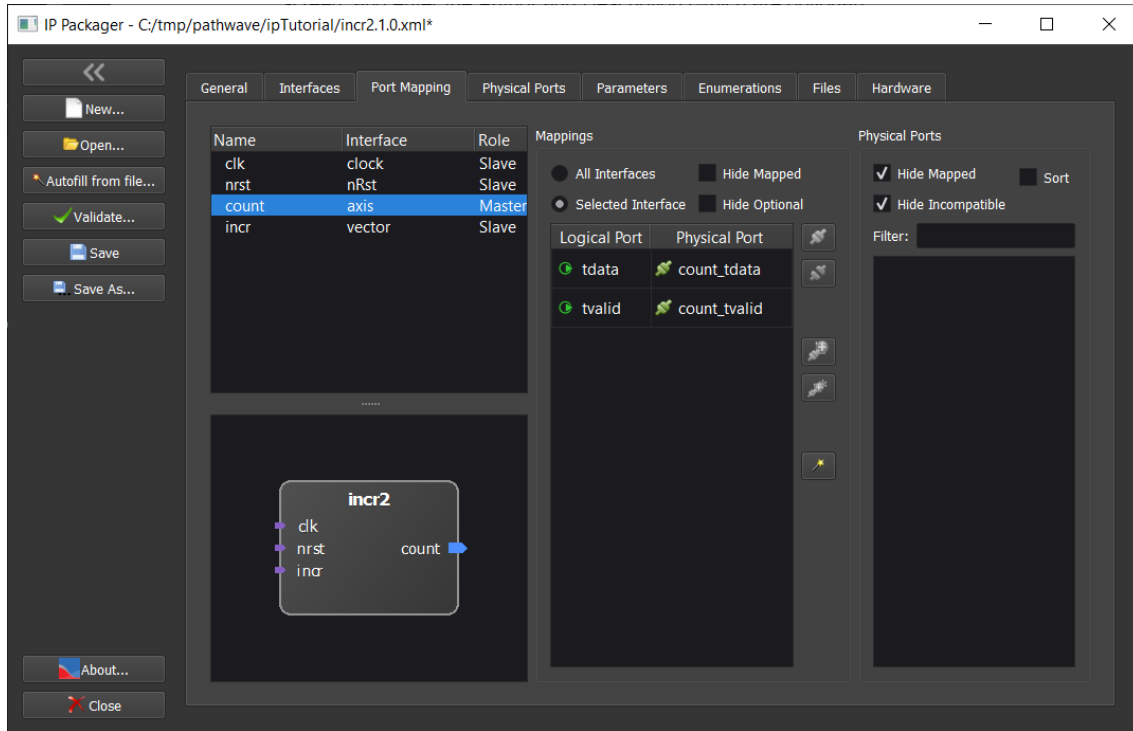
As in the earlier tutorial, the IP Packager determines the ports and interfaces from the HDL source file. The *Interfaces* tab shows the module's logical interfaces:



In this case, the tool found four interfaces, a clock interface, a reset interface, an AXI4-streaming interface, and a vector interface.

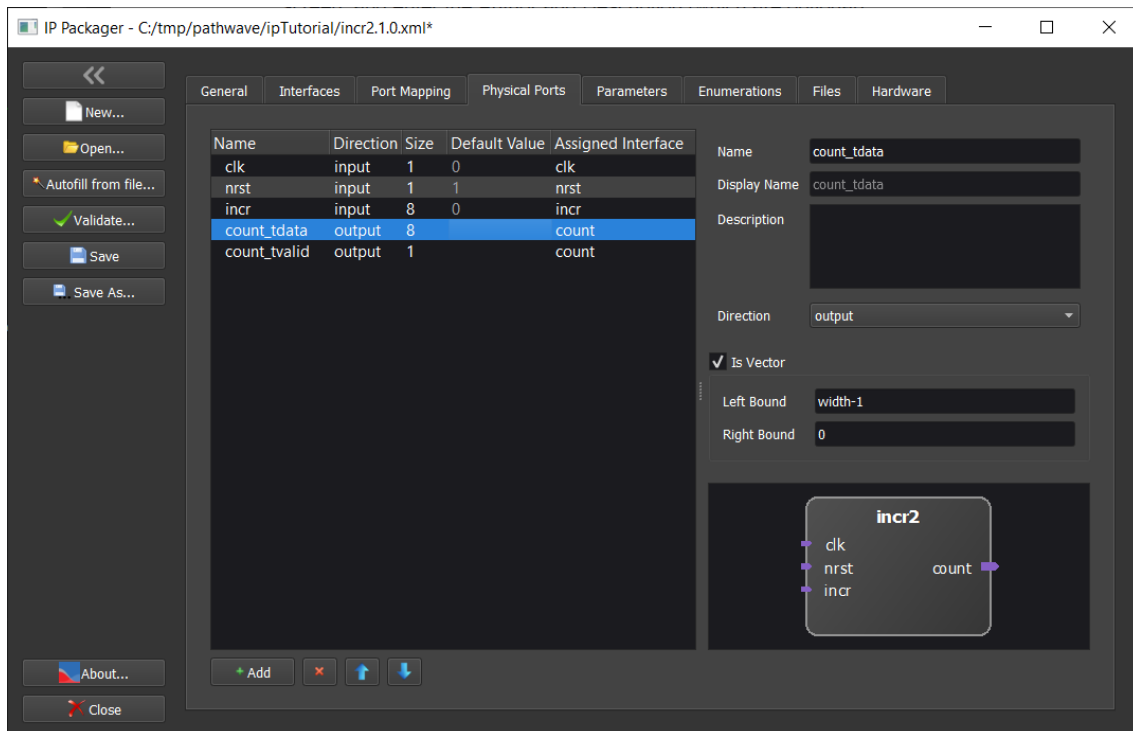
If desired, you can change the *Display Name* to something other than the interface name to make it more descriptive. You may also enter a *Description* for the interface to help explain to users what the interface is used for.

By clicking on the *Port Mapping* tab and selecting one of the interfaces, you can see how the logical ports of the interface map to the physical ports in the hardware:



This port mapping was able to be inferred from the HDL since the port names followed the PathWave FPGA naming convention.

Clicking on the *Physical Ports* tab shows the physical ports for the module:

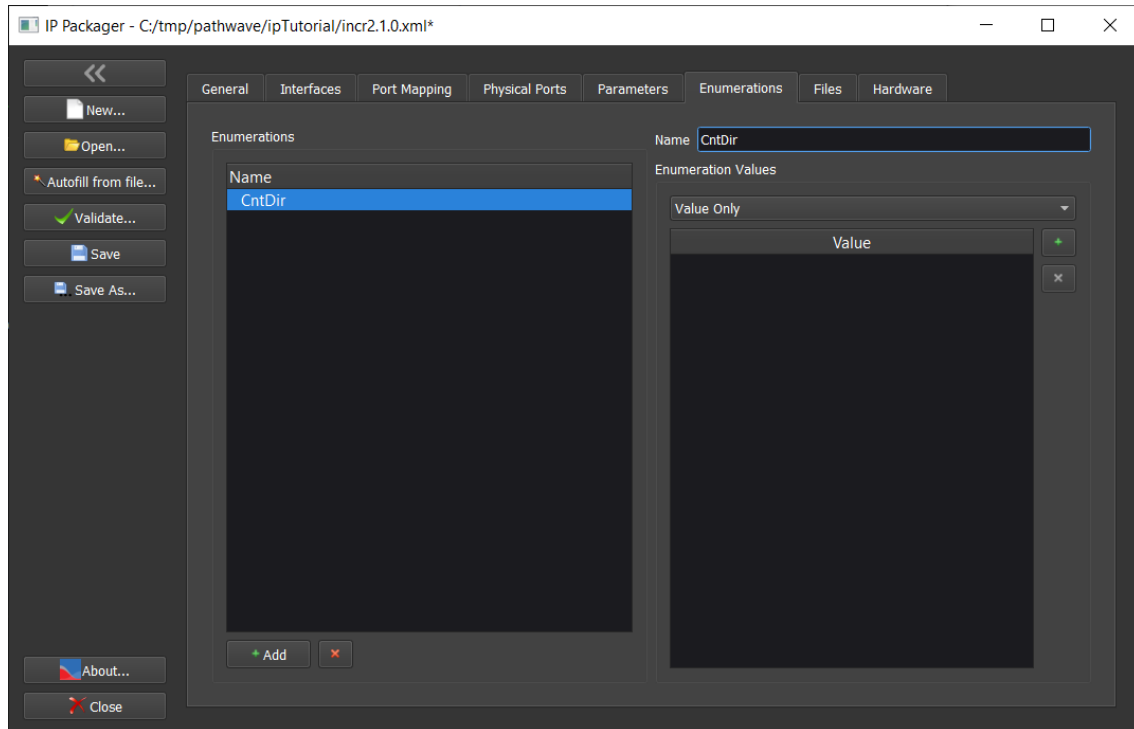


This tab is a bit different than that in the earlier tutorial. Notice that instead of the *Left Bound* being a number, it is the expression *width-1*, where (as we'll see in a moment) *width* is a parameter.

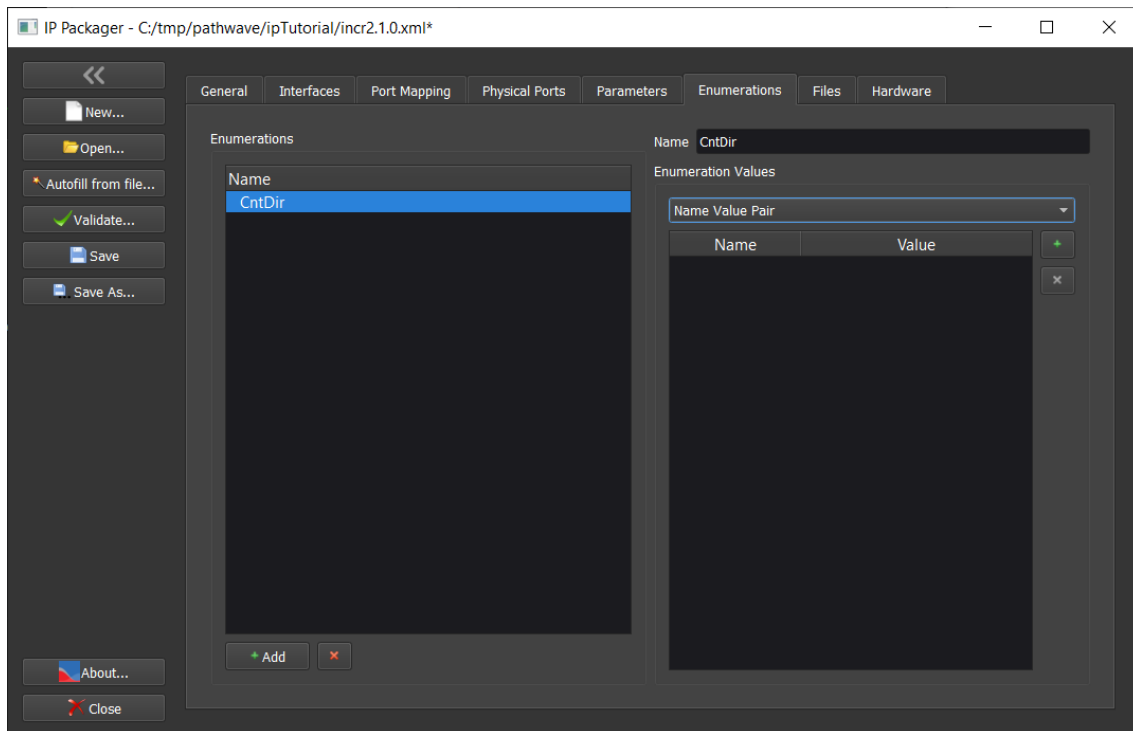
Note that the reason the IP Packager could figure out the Axi4-streaming interface *count* is because the ports associated with this interface followed the convention of `<interfaceName>_tdata` and `<interfaceName>_tvalid`. If desired, the *Display Name* can be changed and a *Description* added just as in the *Interfaces* tab.

Now let us consider the parameters for this block. There are several types of parameter. A parameter can be an integer, optionally with bounds to limit what values a user may enter. For example, it doesn't make sense for a bit width to be -2 bits. A parameter may be a single bit or boolean. This would show up in the PathWave FPGA IP Packager as a check box, with unchecked being 0, and checked being 1. A parameter may also be enumerated. An enumeration is a list of possible values that a parameter is allowed to take along with optional text labels. In the PathWave FPGA IP Packager, these show up as drop down selection list. This can be convenient so the user doesn't have to know how various options are encoded. In this example, instead of making the user know that `dir=0` means count up and `dir=1` means count down, we'll use an enumeration.

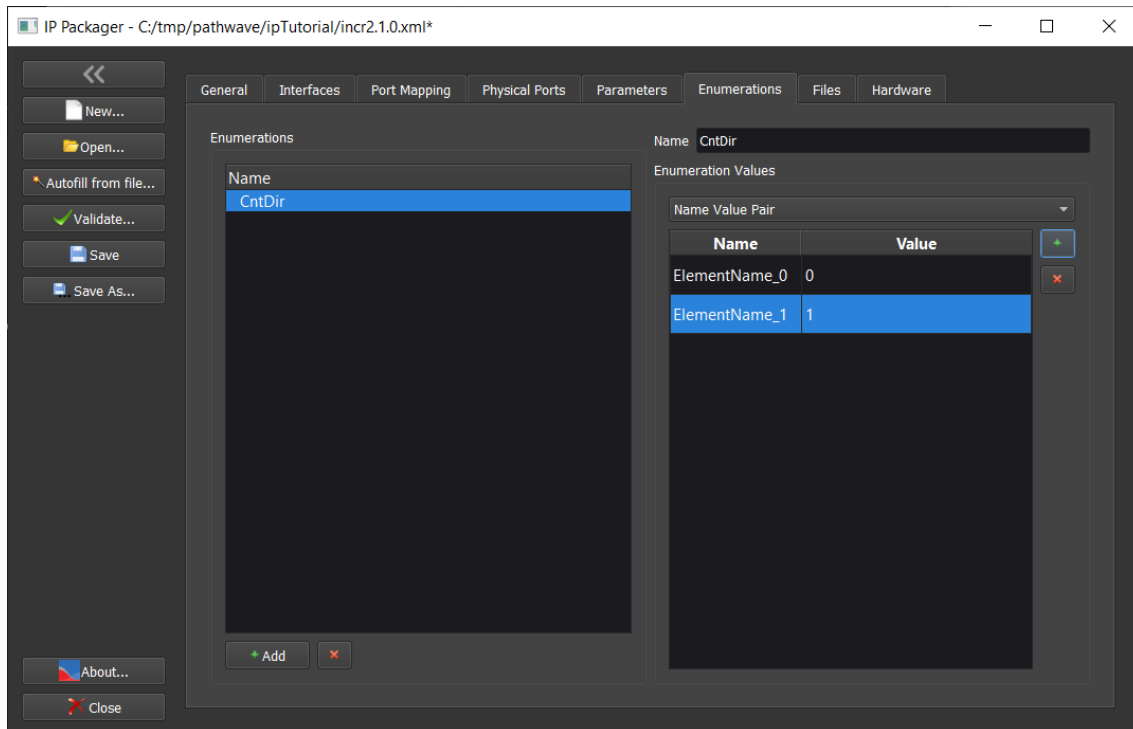
Click on the *Enumerations* tab. This will initially be empty. Click on the *+Add* button to add an enumeration. Change the enumeration name to *CntDir*:



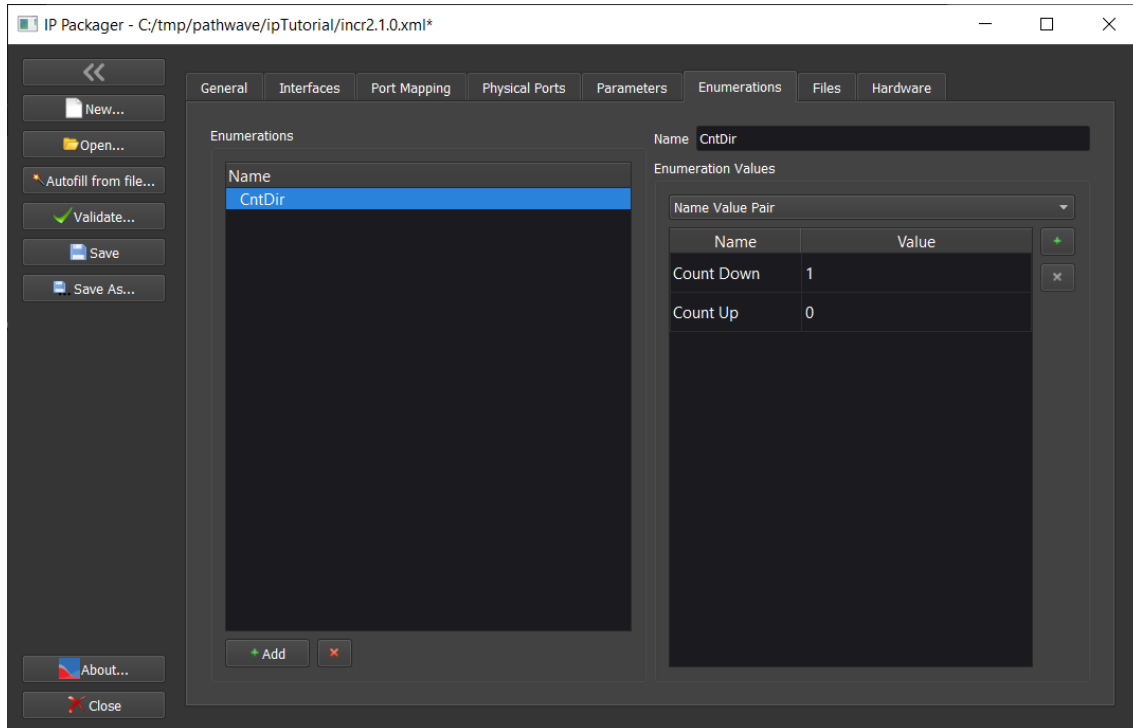
Now we have to specify the possible values. We'll be labeling these values with text labels, so change the *Enumerated Values* to be *Name Value Pair*:



Now we need to create two Name Value Pairs. Click the + button twice to create two pairs:

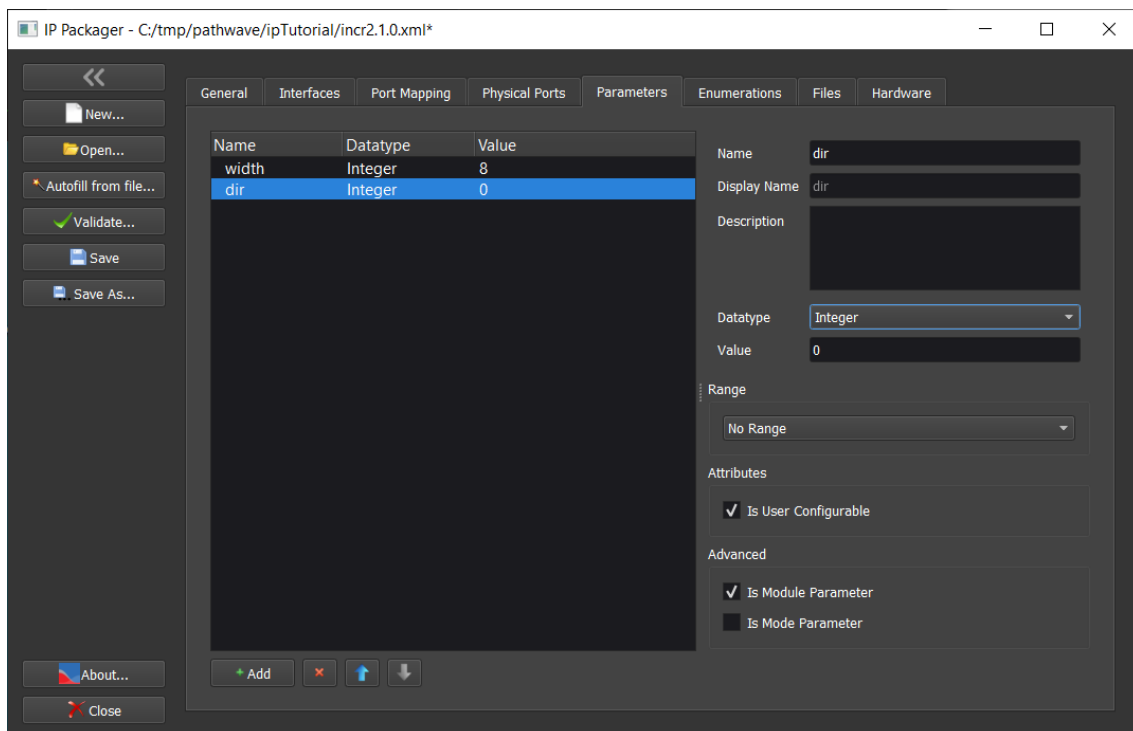


Double click on *ElementName_0* and change it to *Count Up*. Likewise, double click on *ElementName_1* and change it to *Count Down*:



In this case, the value fields 0, and 1, are okay. If different values were desired, you can double click on those and change them to other values. Note that the values are sorted by the *Name* entries. Hence *Count Down* shows up before *Count Up*, even though it has a higher value.

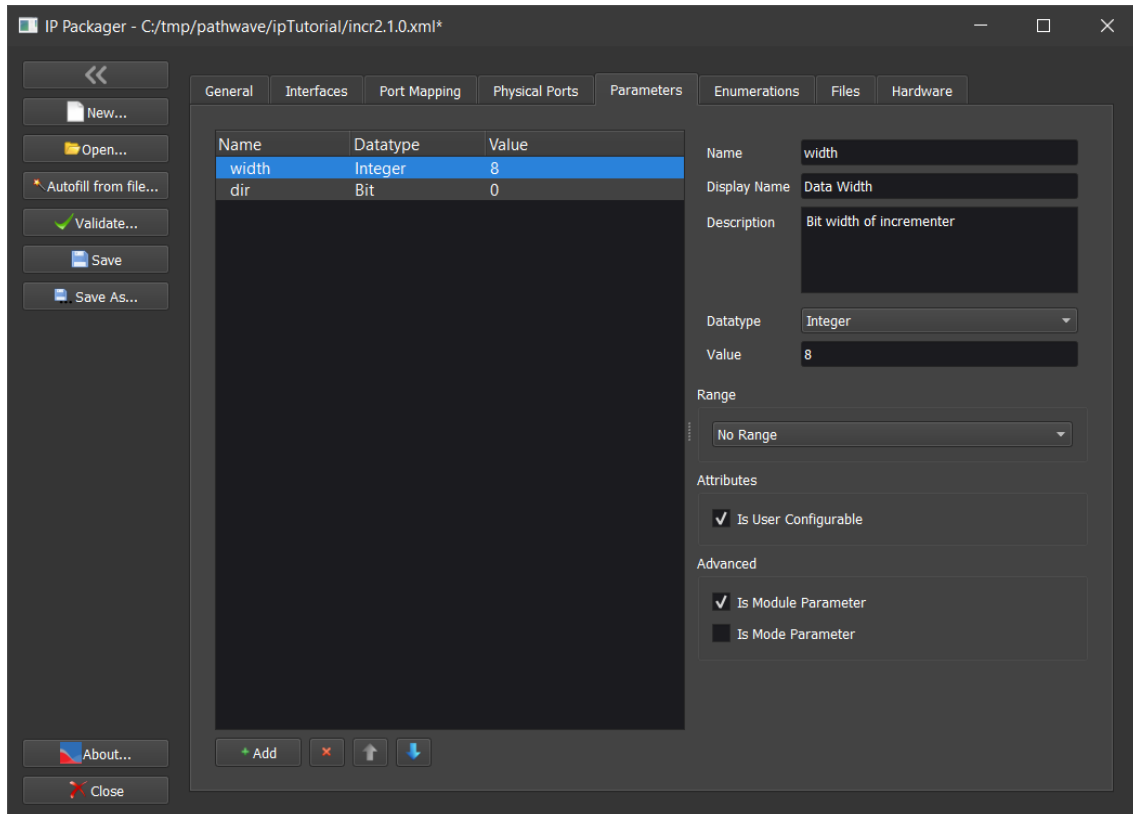
Now we can customize the parameters. Select the *Parameters* tab:



In this example, the IP Packager found the parameters from the source HDL file. If needed, the *+Add* button can be used to create new parameters, but that isn't needed in this example.

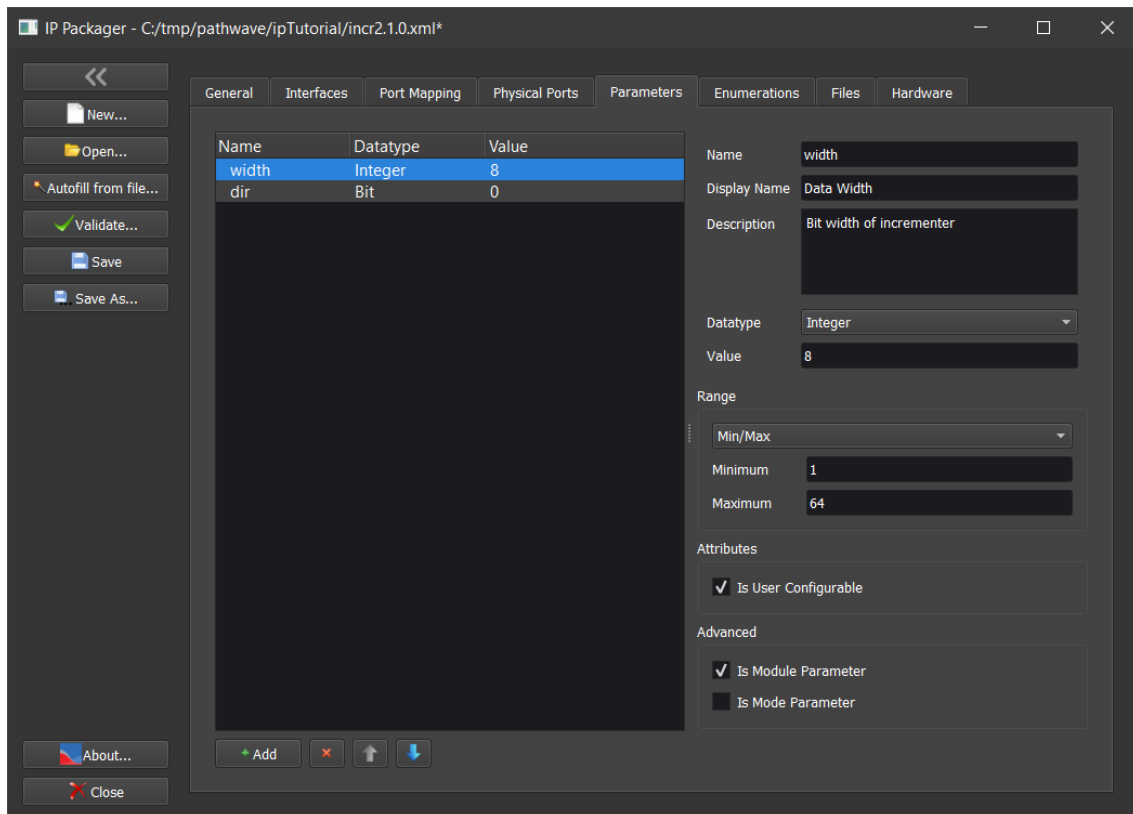
Select the *width* parameter. The parameter's *Name* is the name of the parameter in the source HDL. The *Display Name* is what is displayed by the PathWave FPGA gui when the IP is instantiated. The *Description* field can be used to provide more information to the user. In the PathWave FPGA gui, this text is shown when the user hovers the mouse over that

parameter. Change the *Display Name* to *Data Width* with a description of *Bit width of incrementer*:



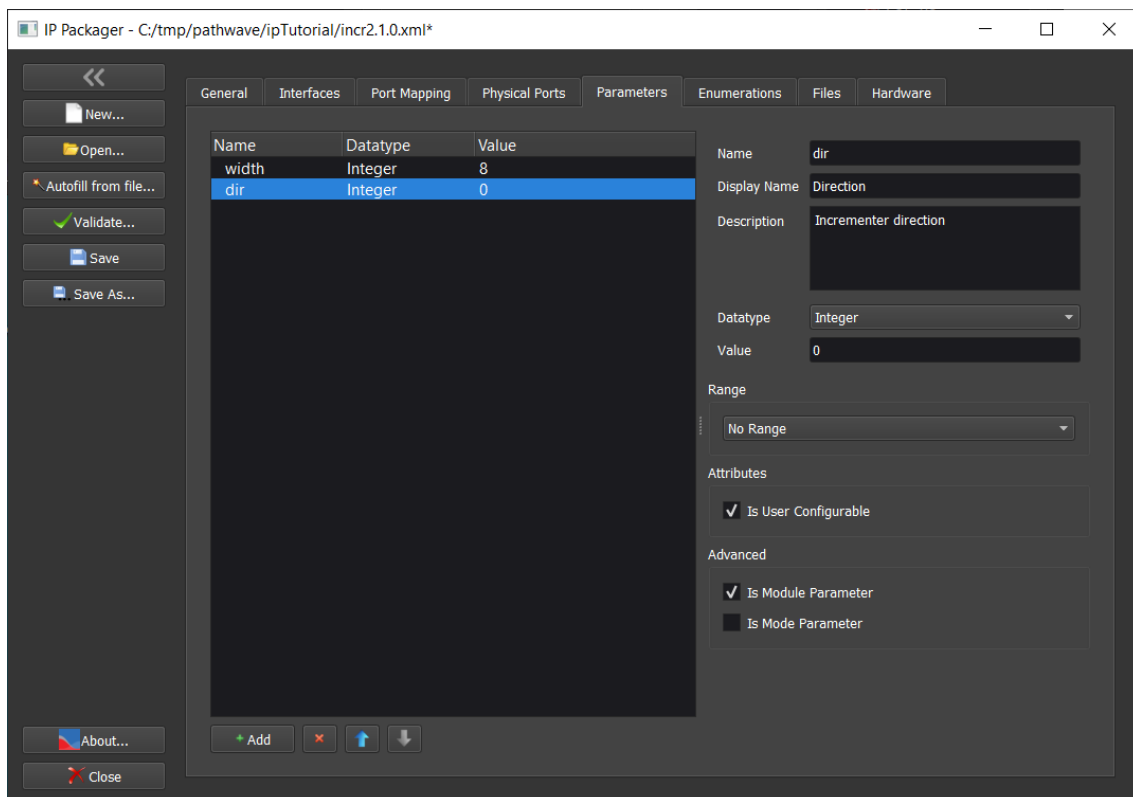
This parameter should be an integer, so the *Datatype* can be left as *Integer* with the default *Value* of 8. The default *Value* is the value that initially shows in the PathWave FPGA gui before the user changes it.

For this IP, we'll say the minimum value for width should be 1 (it doesn't make sense to have 0 or a negative width). We'll also define a reasonable upper limit for the width to prevent users from entering an unreasonably large number which won't build. Change the *Range* selection to *Min/Max*, and set the *Minimum* to 1 and the *Maximum* to 64:

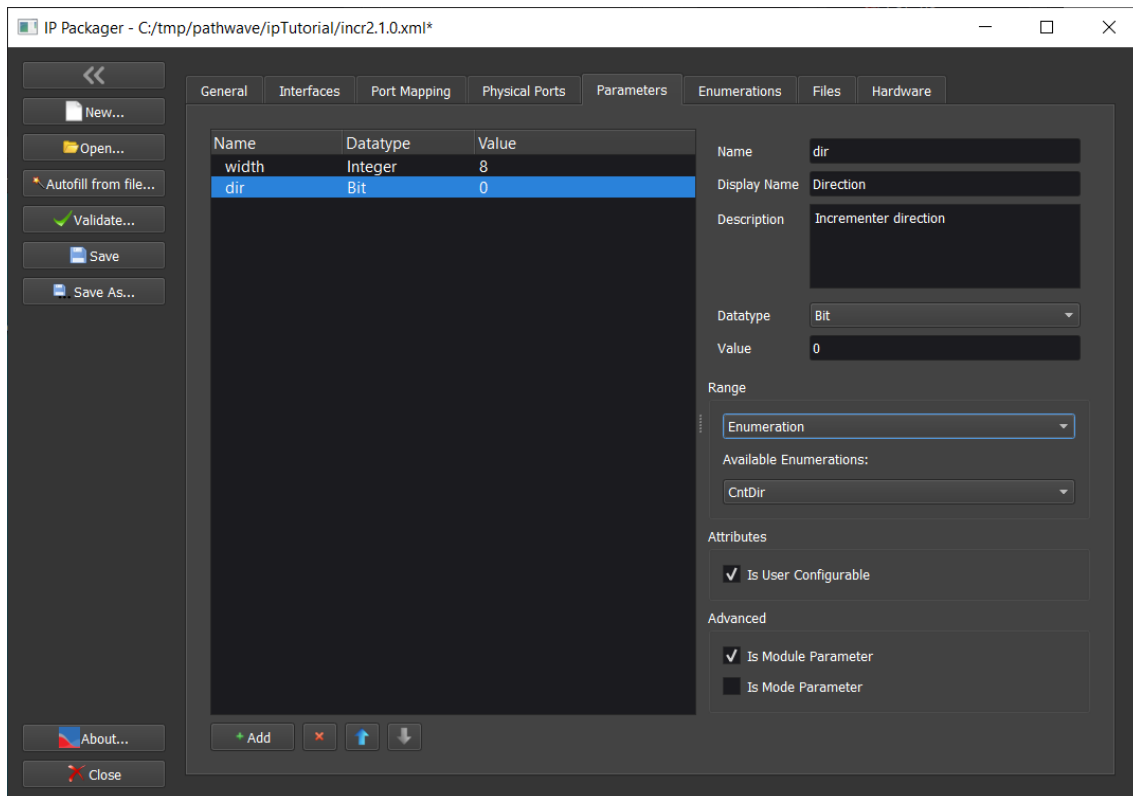


We'll leave the *Attributes* and *Advanced* check boxes at their default values.

Select the *dir* parameter and change the *Display Name* and *Description* as shown:

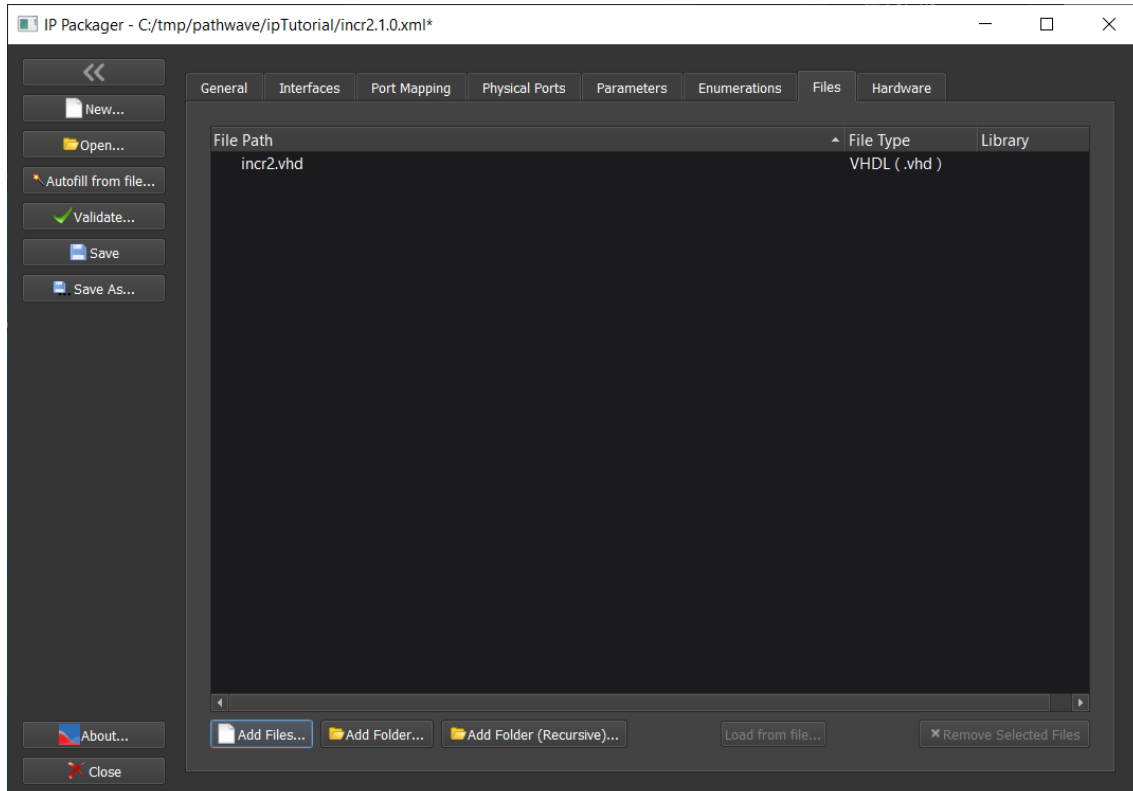


Change the *Datatype* to *Bit* since this is a boolean parameter. Next, change the *Range* pull down to *Enumeration*:



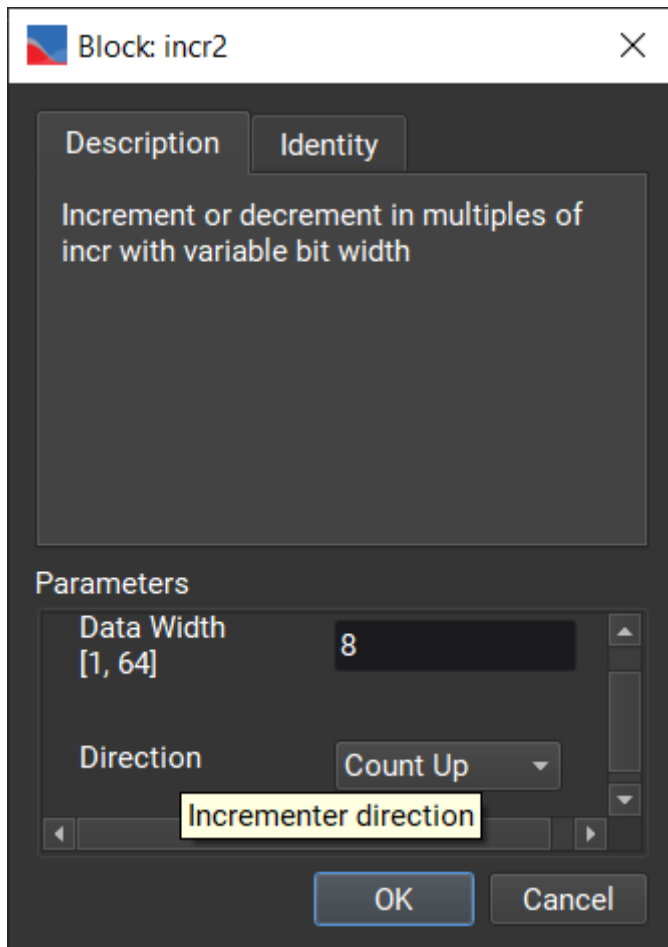
In this case, there is only one enumeration defined, namely *CntDir*. If more than one enumeration had been defined, you'd select which one to use here.

The *Files* tab will only have the one source file *incr2.vhd*:

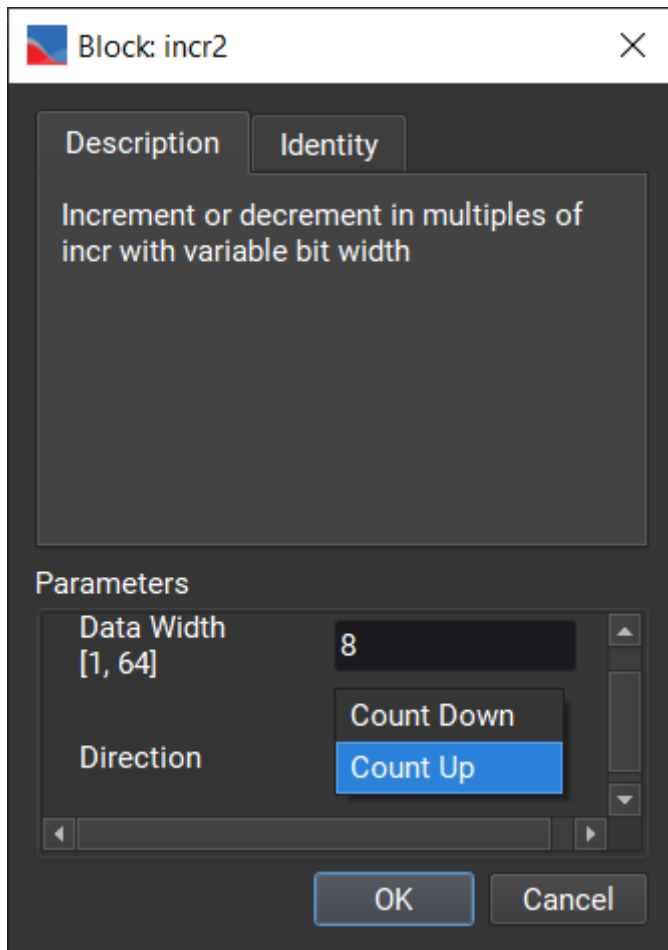


Click **Save** to save the IP-XACT file, then close the module and exit the IP Packager. That's all there was to packaging this example.

When this block is instantiated in PathWave FPGA, the user sees:



It shows the description that was entered in the *General* tab along with any user configurable parameters. In this case we see *Data Width* and *Direction*. If no *Display Name* had been entered, then the name of the parameter would have been used. The limits of [1,64] for the *Data Width* parameter are shown. If a user tries to enter a value outside this range, PathWave FPGA will flag it as an error and not let the user continue until he fixes the problem by entering a valid value. The mouse-over on *Direction* shows the parameter's description field. The *Direction* parameter uses a pull down box to select whether to count up or count down:



Advanced IP Packaging

The previous tutorials showed how to easily package IP possibly using simple parameters. This tutorial will show how to handle more complex situations. These include using non-user configurable parameters, parameter expressions, multiple clock domains, and design hierarchy.

For this tutorial, we'll use a fictitious AWG (arbitrary waveform generator) design `myAwg.v`. Unlike the earlier tutorials, this HDL uses Verilog rather than VHDL. Since the packaging operation only needs information about the IP block's ports, this HDL file does not have any body, it is just the port declarations. Thus it can't be built. This fictitious block has an MEM bus interface on one clock domain for loading the AWG data, and an AXI4-streaming interface on a different clock domain for streaming the data out. It calls two other IP blocks, `myClockCrosser.v` and `myRam.v`, which are located in a subdirectory called `ip`. For the purposes of packing the IP, the contents of these files aren't needed so these files are essentially blank.

The `myAwg` block is designed to output supersampled data. That means each output word can hold multiple samples in parallel. The size of each sample is set by the *dsample* parameter, and the number of parallel output samples/clock is set by the *supersample* parameter. Each output sample can have one or more *tuser* bits. The number of these bits is set by the *usize* parameter. Finally, the size of the memory used in the AWG is set by the *depth* parameter which denotes the number of 32-bit words used in the RAM. This is also the number of bits needed in the MEM bus's address bus.

In addition to the user configurable parameters discussed above, there are a couple parameters that are not user configurable. These parameters are not visible in the IP Packager and can't be directly changed by the user. Instead, they are calculated from the user customizable parameters. Sometimes this can be convenient if a calculated value is used in multiple port declarations so that the expression is only entered once and the results of the expression are

used many times. Sometimes separating the expression out can also make it clearer what the code is doing. This is not needed, however, as one can also use the more complicated expressions directly without defining a separate parameter for that expression. This example does it both ways.

For parameters to be correctly autofilled from Verilog, they must be defined in the IP block's module definition line (prior to the port list). Parameters defined in the body of the Verilog code will not be autofilled though they can always be added manually.

The parameters used in the IP Packager are IP-XACT parameters. These parameters follow the syntax used by the IP-XACT specification which is the same syntax used by System Verilog. These are related but distinct from the parameters/generics used in the HDL code. When a design is built in PathWave FPGA, Pathwave FPGA will evaluate any parameter expressions using the IP-XACT syntax and use evaluated constants when generating the output HDL file for synthesis.

In this tutorial, we will autofill from the source HDL file as that is much easier than manually entering all that data by hand. This tutorial uses the following top level Verilog file:


```

/* -*-Verilog-*-
*****
*
* File:          myAwg.v
* Description:   IP Packager tutorial example
*
*****
*/

module myAwg #(parameter
    // First define user configurable parameters
    dsize = 16,           // Size of each sample
    supersample = 1,      // Supersample factor
    depth = 1024,         // Depth of memory in 32 bit
words
    usize = 1,           // Number of tuser bits per
sample
    // The following parameters are calculated
    asize = $clog2(depth), // Number of address bits needed
    osize = supersample*dsize // Size of output stream
)
(
    // First declare mem interface for loading the AWG
    input          m_clk,           // Mem bus clock
    input          m_nrst,          // Mem bus reset
    input [asize-1:0] m_address,     // Mem address bus
    input [31:0]    m_wrdata,        // Mem write data bus
    output [31:0]   m_rddata,        // Mem read data bus
    input          m_rden,          // Mem read enable
    input          m_wren,          // Mem write enable

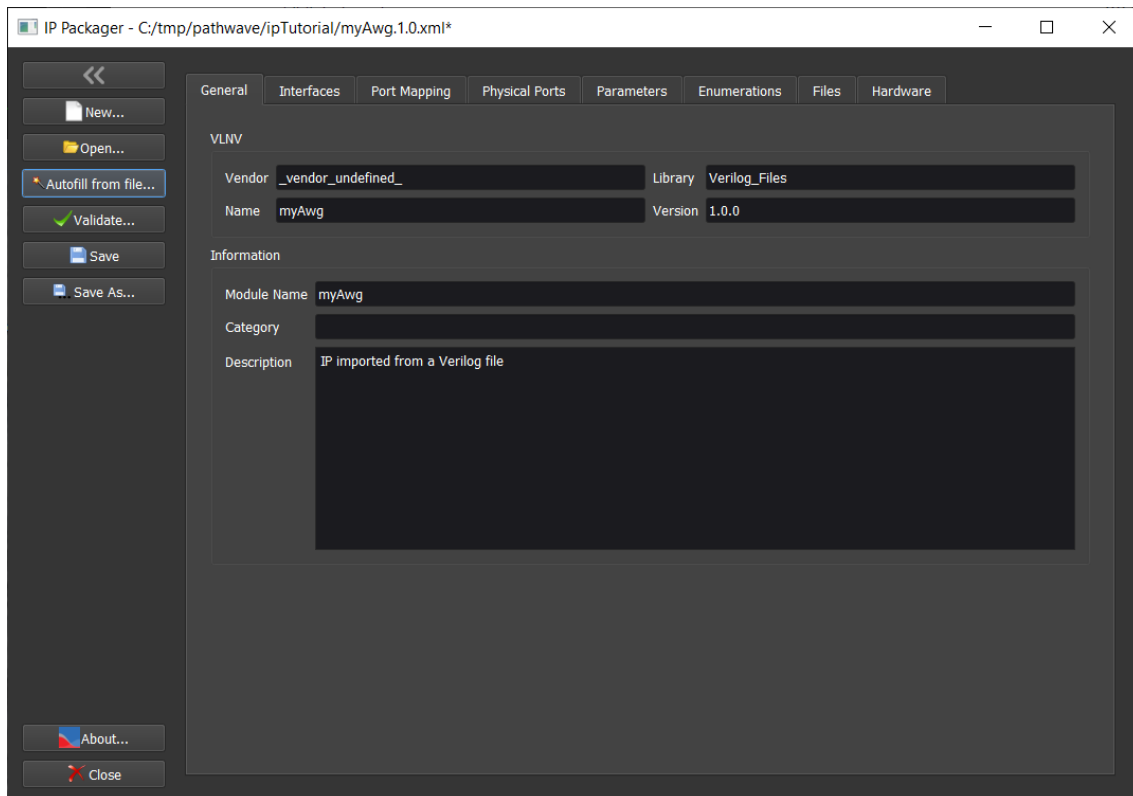
    // Next declare the axi-streaming interface
    input          a_clk,           // Axi bus clock
    input          a_rstn,          // Axi bus reset
    output [osize-1:0] awg_tdata,    // Awg output data
    output         awg_tvalid,      // Awg output valid
    input          awg_tready,      // Awg stream ready
    output [usize*supersample-1:0] awg_tuser
    // Awg tuser bits
);

    // For this tutorial, only the port and parameter definitions are
    needed
    // Hence the body is empty

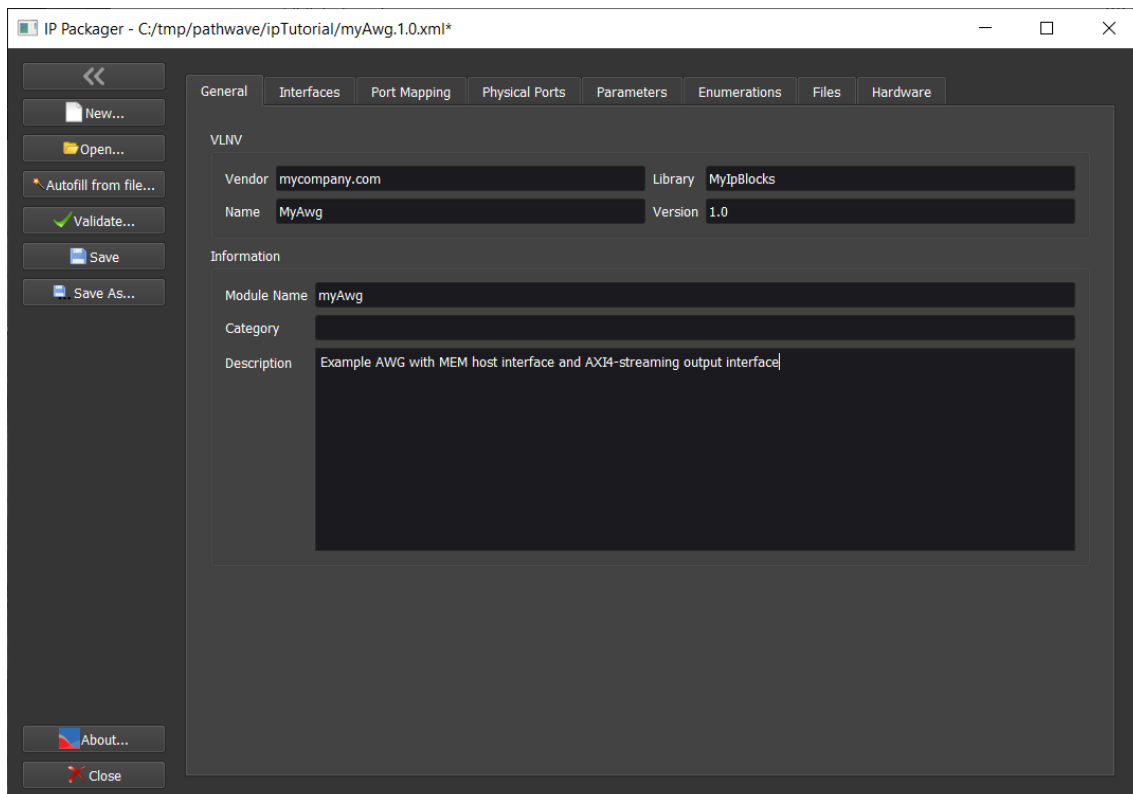
endmodule // myAwg

```

As in the previous tutorials, start the IP Packager, create a new IP-XACT file called *myAwg.1.0*, and autofill it from the *myAwg.v* file:

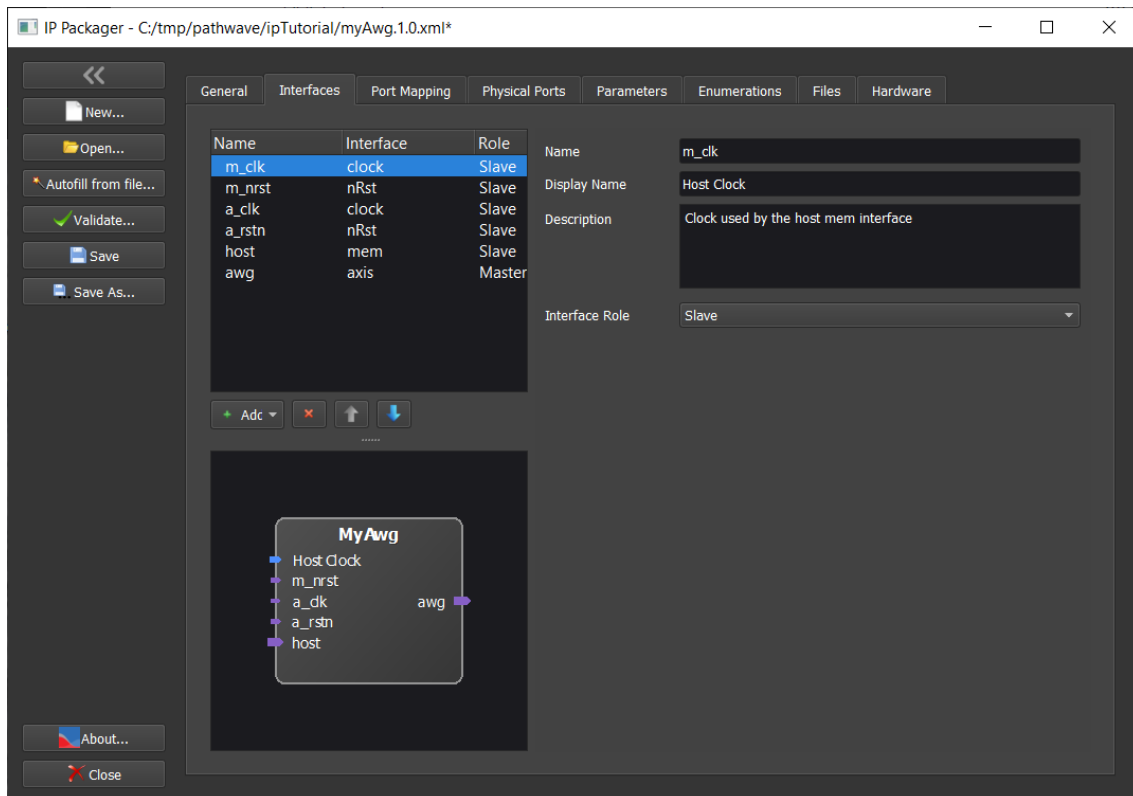


Fill out the VLNv and description as shown below:

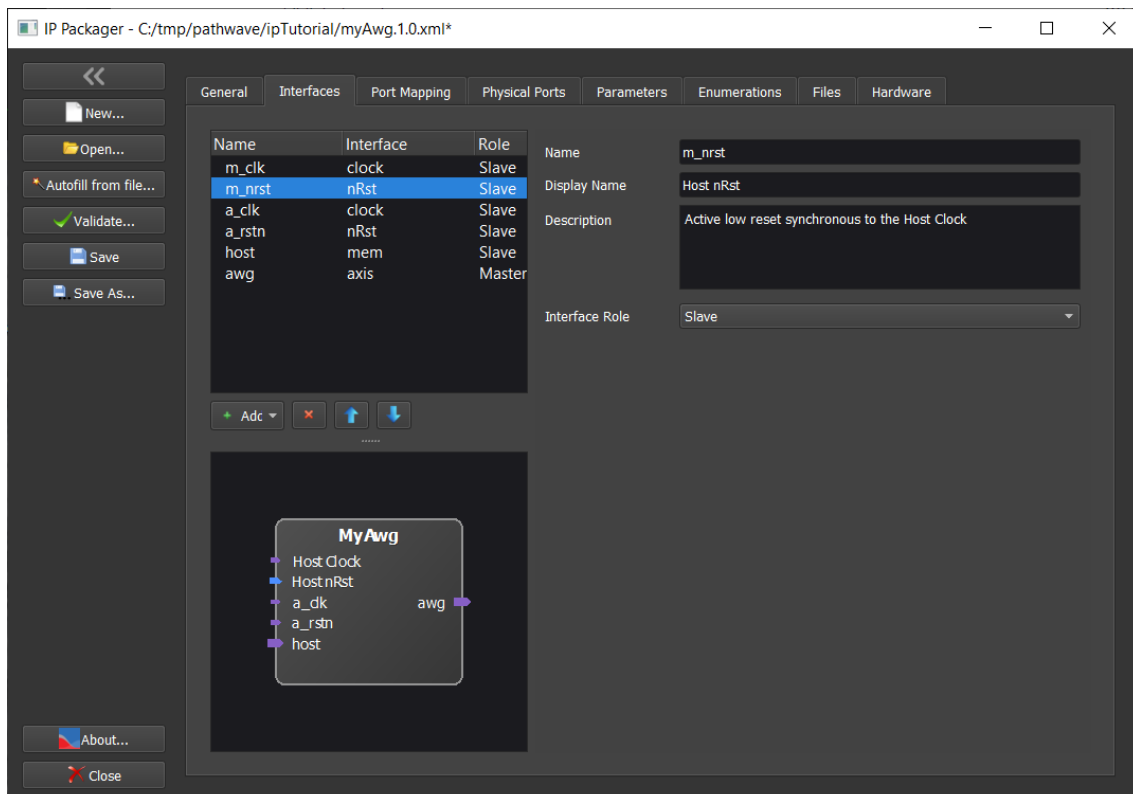


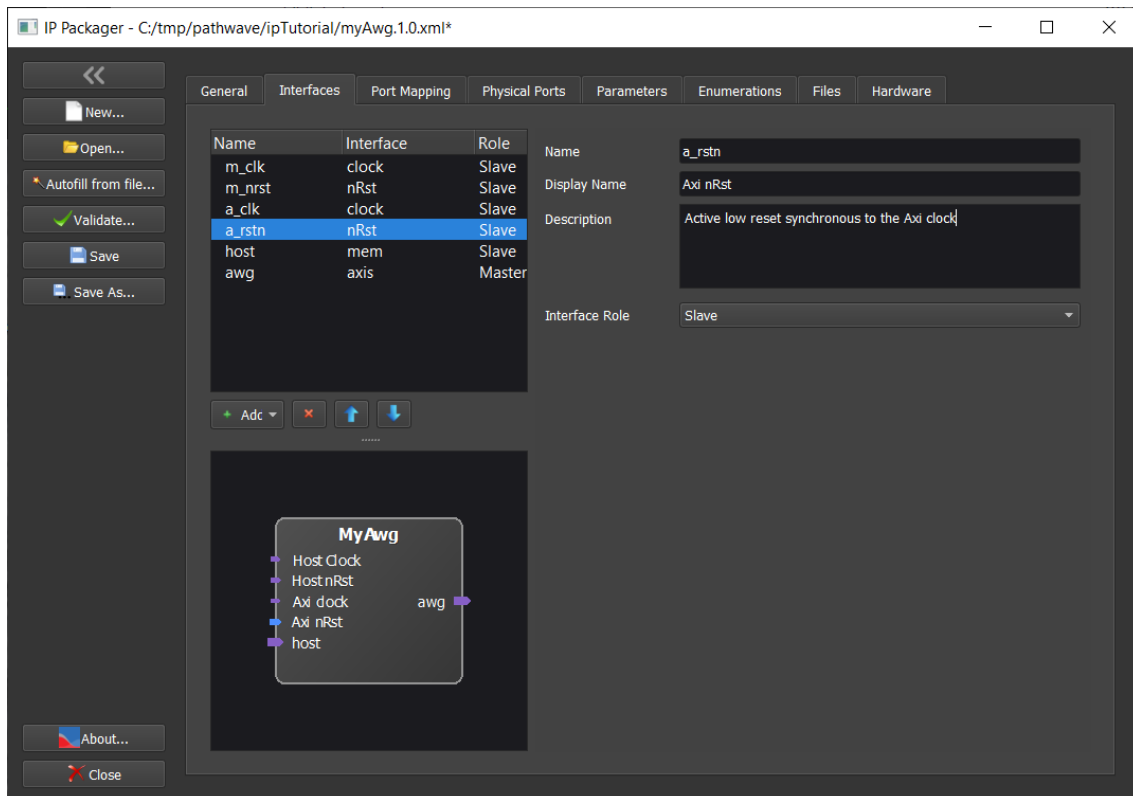
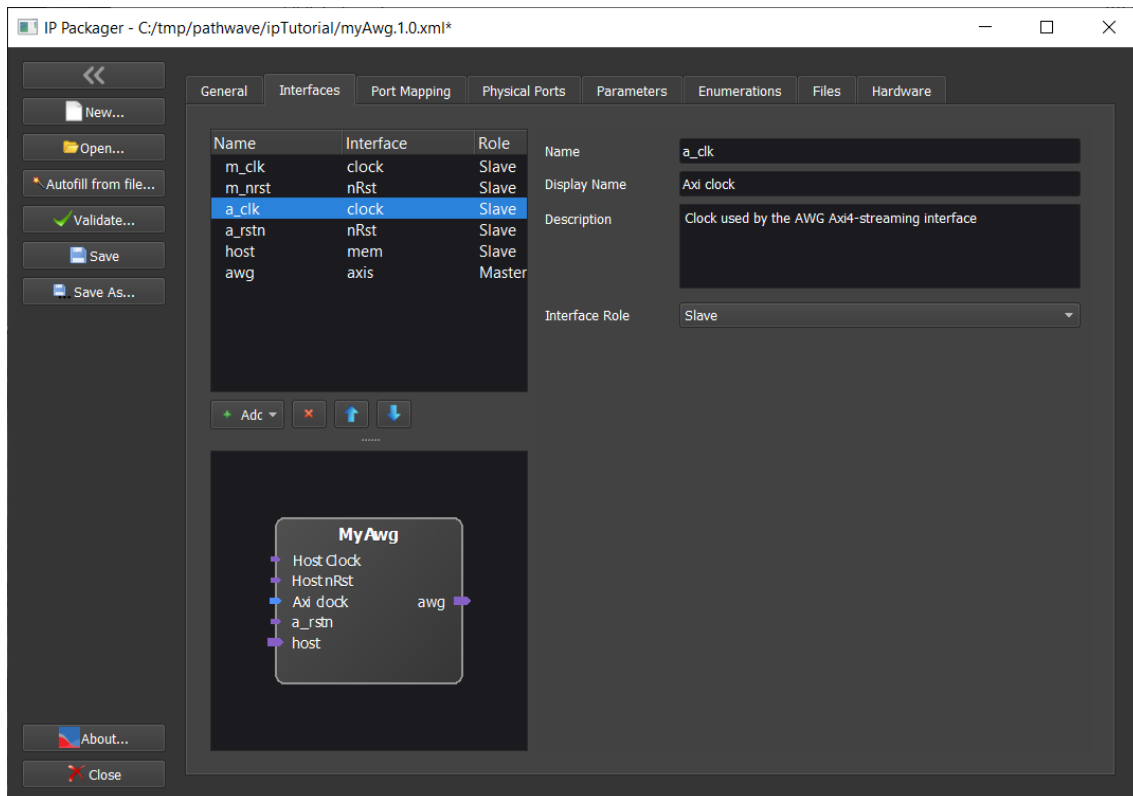
Note that the *Name* does not have to match the *Module Name*.

Click on the *Interfaces* tab and select the m_clk interface. Change the *Display Name* to *Host Clock* and the *Description* to *Clock used by the host mem interface*:

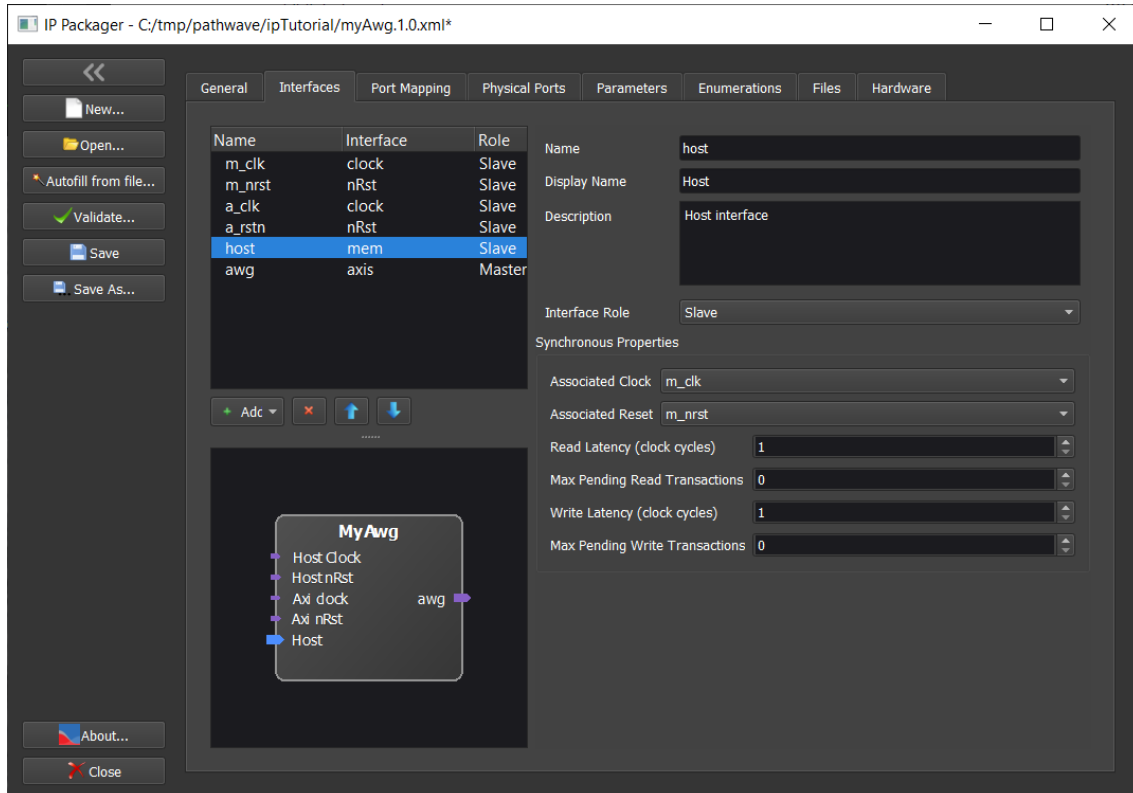


Likewise, update the *Display Names* and *Descriptions* for the *m_nrst*, *a_clk*, and *a_rstn* ports:

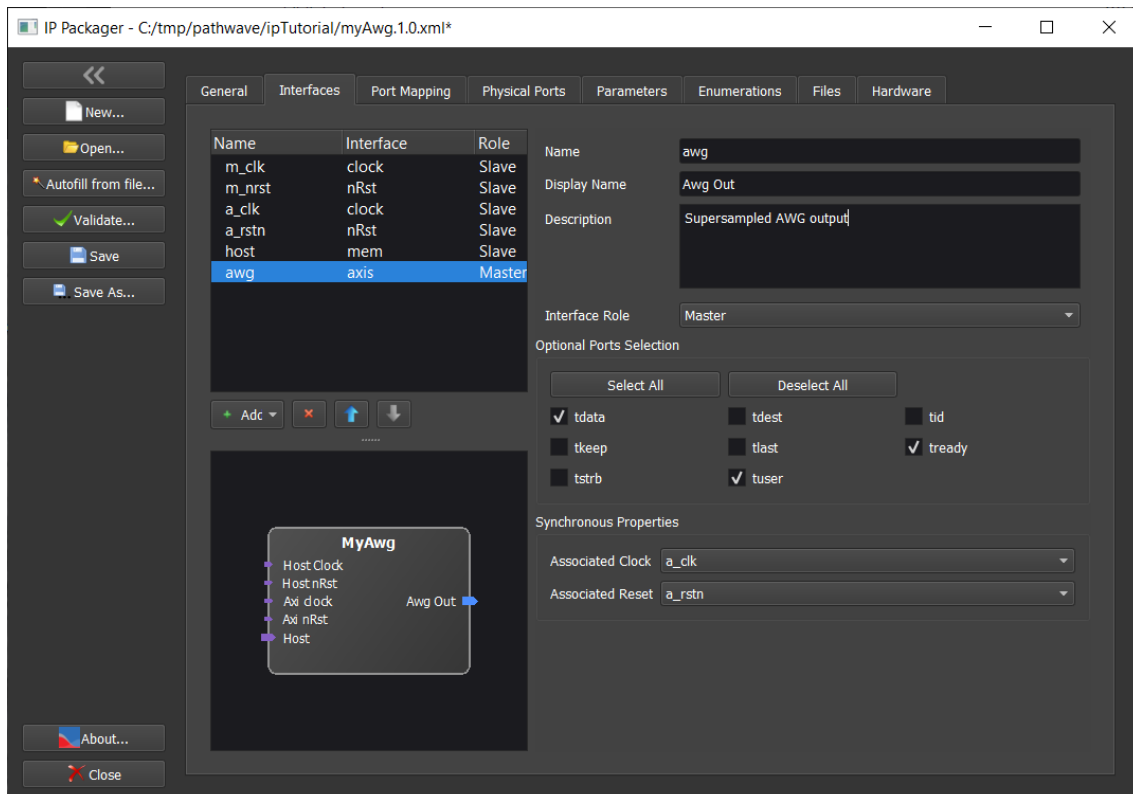




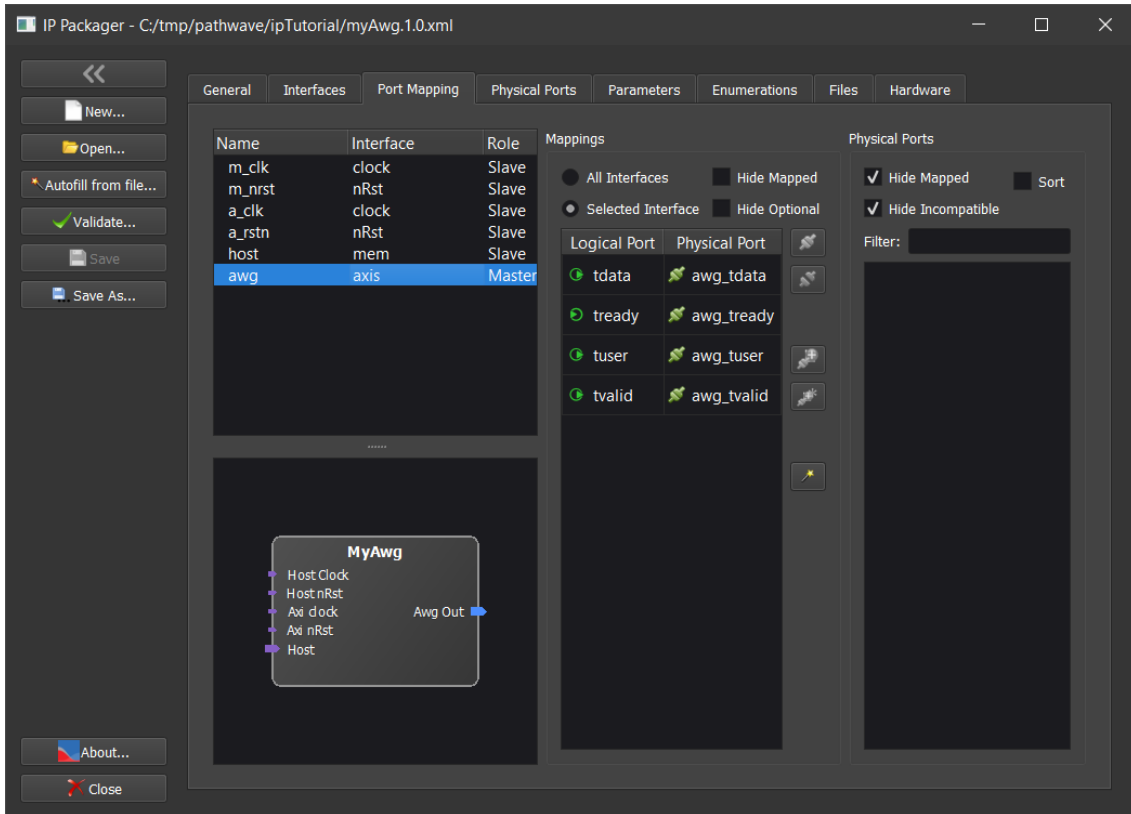
Select the *host* interface. This has additional fields, in particular the *Associated Clock* and *Associated Reset* fields. These should indicate the clock and reset used by this interface. In this case they are already correct so we can just leave them as is. Update the *Display Name* and *Description*:



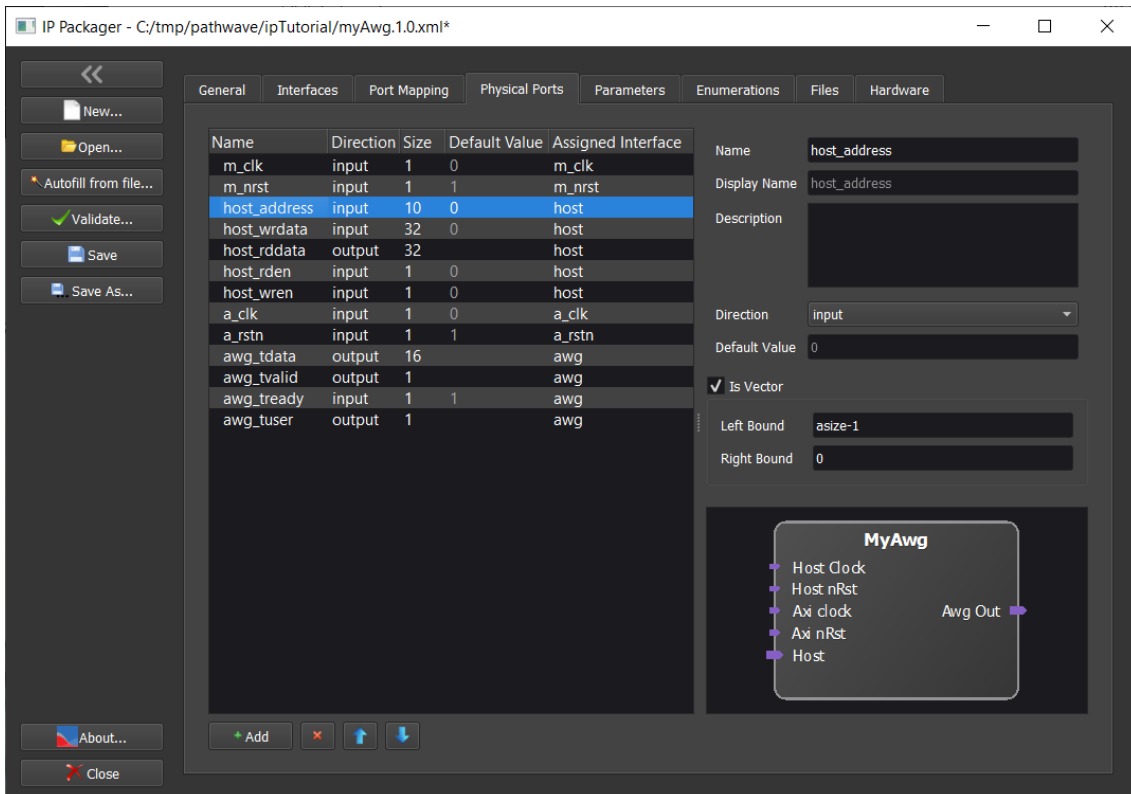
Select the *awg* interface. In addition to changing the *Display Name* and *Description*, we'll need to change the *Associated Clock* and *Associated Reset* to *a_clk* and *a_rstn* as this interface uses that clock:



The *Port Mapping* tab shows the port mappings. Nothing needs to be changed here:

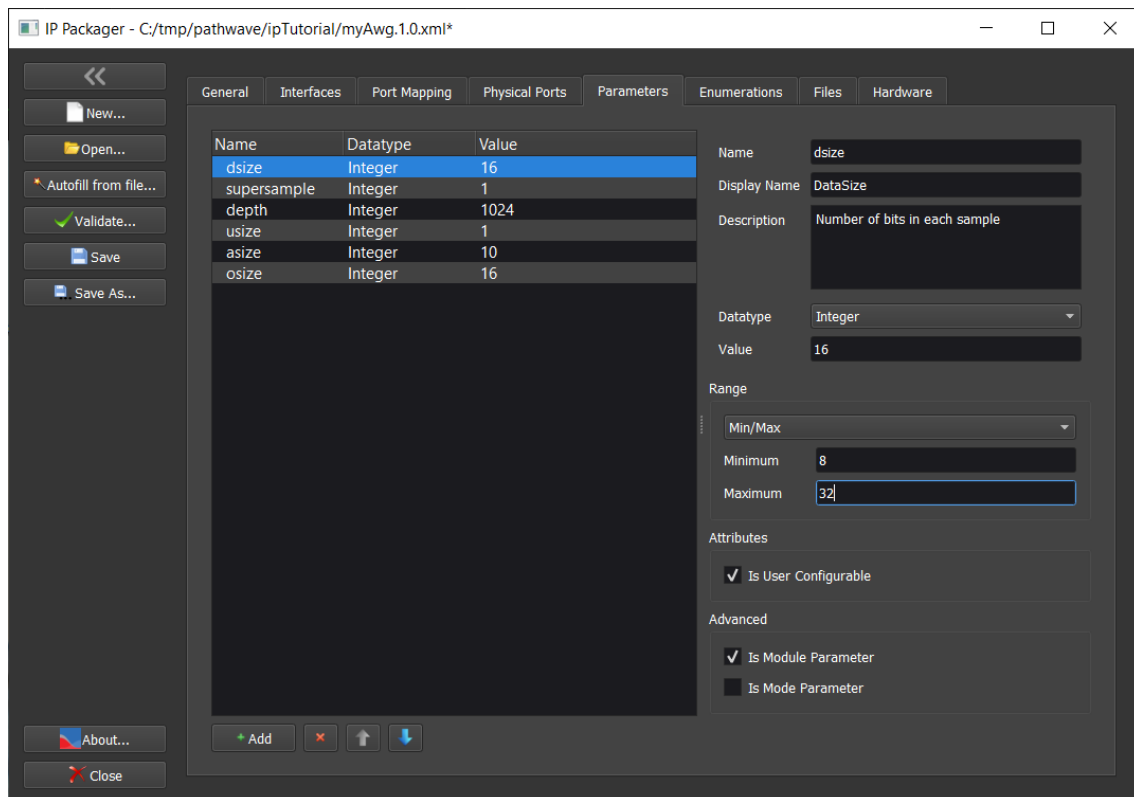


Click on the *Physical Ports* tab. You can see all the ports and which interfaces they are assigned to. While nothing needs to be changed here, there are a few things to note. Click on the *host_address* port and see that the *Left Bound* of the bus is the expression *asize-1*, just as in the source HDL code:

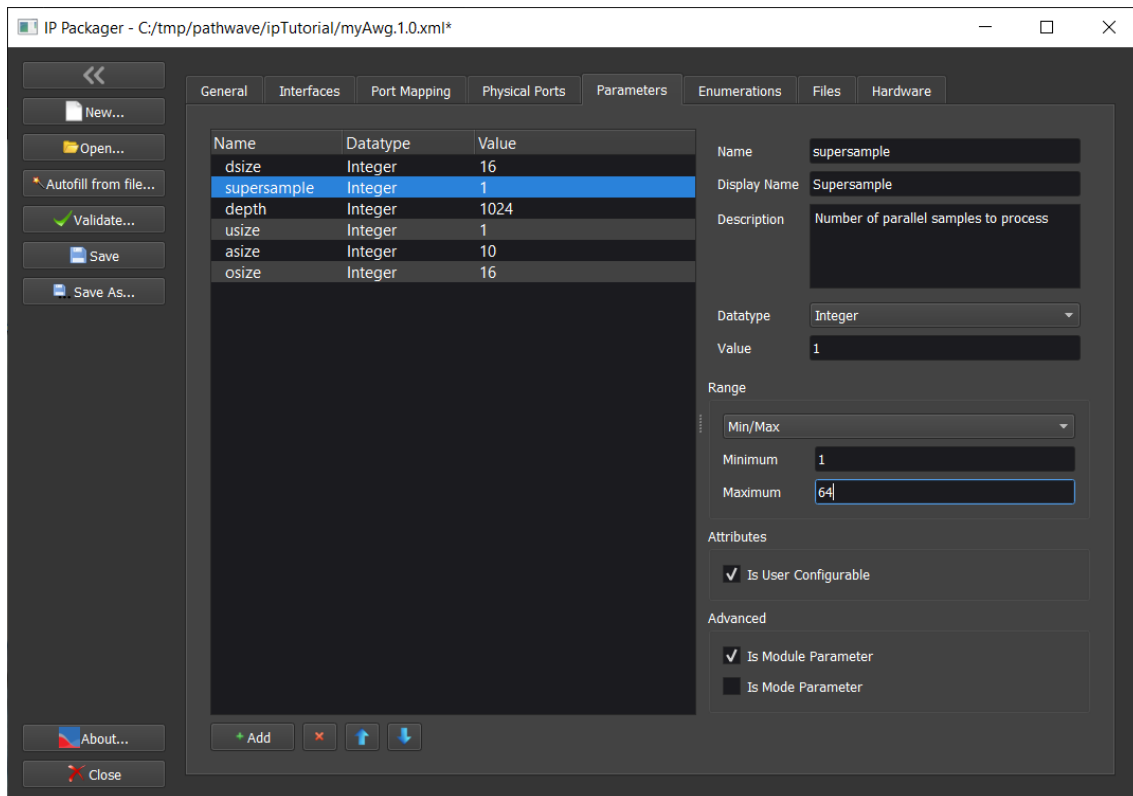


Click on the *awg_tuser* port and see that the *Left Bound* is the more complicated expression *usize*supersample-1*. The bounds for vector size can use complicated expressions if desired.

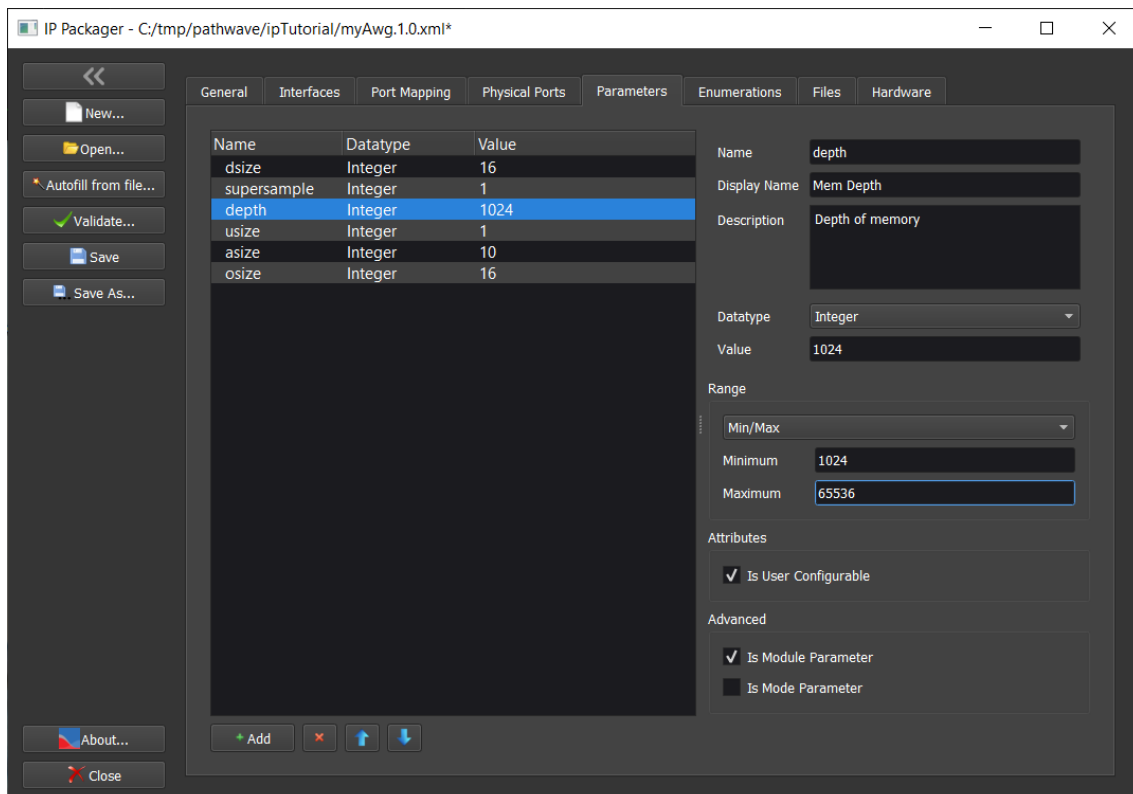
Now let's work on the parameters. Click on the *Parameters* tab and select *dsize*. Change the *Display Name*, *Descriptoin*, and *Range* as shown below:



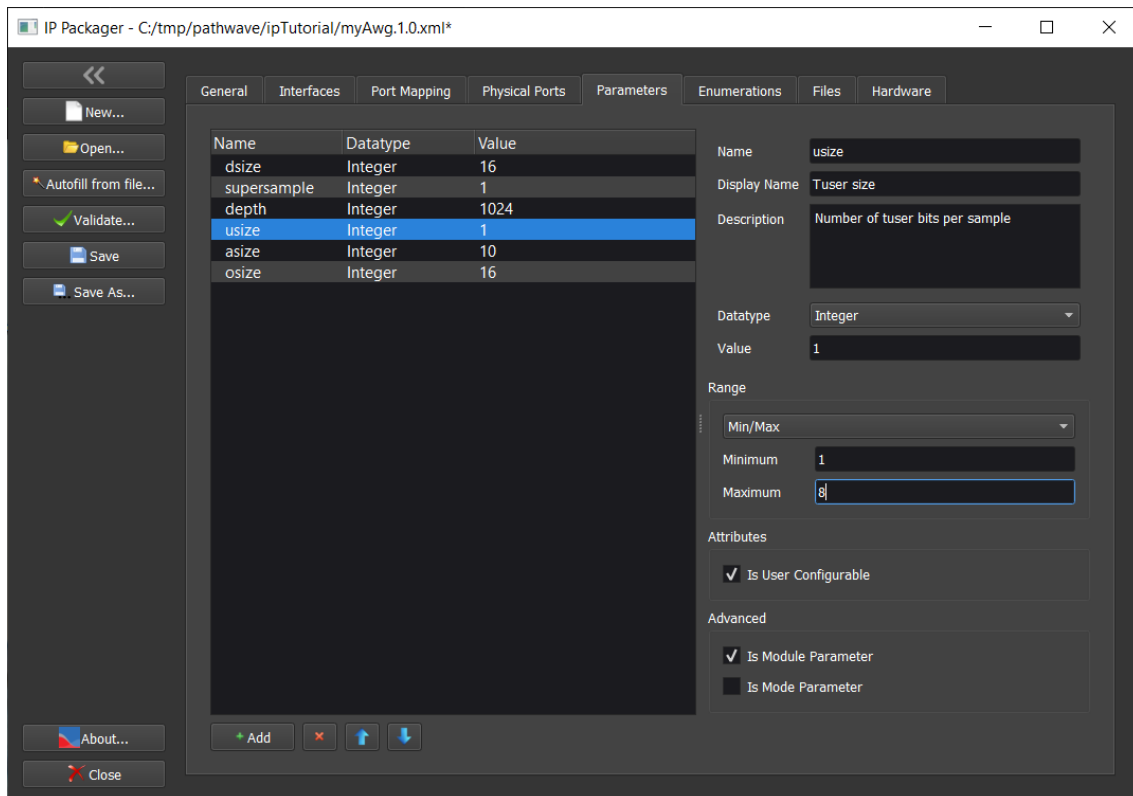
Click on *supersample* and change its entries as shown below. Note that some of these values, such as *Maximum* are subjective.



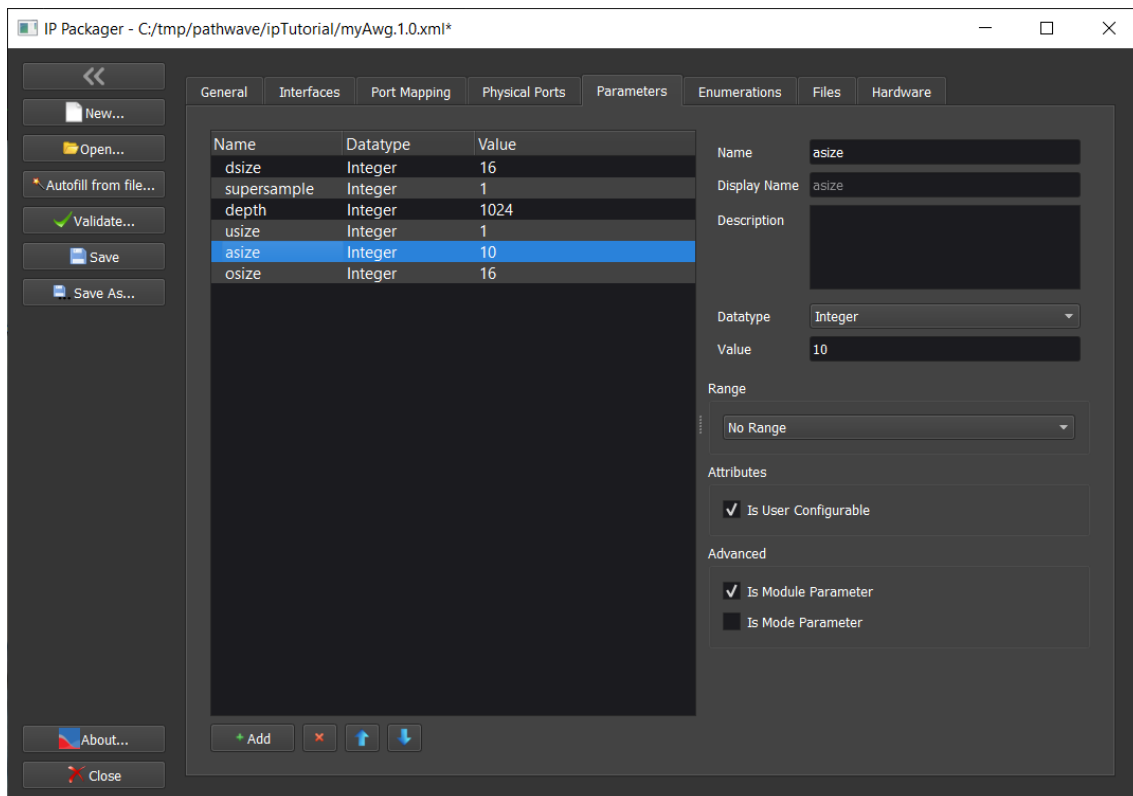
Likewise, click on *depth* and change its entries as shown below:



Finally, click on *usize* and change its entries as shown below:

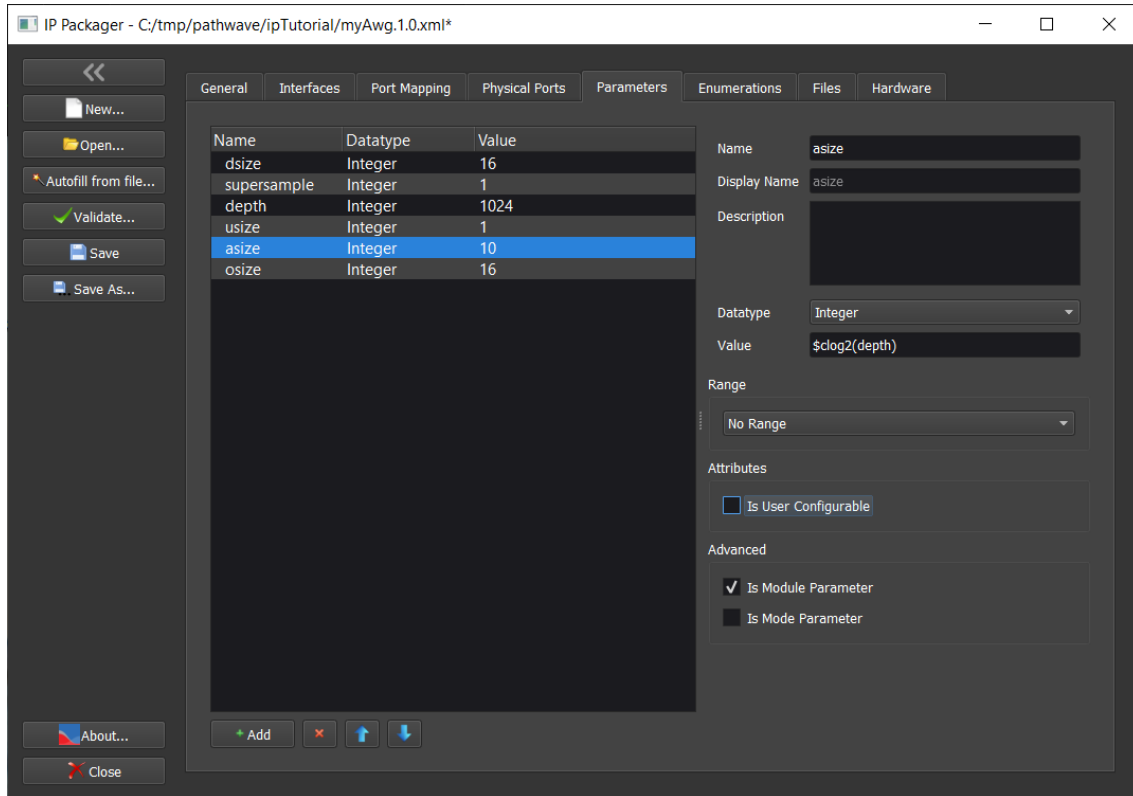


We've finished with the user configurable parameters. We now move onto the non-user configurable parameters. Click on *asize*:



Since *asize* is not configured by the user, there is no point in changing the *Display Name* or *Description* or setting a range. Uncheck the *Is User Configurable* box to indicate this is not a user configurable parameter.

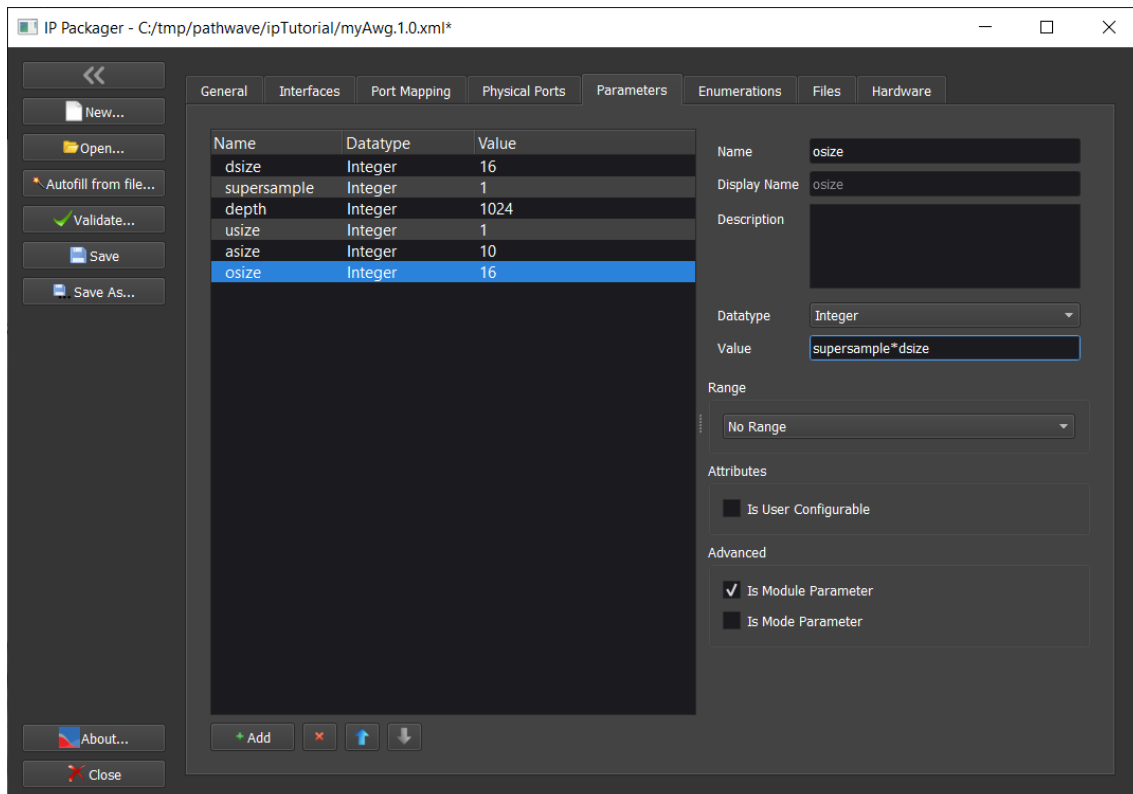
Note that for a *Value* the number 10 is shown. When importing parameters involving expressions, the expression is evaluated using the default value of the other parameters, and that number is used for *Value*. We want the *Value* to be the expression, not just the number, so that it changes when the other parameters change. In the *Value* field enter the expression for this parameter, remembering to use IP-XACT (System Verilog) syntax which in this case is $\$clog2(\text{depth})$:



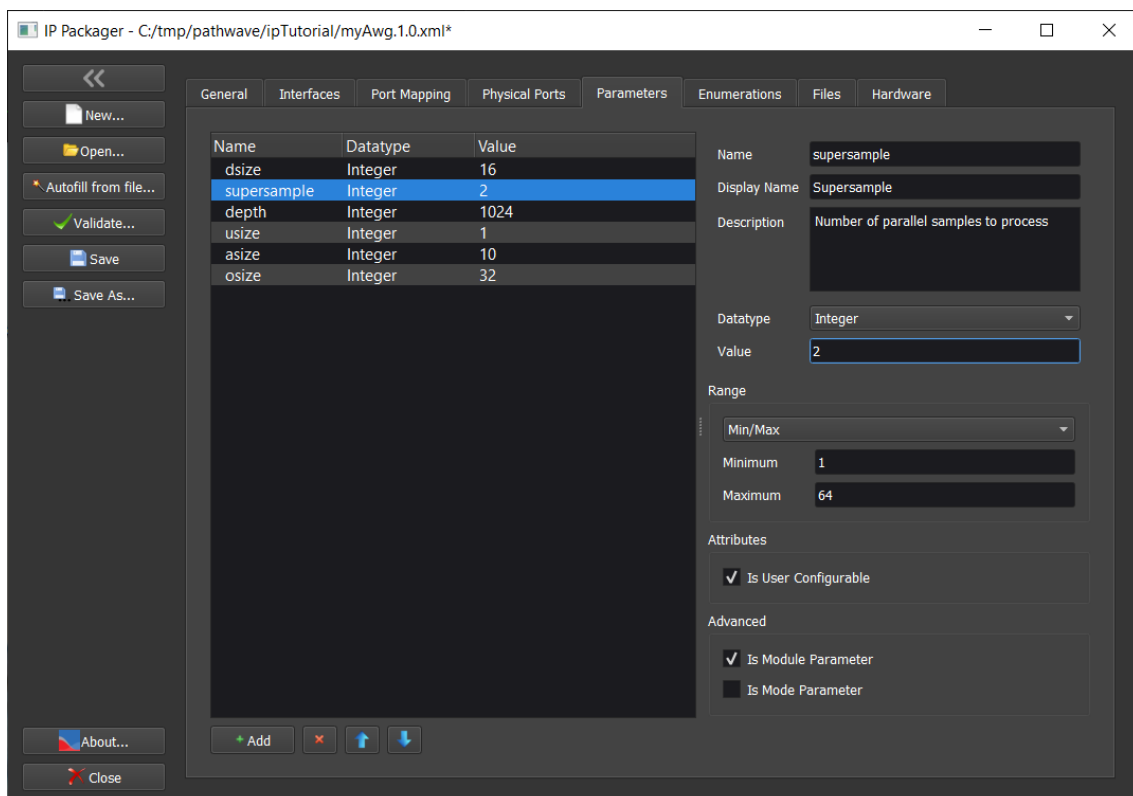
Notice that while entering the expression in the *Value* field, the text turns red when the expression isn't valid, as it isn't until the whole expression is finished.

The $\$clog2(x)$ function calculates $\text{ceil}(\log_2(x))$ which is the number of binary address bits needed to address this much memory.

Next click on *osize*. Uncheck the *Is User Configurable* box, and change the *Value* field to the correct expression $\text{supersample} * \text{dsize}$:

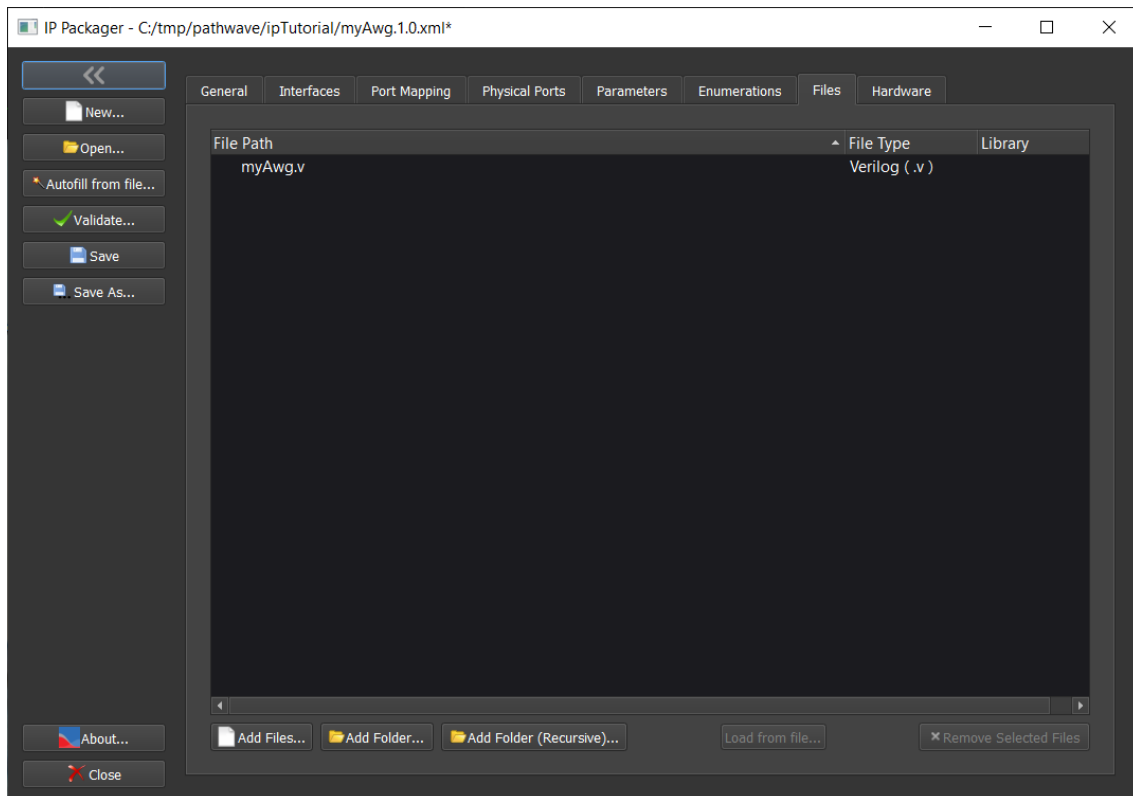


If we now click on *Supersample* and change its value, say to 2, note that the value shown for *osize* also changes, in this case to 32:



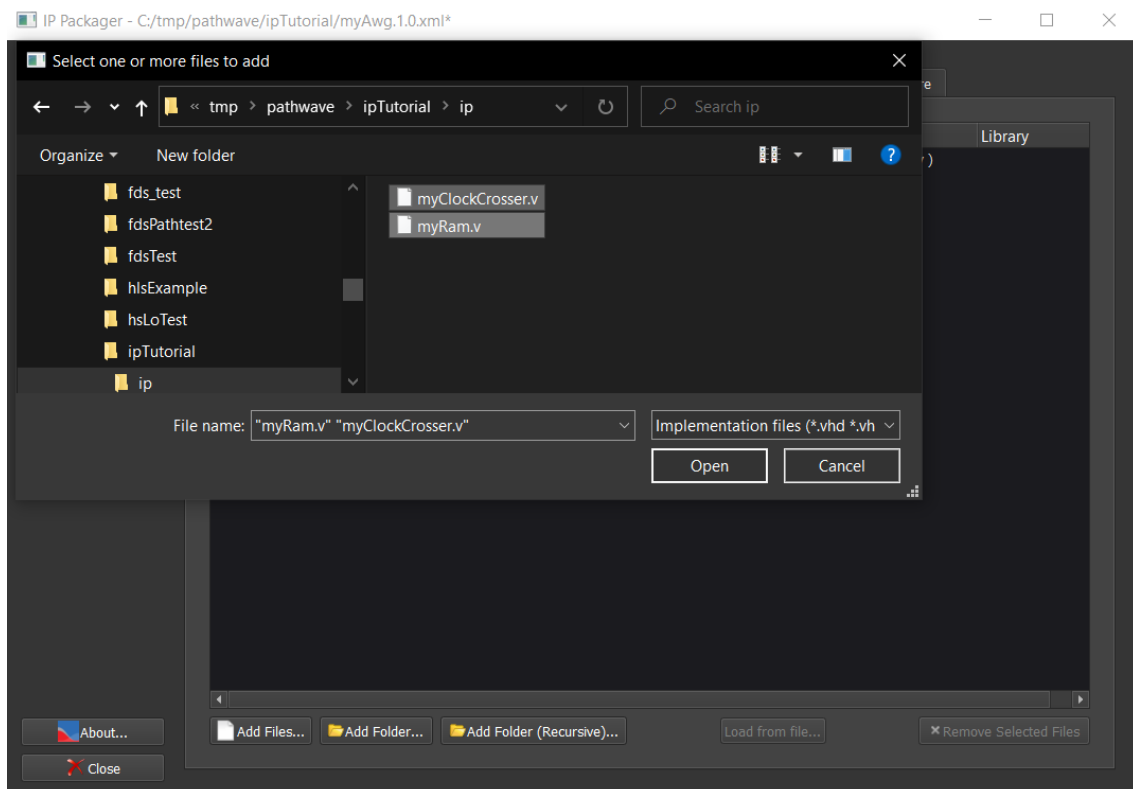
If you've changed the *supersample* value, change it back to 1.

The remaining task for this IP block is to update the fileset with the extra source files needed to build the block. Click on the *Files* tab:

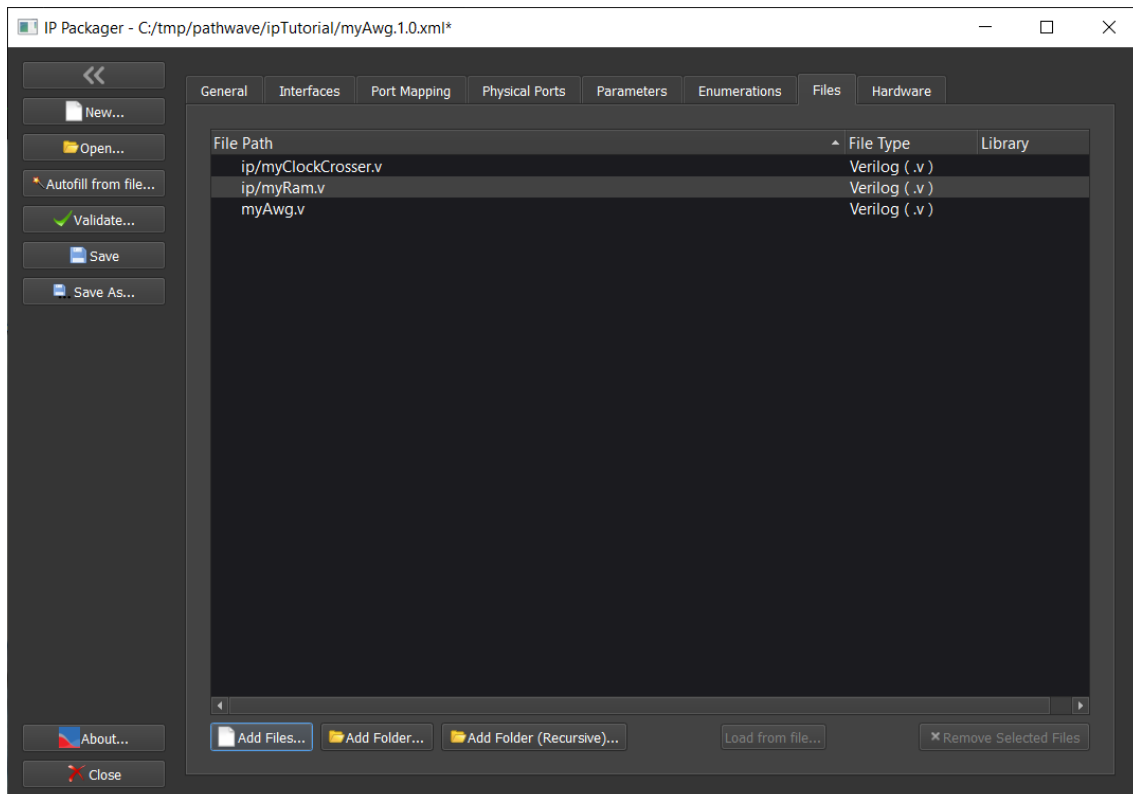


The IP Packager has already entered the source file used to *Autofill from file...*. We need to add the two other source files, the ram block and the clock crossing block.

Click *Add Files...*. Since these other files are in the IP subdirectory, click on the *ip* folder:

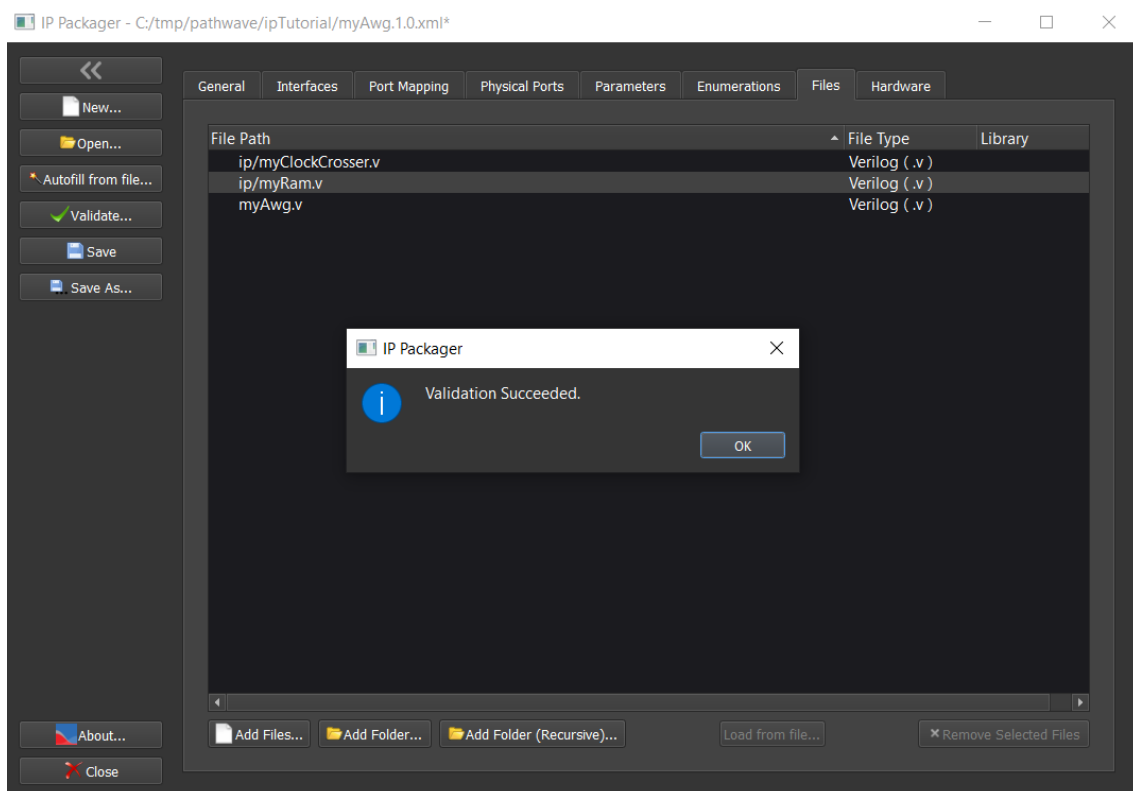


Note that you can add the two files separately by clicking *Add Files...* more than once, or you can select multiple files at the same time. After selecting the two files, click *Open* to add them to the fileset:



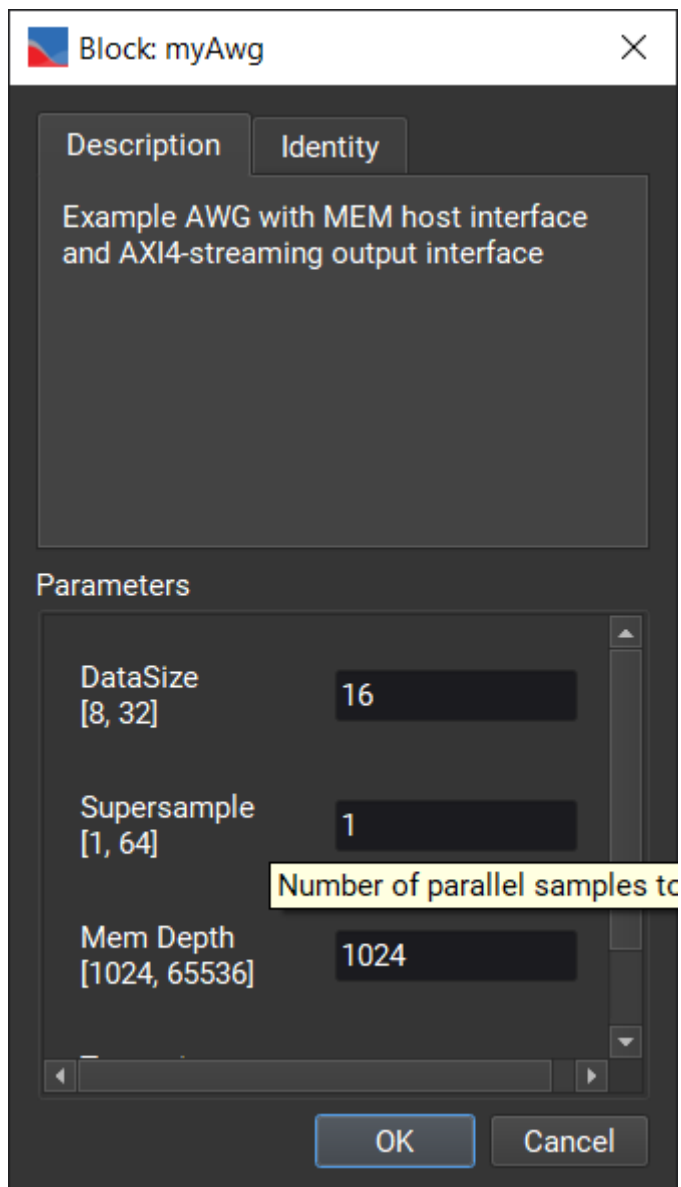
If you make a mistake and add the wrong file, you can use *Remove Selected Files* to remove the wrong one.

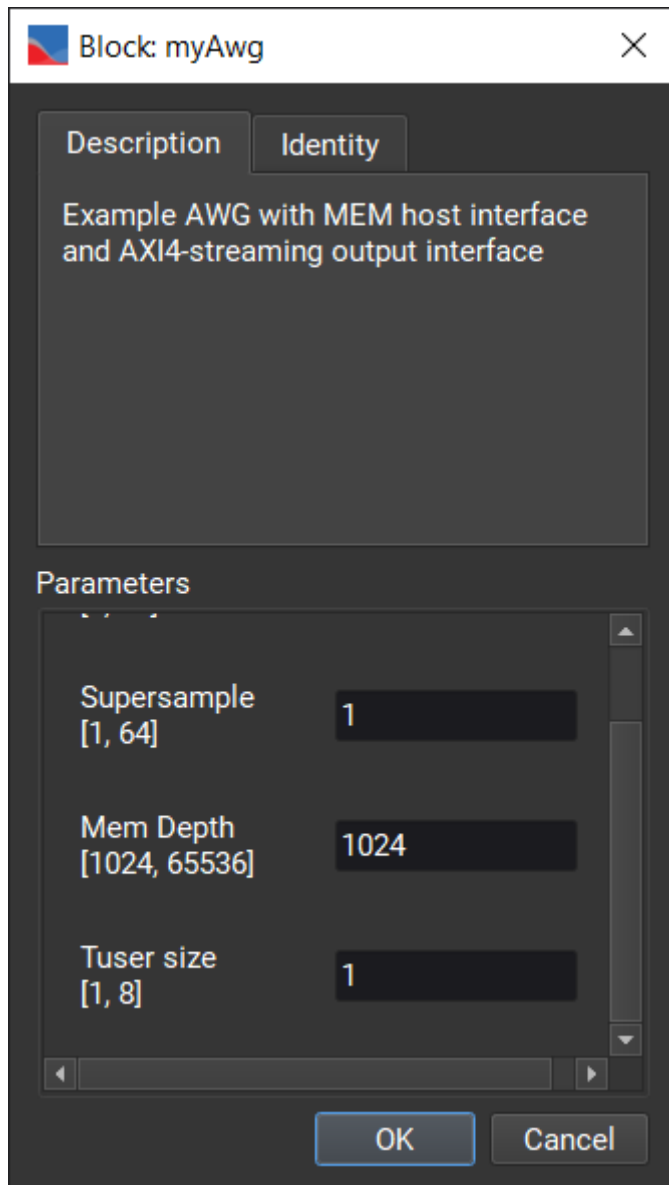
If desired, you can click *Validate* to test to see if IP Packager finds any problems with your settings:



Now we can *Save* the IP-XACT file. Note that saving the file will also do a validation and indicate any errors found. Click *Close* and *Exit* to complete and exit the IP Packager.

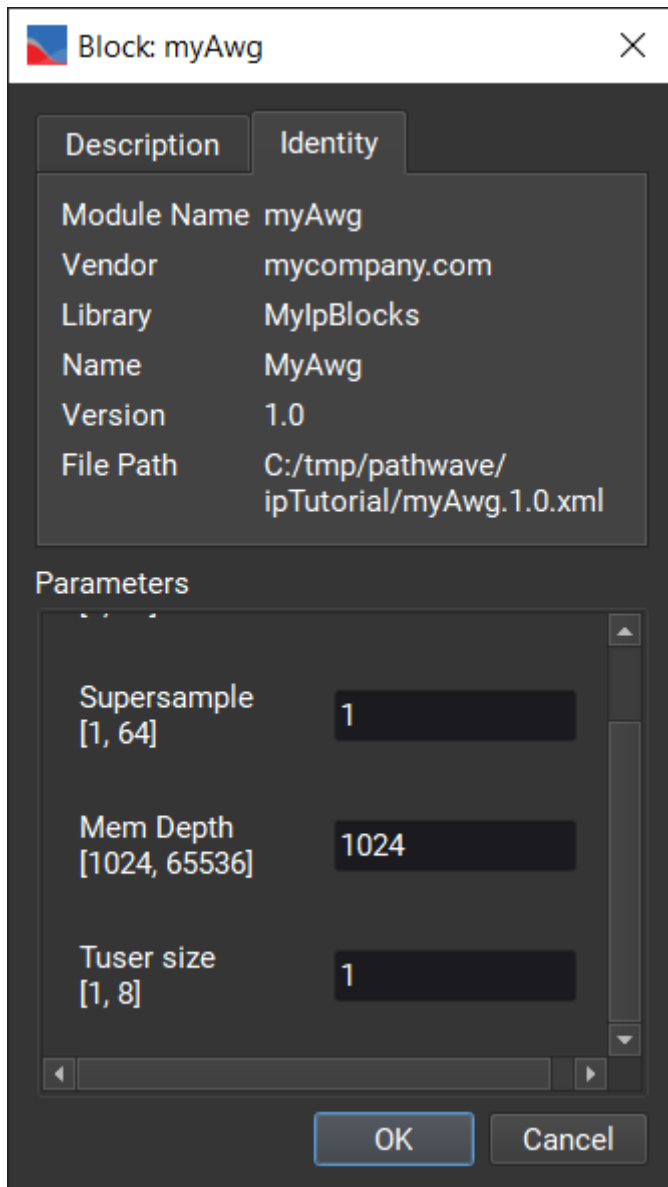
This IP is now packaged and may be used in PathWave FPGA. When the user instantiates it, they can directly modify only the user configurable parameters:





Since there are more parameters than fit on one screen, the *Parameters* window needed to be scrolled to see all of them. The non-user configurable parameters are not shown but are calculated based on these parameters.

The *Identity* tab shows the VNLV and filepath for the module:

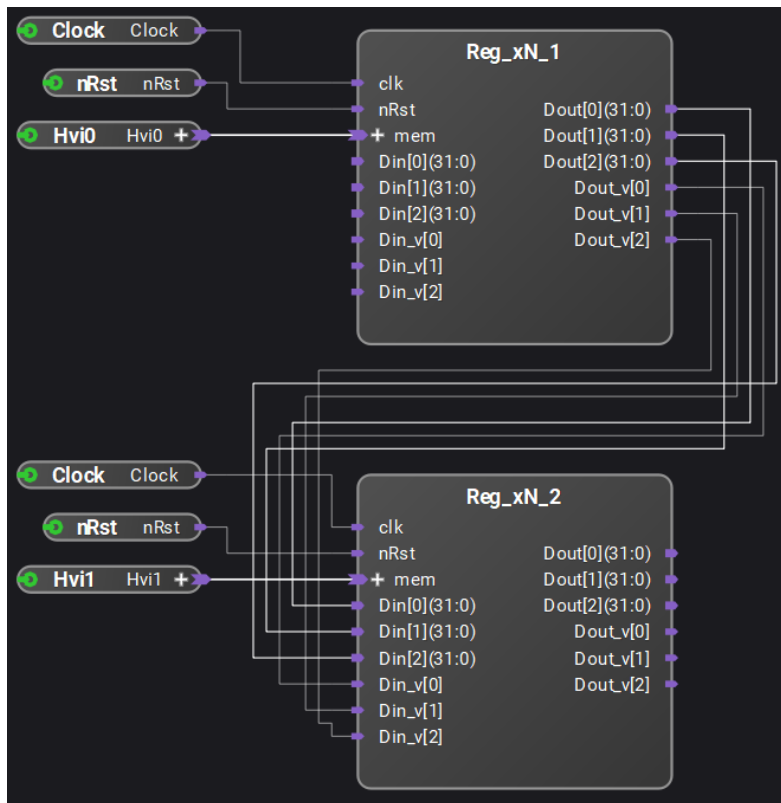


HVI Tutorial

HVI, or Hard Virtual Instrument, is a way of creating deterministic time execution sequences for one or more hardware modules using a graphical flowchart programming environment.

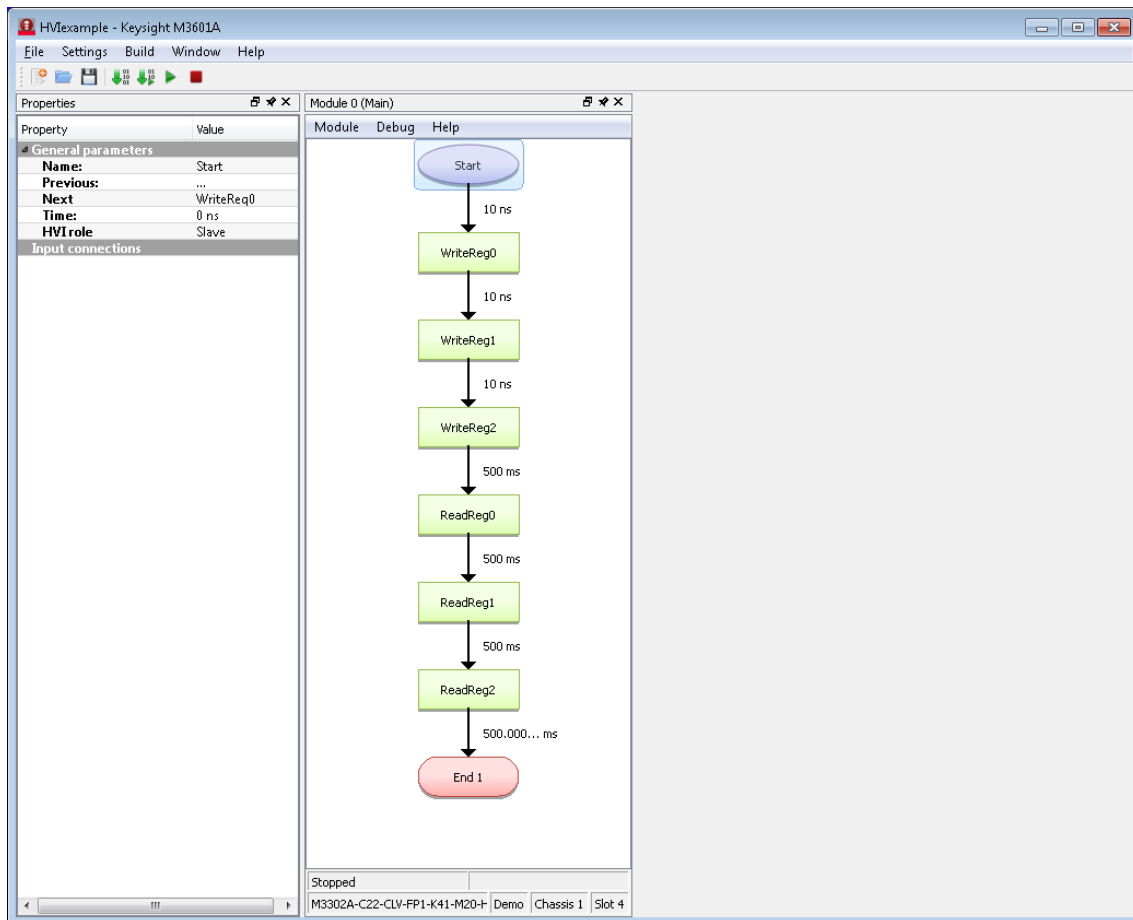
PathWave FPGA comes with an HVI example located in the examples directory of the PathWave FPGA install location. To run the HVI example you will need to install PathWave FPGA and the Keysight M3601A HVI software.

Navigate to the example directory at listed above, copy the HVlexample directory to a location with write permissions, and open the HVlexample.kfdk project file in PathWave FPGA.

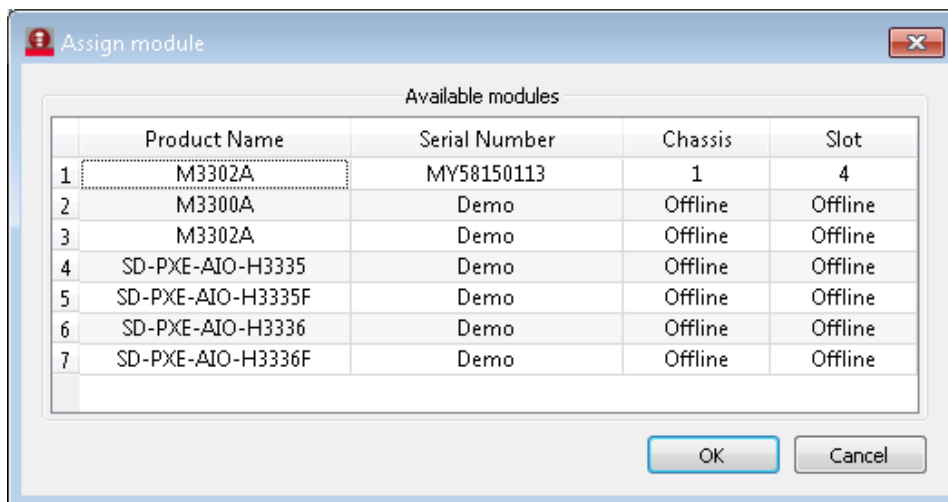


The PathWave FPGA project instantiates two register blocks each with three registers. The first register block (Reg_xN_1) connects to an HVI port called Hvi0. The second register block (Reg_xN_2) connects to an HVI port called Hvi1. The register output from Reg_xN_1 is connected to the register input of Reg_xN_2. This allows the HVI code to write values to Hvi0 and read back the values at Hvi1.

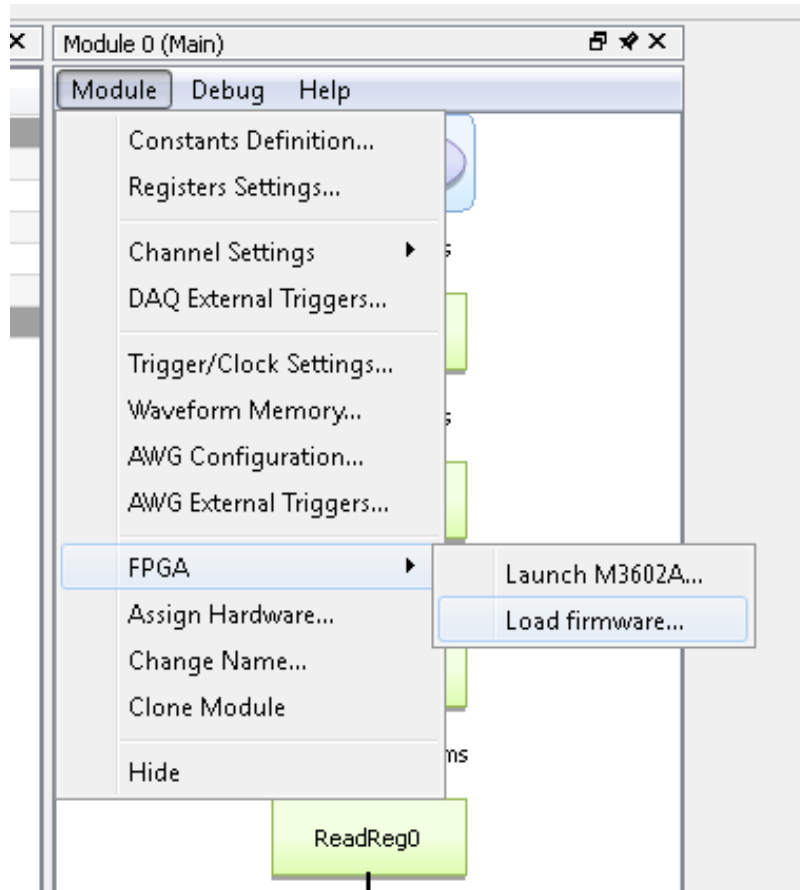
Copy the HVI project file HVlexample.HVlprj in HVlexample and the FPGA bitfile HVlexample.sbp in HVlexample\bitfile to a computer connected to an M3302A module. Start the M3601A software and open HVlexample.HVlprj.



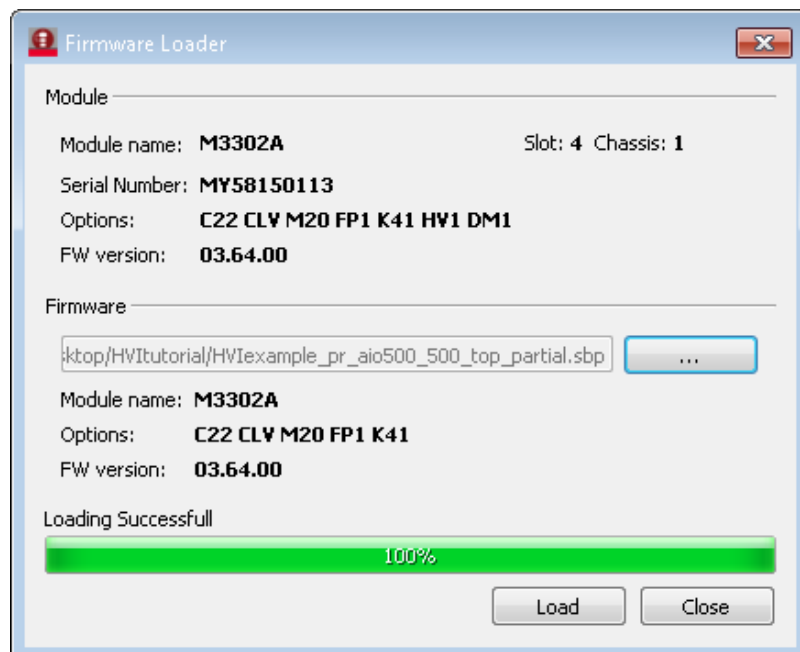
The HVI project comes setup to use the M3302A demo module. To use the real hardware module select Module→Assign Hardware... and select your hardware module.



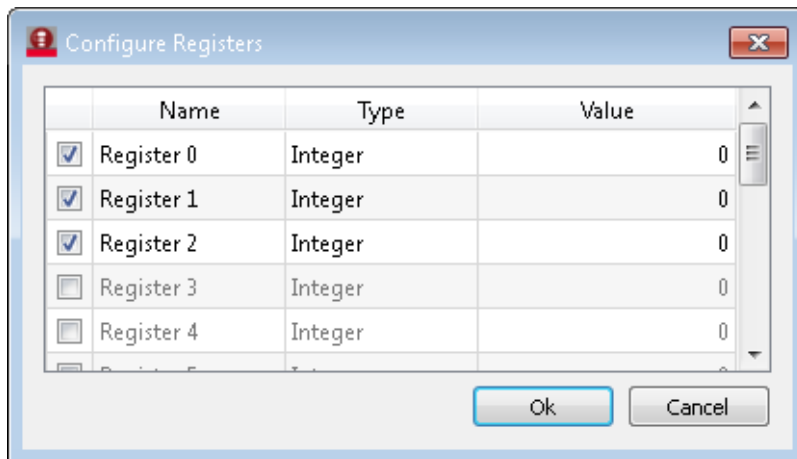
Load the FPGA bitfile onto the M3302A by selecting Module→FPGA→Load firmware...



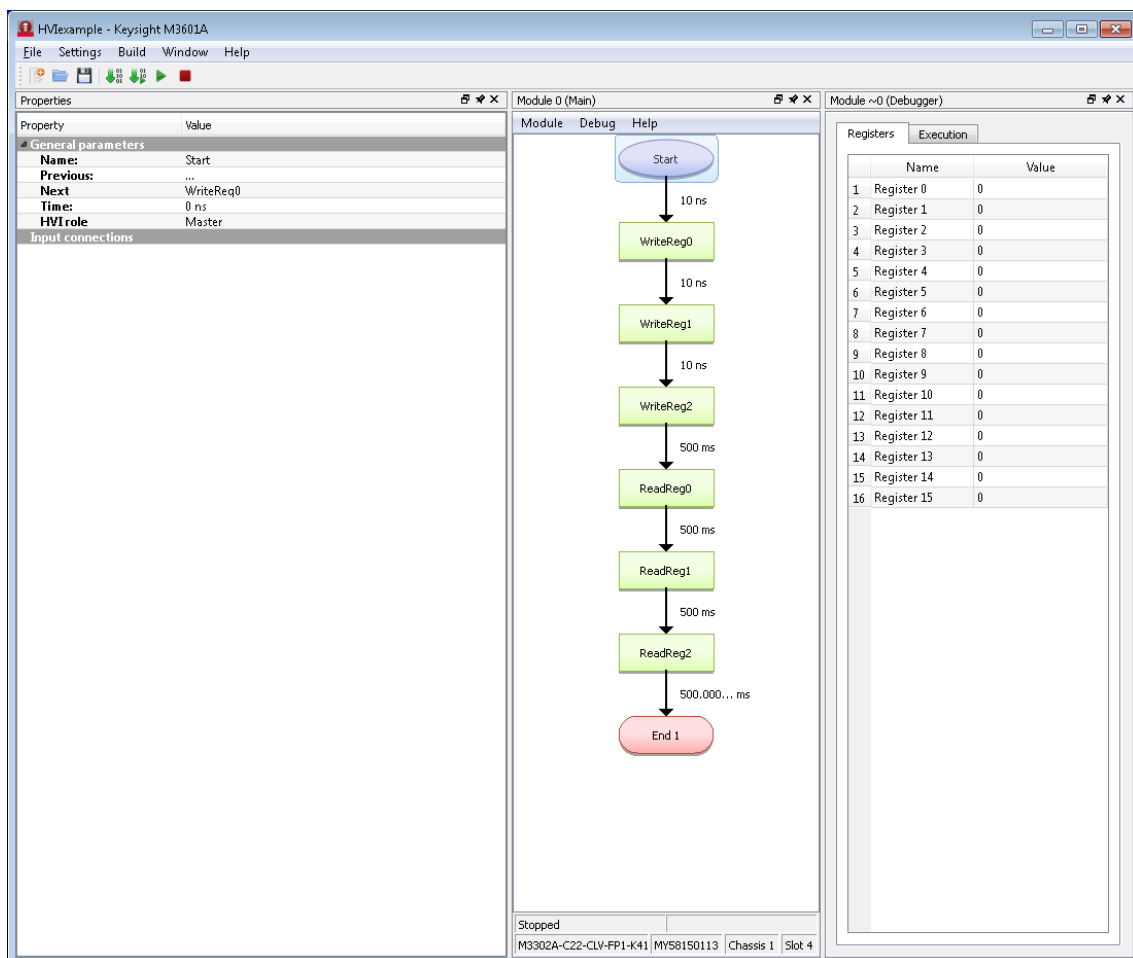
Navigate to the location of the sbp file on the computer connected to the remote hardware. Select the file and click the Load button.



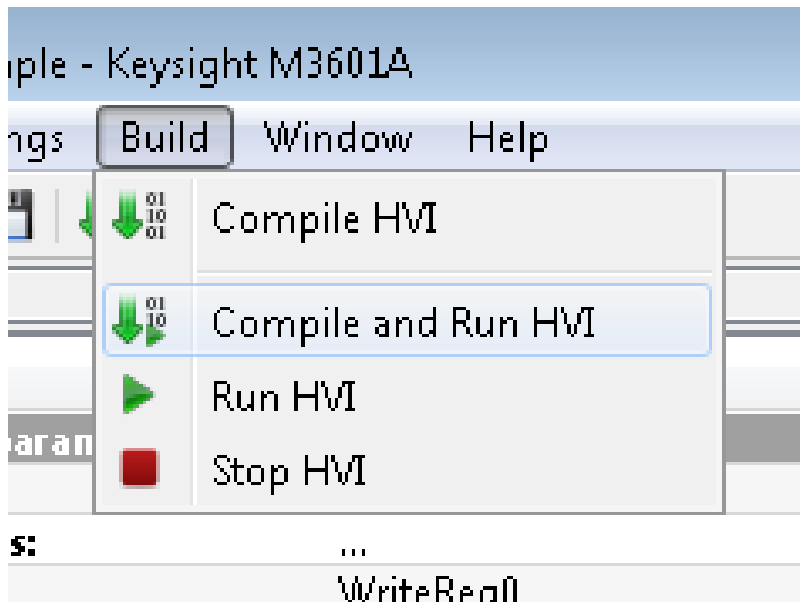
Click on Module→Register Settings... and make sure that registers 0 through 2 are enabled.



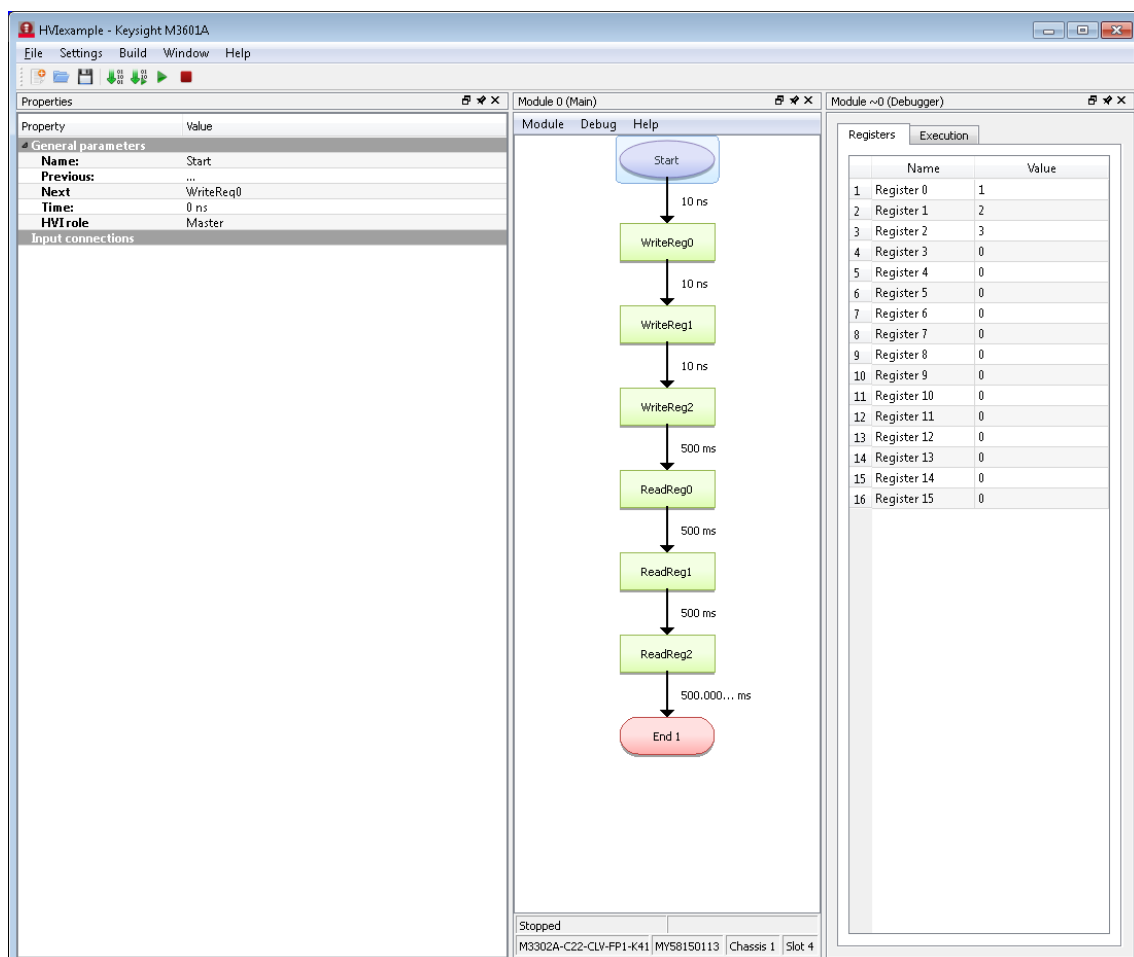
To see the register output click on Debug→Show debug window



Run the HVI example by clicking on Build→Compile and Run HVI



Once the HVI example has finished running you should see the values written to the Hvi0 port be read back on the Hvi1 port and displayed in the register debug window in the M3601A software.



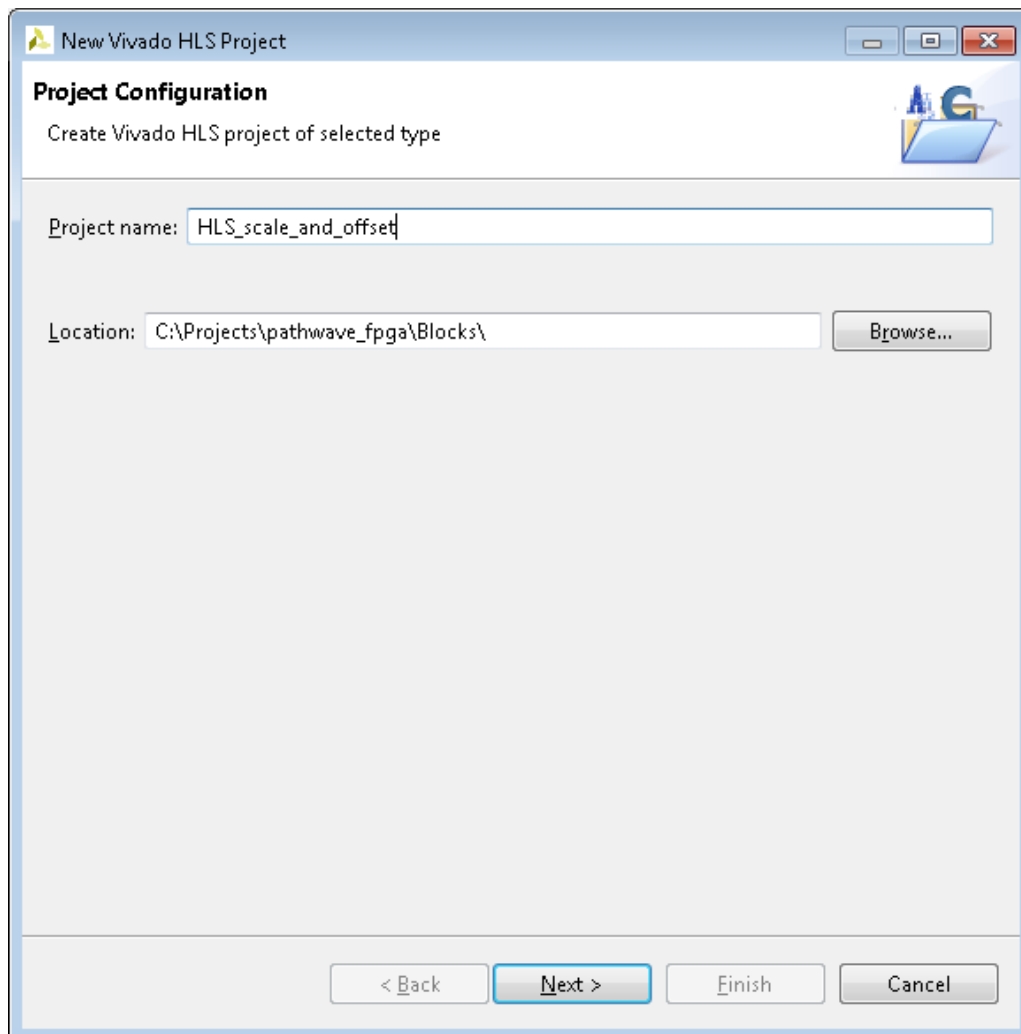
Import Vivado High-Level Synthesis (HLS) generated IP

Vivado High-Level Synthesis (HLS) accelerates IP creation by enabling C, C++ and System C specifications to be directly targeted into Xilinx FPGAs without the need to manually create HDL. This tutorial describes the creation of an IP using HLS. The design is a scale and offset circuit. The input and output data streams use an AXIS interface. The scale and offset are programmable via an AXILite interface.

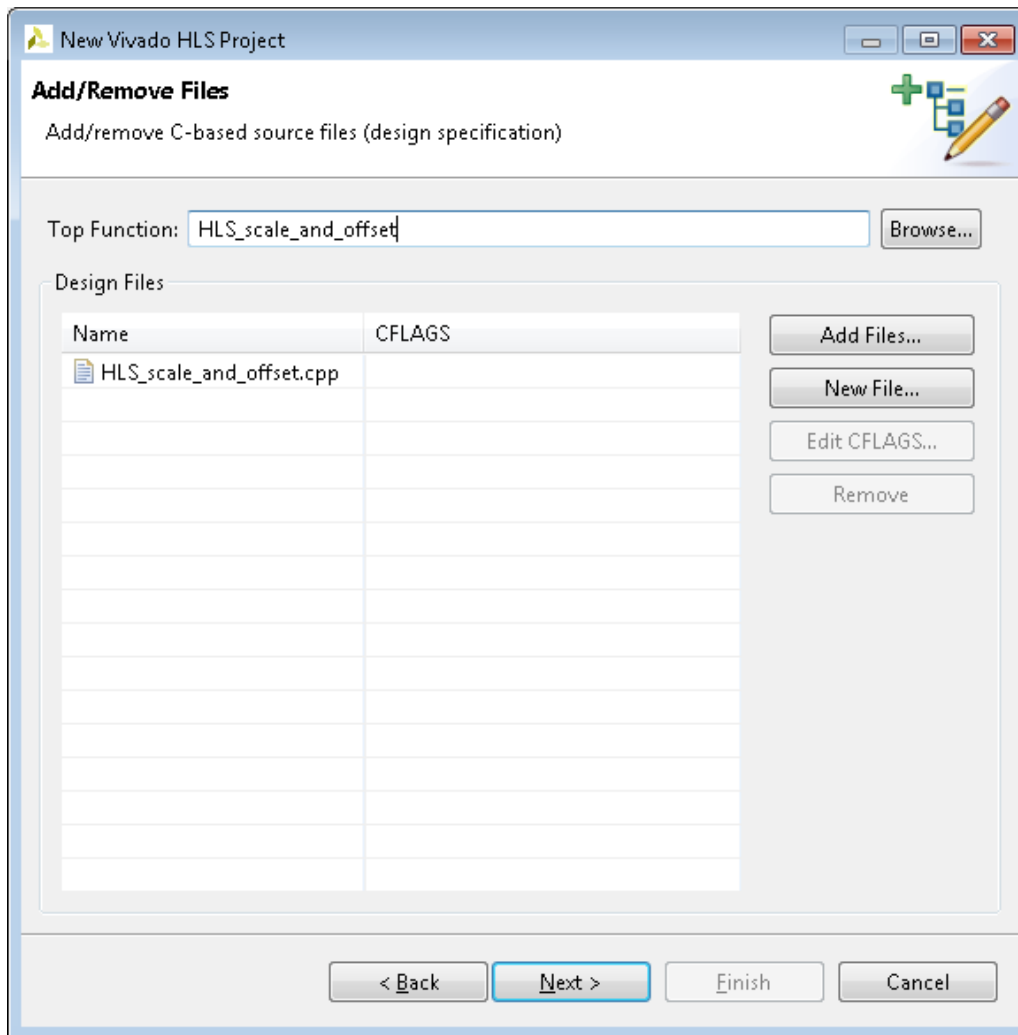
Create the Vivado HLS IP

Creating a Vivado HLS project

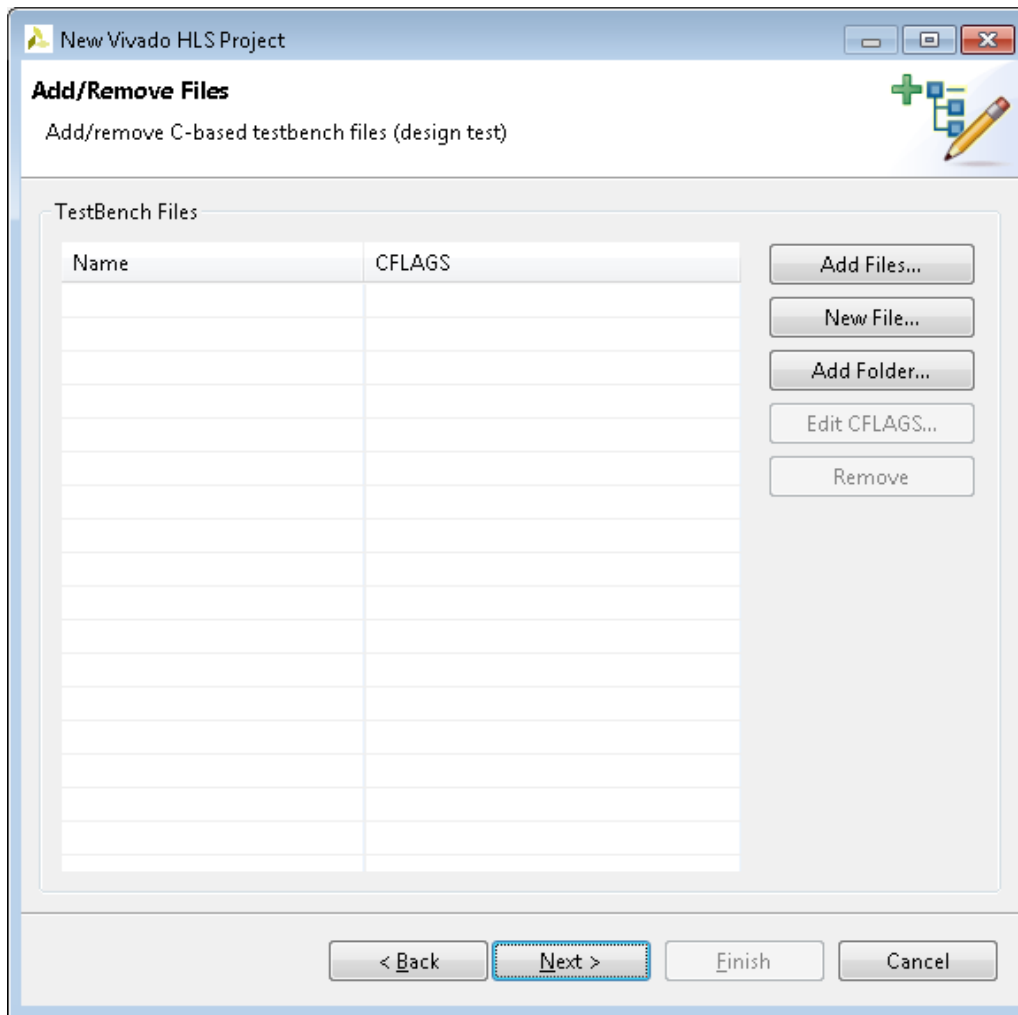
- Start Vivado HLS application
- Click on Create New Project, then set the project name to *HLS_scale_and_offset* as shown in the figure below. Then click Next.



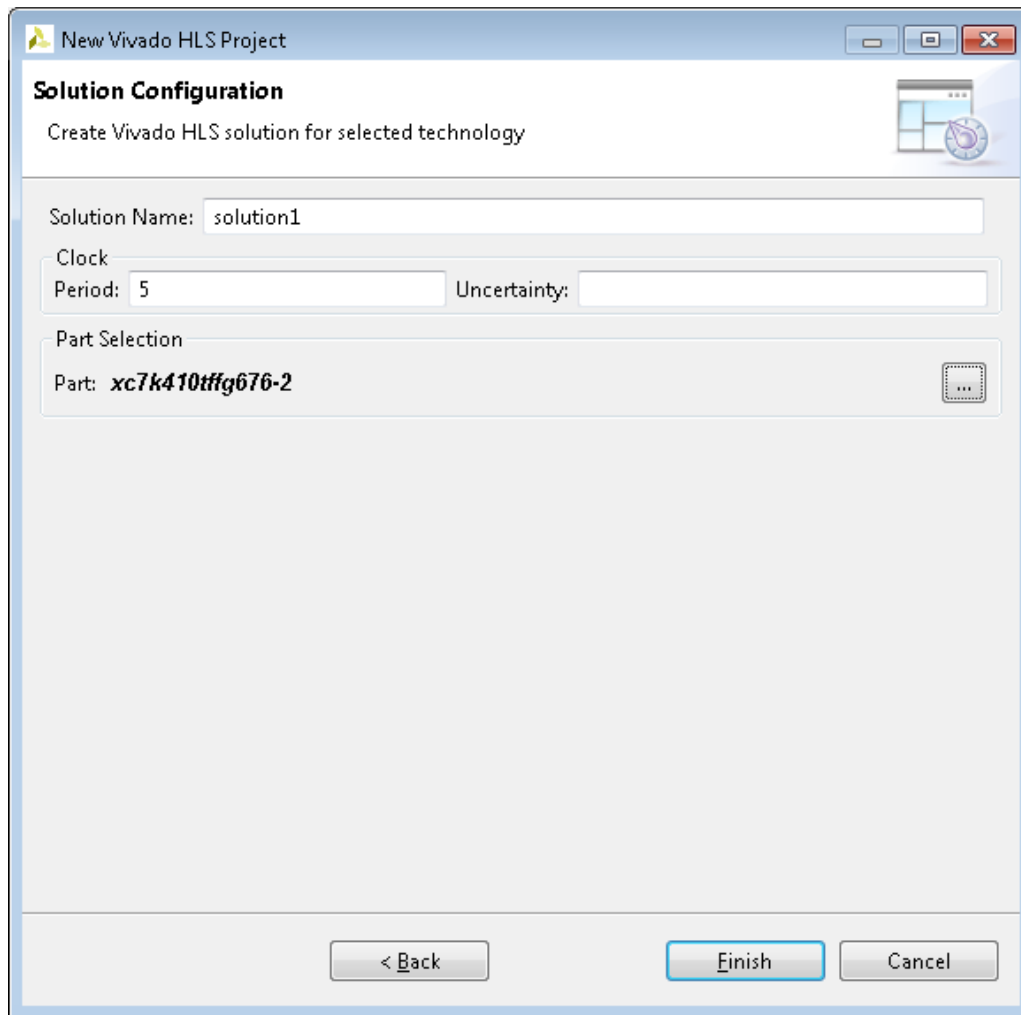
- In the next window click on *New File*, name the file *HLS_scale_and_offset.cpp* and save it in the same location as the HLS project. Then set the top function to *HLS_scale_and_offset* as shown in the figure below. Then click Next.



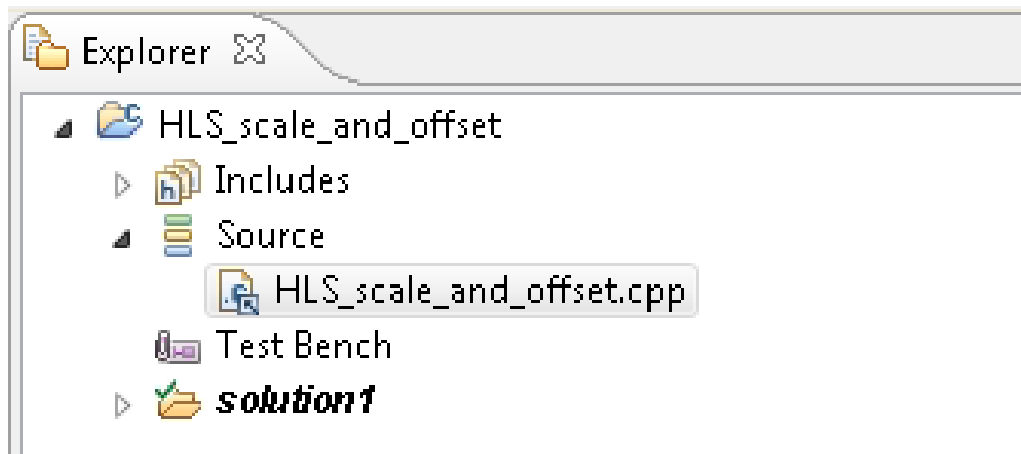
- In the next window click Next.



- In the next window set the clock period to 5 (200 MHz clock) and set the part to xc7k410tffg676-2 for the M3202A as shown in the figure below. Then click Finish.



- In the Explorer window double click on *HLS_scale_and_offset.cpp*.



Implementing the IP function in C

- To create the IP function in C, open the file *HLS_scale_and_offset.cpp* and paste the following code block:

Code Block 2 HLS_scale_and_offset.cpp

```

#include <ap_fixed.h>
#include <hls_stream.h>

using namespace hls;

typedef ap_fixed<16, 1, AP_TRN, AP_SAT> SAMPLE_T;
typedef stream<SAMPLE_T> SAMPLE_FIFO_T;



void HLS_scale_and_offset(SAMPLE_FIFO_T data,
                          SAMPLE_T scale,
                          SAMPLE_T offset,
                          SAMPLE_FIFO_T output)
{
#pragma HLS PIPELINE II=1 enable_flush
#pragma HLS INTERFACE axis register both port=output name=DataOut
#pragma HLS INTERFACE axis register both port=data name=DataIn
#pragma HLS INTERFACE s_axilite register port=scale bundle=Control
#pragma HLS INTERFACE s_axilite register port=offset bundle=Control
#pragma HLS INTERFACE s_axilite port=return bundle=Control

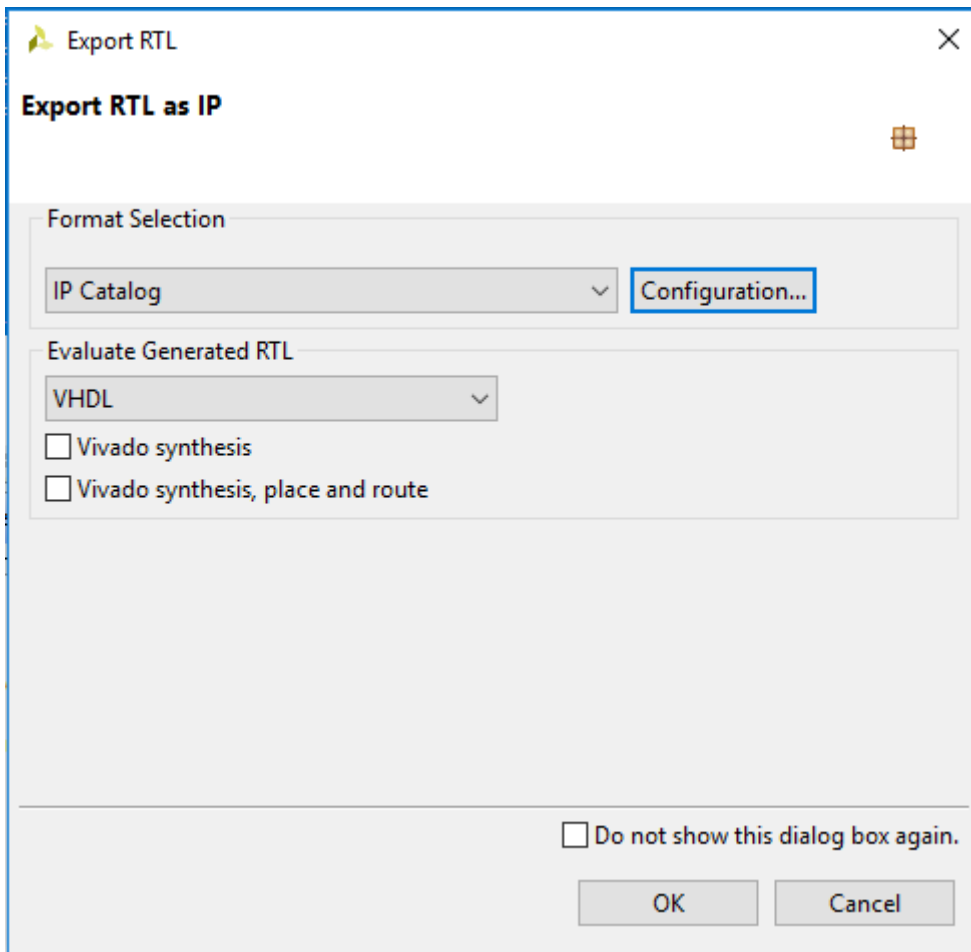
    SAMPLE_T product;

    data >> product;
    product = (product * scale + offset);
    output << product;
}

```

Generating the synthesizable HLS IP

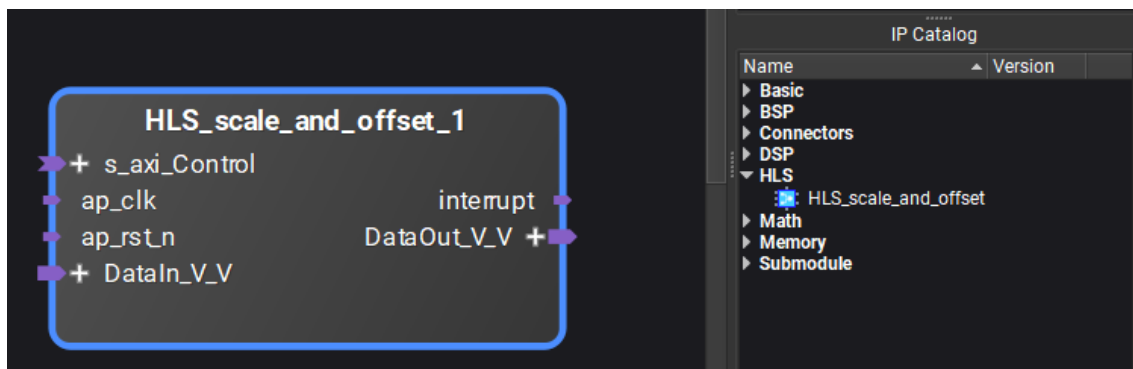
- Next, click the *C Synthesis* button .
- Next, click on the *Export RTL* button . Make sure 'IP Catalog' is selected and click OK.



Using the Vivado HLS IP in PathWave FPGA

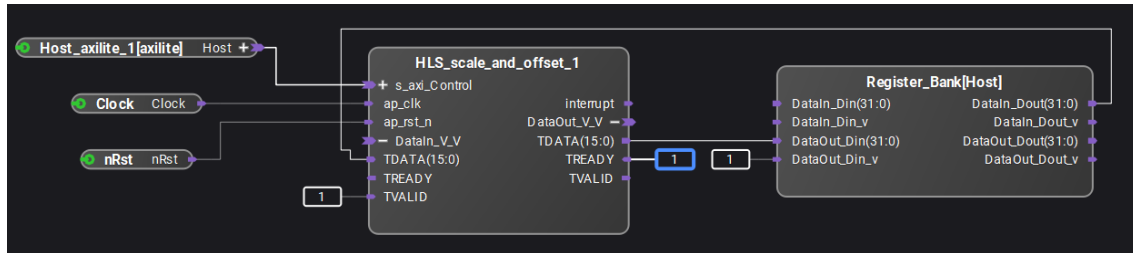
Importing the HLS IP into a project

- Start PathWave FPGA
- Create any project using the M3202A BSP. Even though the HLS IP was created for FPGA part xc7k410tffg676-2, it can be used with any Kintex7 family board, as long as the clock is 200 MHz.
- Click on Add External Block and browse to the location of the HLS project. Inside the HLS project directory, go to sub-directory *solution1/impl/ip* and select file *component.xml*.
- Click Open. This will bring the newly created HLS IP into the *PathWave FPGA* project.
- At this point, you are ready to use the IP in your design.



Create a design using the HLS IP

A sample design is the following:



This design is using the *HLS_scale_and_offset* IP that was created earlier. It has its clock and reset ports connected to the sandbox's clock and reset signals. The input and output ports of the block are connected to a host register bank. Note that only the 16 LSBs of the register bank values are used since the *HLS_scale_and_offset* block is using 16 bit data.

To control the value of scale and offset, the sandbox's Host interface is connected to the *s_axi_Control* interface of the IP at an address offset of 0x0000. The register bank is at an address offset of 0x1000.

Note that the scale, offset, and data values in this example are fixed point numbers representing values +/-1. Thus for a gain of 0.999 one should program a gain value of 0x7fff.

At this point, we can run the Implementation process to generate the bitstream file to be loaded into the FPGA.

Running design on FPGA

Before we can be able to see valid data from the output channel, we need to set the scale and offset values and start the HLS IP. This can be done using the Host interface and write to the following addresses (this information is provided in the file *<HLS Project Directory>/solution1/impl/ip/drivers/HLS_scale_and_offset_v1_0/src/xhls_scale_and_offset_hw.h*):

```
// 0x00 : Control signals
//      bit 0 - ap_start (Read/Write/COH)
//      bit 1 - ap_done (Read/COR)
//      bit 2 - ap_idle (Read)
//      bit 3 - ap_ready (Read)
//      bit 7 - auto_restart (Read/Write)
//      others - reserved
// 0x10 : Data signal of scale_V
//      bit 15~0 - scale_V[15:0] (Read/Write)
// 0x18 : Data signal of offset_V
//      bit 15~0 - offset_V[15:0] (Read/Write)
```

To start processing data through the HLS block, you need to write 0x81 to the control signals (address 0x00). This will start the HLS design and cause it to continually process data until the *auto_restart* bit is cleared.

Power of Two Decimation Tutorial

Purpose of Tutorial

This tutorial will show how to create a design that uses a power-of-two decimator that streams data into and out of the DDR memory.

Requirements

1. PathWave FPGA
2. M3XXXA BSP
3. Vivado 2017.3 or newer
4. Microsoft Visual Studio
5. CMake 3.11.3 or newer

Description of Decimator Design

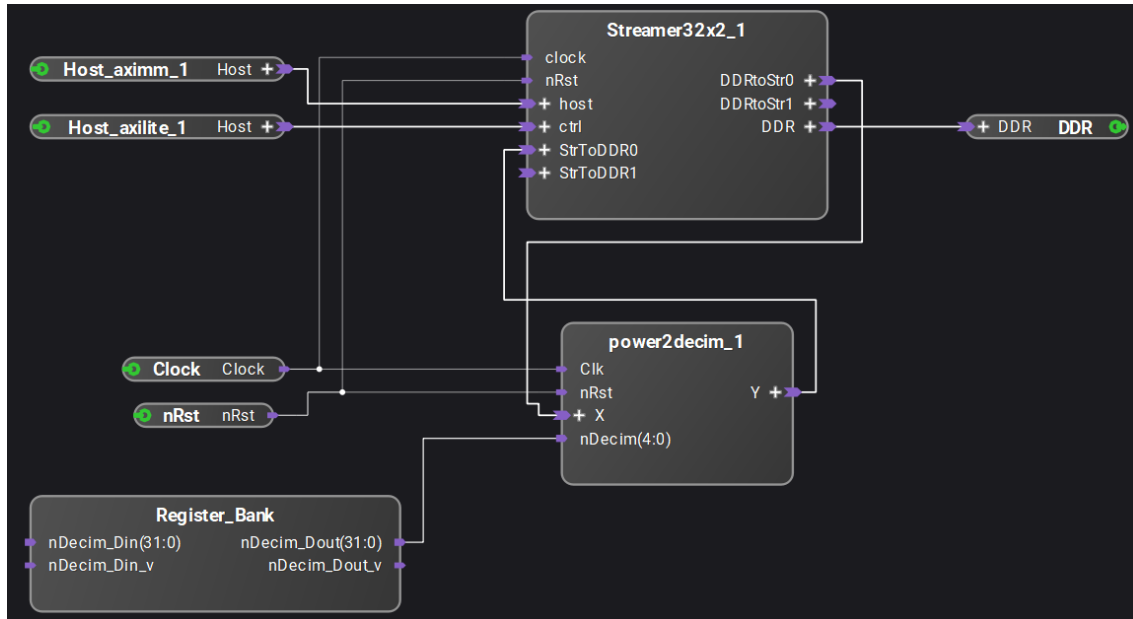
This design uses a power-of-two decimator block written in Verilog HDL. This block implements a two channel (which could be used for complex (real/imaginary) data, or could be used for two independent data streams) decimation filter that accepts input samples at up to one sample per clock, and outputs at a relative rate of $1/2^n$, where n can be 1 to 16. Since the data comes from DDR memory and the results go back into DDR memory, there is no real sample rate - the data is just samples. The data could represent data that was sampled at any arbitrary rate. For convenience in this tutorial, we'll assume the input samples represent data sampled at 100 Ms/s. In that case, the output of the decimator would represent data rates of $100/2^n$ Ms/s or 50 Ms/s, 25 Ms/s, 12.5 Ms/s, ... , 1.526 ks/s.

The decimation process has two parts. First the data is low pass filtered to protect against aliasing, and second the sample rate is reduced by throwing away samples. This particular filter is implemented as 16 cascaded stages with each stage low pass filtering to half the bandwidth and then discarding every other sample to half the data rate. Then the output of one of these stages is selected as the output of the decimator block. This is selected via the `nDecim(4:0)` port. The `nDecim=0` value represents no decimations (the data just passes through) while `nDecim=1..16` selects one of the 16 stages for output. Values larger than 16 should not be used.

The inputs and output ports of the `power2decim` block uses AXI-streaming interfaces. These interfaces use a data bus width of 32 bits, with the 16 low order bits being one input channel and the 16 high order bits being the other input channel. Since these data streams go into and out of DDR, the `power2decim`'s AXI-streaming interfaces must include both forward and reverse flow control using the `TVALID` and `TREADY` signals.

Access to DDR memory is via an addressable random access AXI bus. The `power2decim` block uses non-addressable streaming AXI-streaming interfaces. To facilitate the use of streaming data, Keysight provides a `Streamer32x2` IP block. This block contains 2 independent channels each of which has a read stream and a write stream. In this design, only one of the two channels are used. Inside the `Streamer32x2` block are DMA engines that can be programmed to read or write DDR data and convert this to 32 bit wide streaming data.

In addition, there is a `Register_Bank` consisting of one register that is used to select which of the 16 available output bandwidths is selected.



Description of Test Software

The test software is in a Microsoft Visual Studio solution and consists of the main C++ code in main.cpp as well as driver class for the Streamer32x2 IP in ddr.h and ddr.cpp.

This program treats the two channels of the power2decim as independent channels. Here is the basic flow of the program:

1. Find and load the bit file for this test design.
2. Open a connection to the hardware module.
3. Initialize the driver code for controlling the streamer32x2 block.
4. Program the desired number of decimations into the nDecim register in the register bank. In this tutorial, 3 passes of decimation are selected, so the output sample rate is $1/2^3$ or $1/8$ the input sample rate.
5. Create the two test waveforms in memory (described below). One channel is the odd samples, the other channel is the even samples.
6. Write this waveform data into the hardware DDR memory.
7. For diagnostic purposes, we pre-fill the destination area of DDR with a known pattern. In normal usage you would not do this.
8. The two DMA channels (inside the streamer32x2 block) are programmed. Once both channels are configured, data flows from the DDR through the power2decim and back into DDR.
9. The code waits for the Streamer Write DMA operation to finish.
10. For diagnostic purposes, pre-fill the host memory buffer with a known pattern. In normal usage you would not do this.
11. Read the waveform data from DDR into a host buffer.
12. Write the results data into standard output.
13. Release resources to clean up and end.

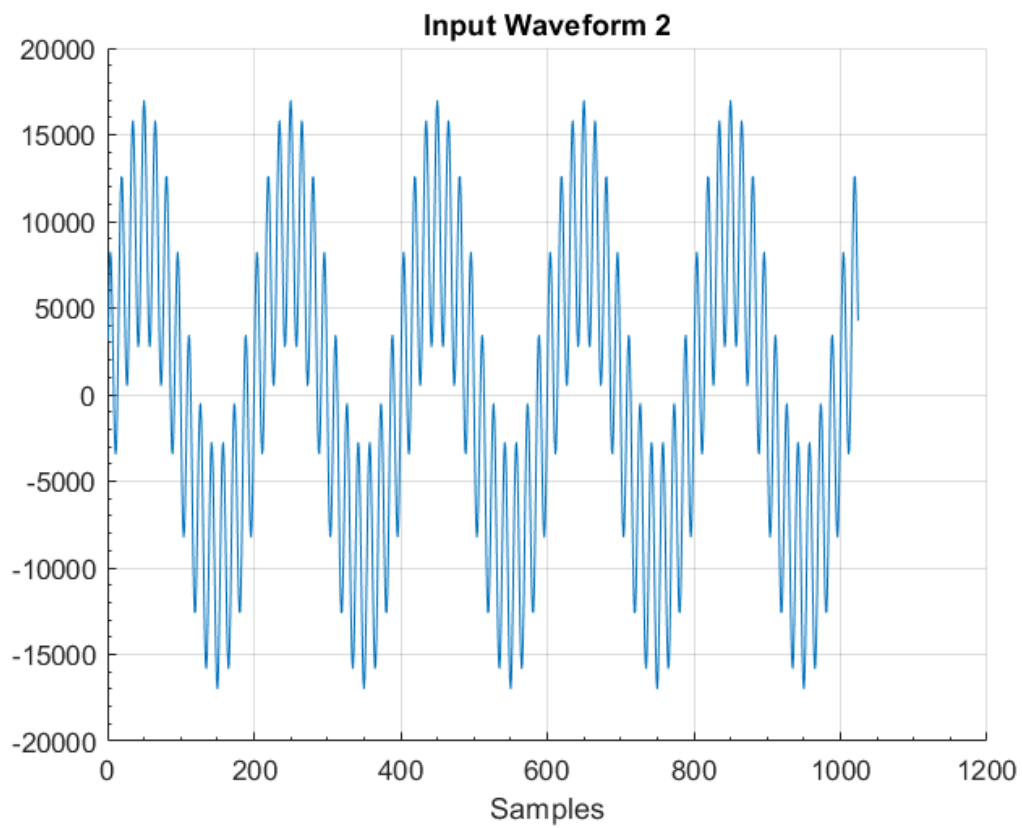
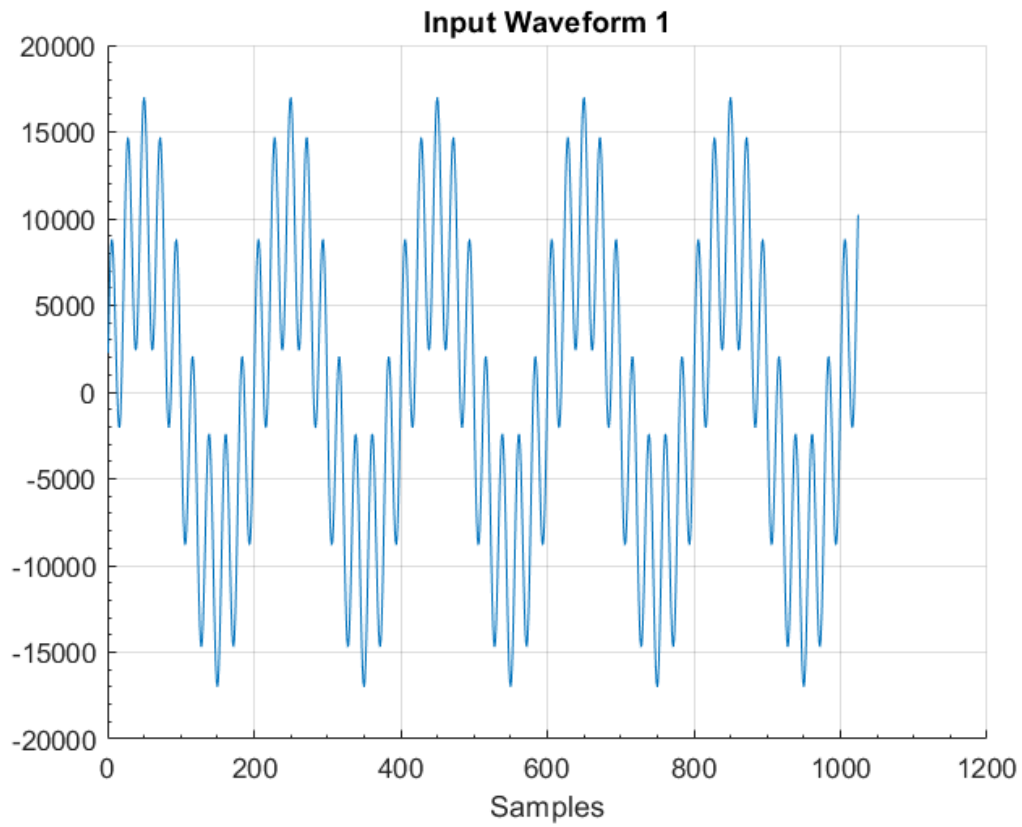
Test Signal Description

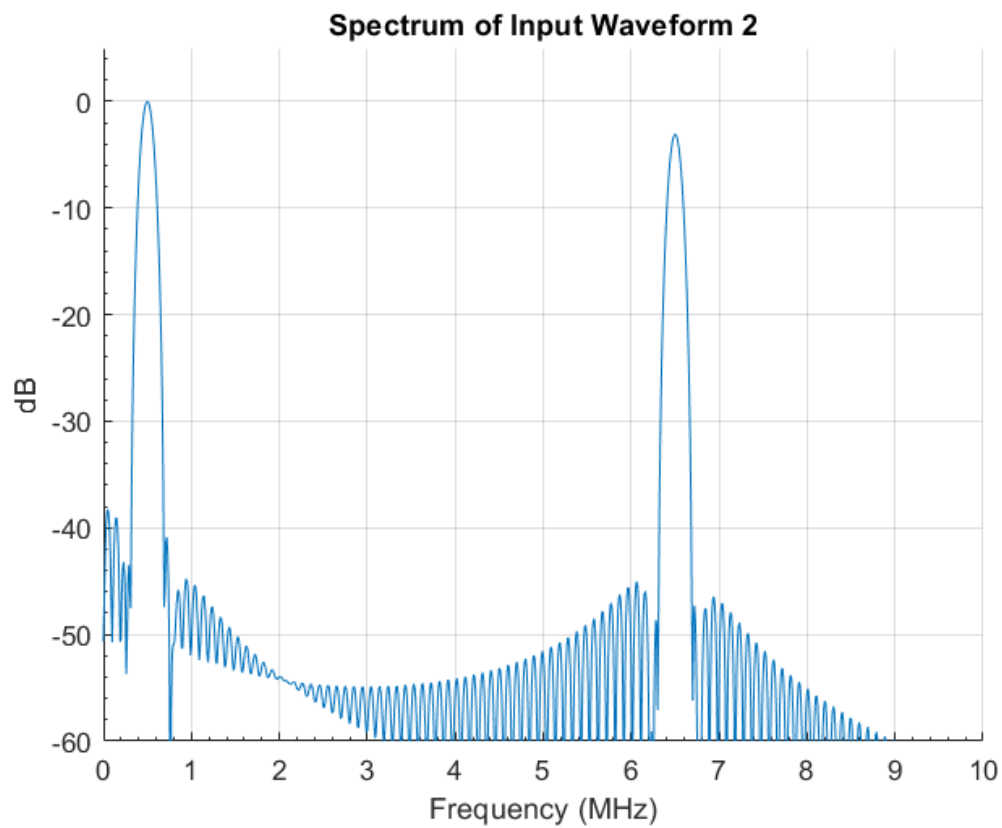
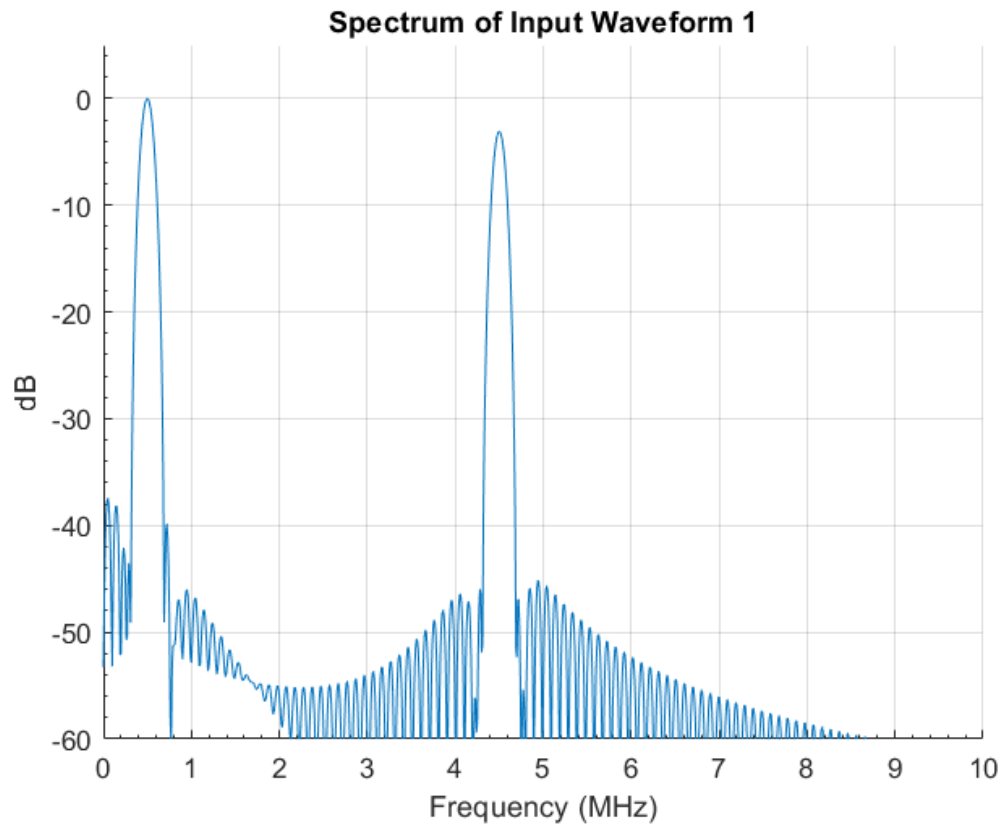
As noted above, this design just uses sampled data which could represent any sample rate, but for the purposes of discussion, we will assume an input sample rate of 100 Ms/s.

This tutorial uses the two channels of the power2decim as independent channels rather than as one complex channel. The two channels are interleaved in memory, with one channel occupying the even samples and the other channel occupying the odd samples in the buffer.

This tutorial uses 3 passes of decimation, so the output sample rate is $1/2^3$ or $1/8$ the input sample rate or $100/8 \text{ Ms/s} = 12.5 \text{ Ms/s}$. The passband is approx. 60% of Nyquist or 3.75 MHz. The stopband starts at 6.25 MHz. Between 3.75 and 6.25 MHz is the transition band of the filter.

One input signal is a tone at 0.5 MHz with a second tone, 3 dB smaller, at 4.5 MHz. The other input signal has the same 0.5 MHz tone, this time with a second 6.5 MHz tone 3 dB lower. Below is shown the time domain waveforms as well as the spectrum of the two signals (note: even though the input bandwidth extends up to 50 MHz, only the lower 10 MHz are shown for clarity).

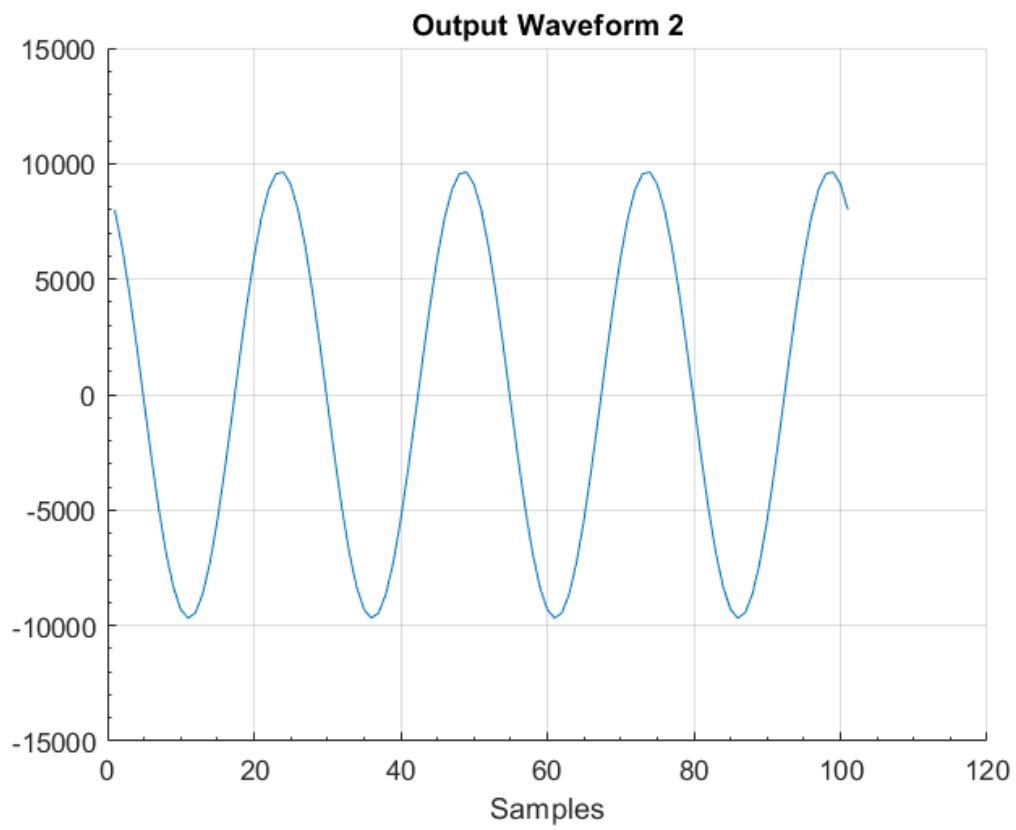


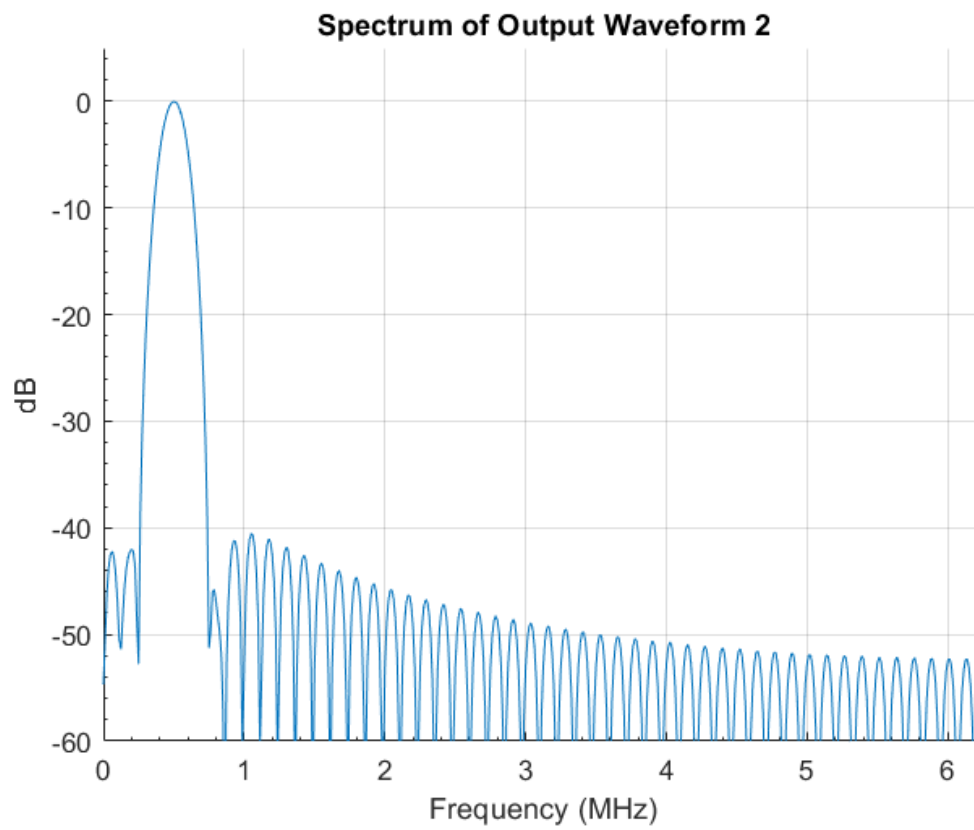
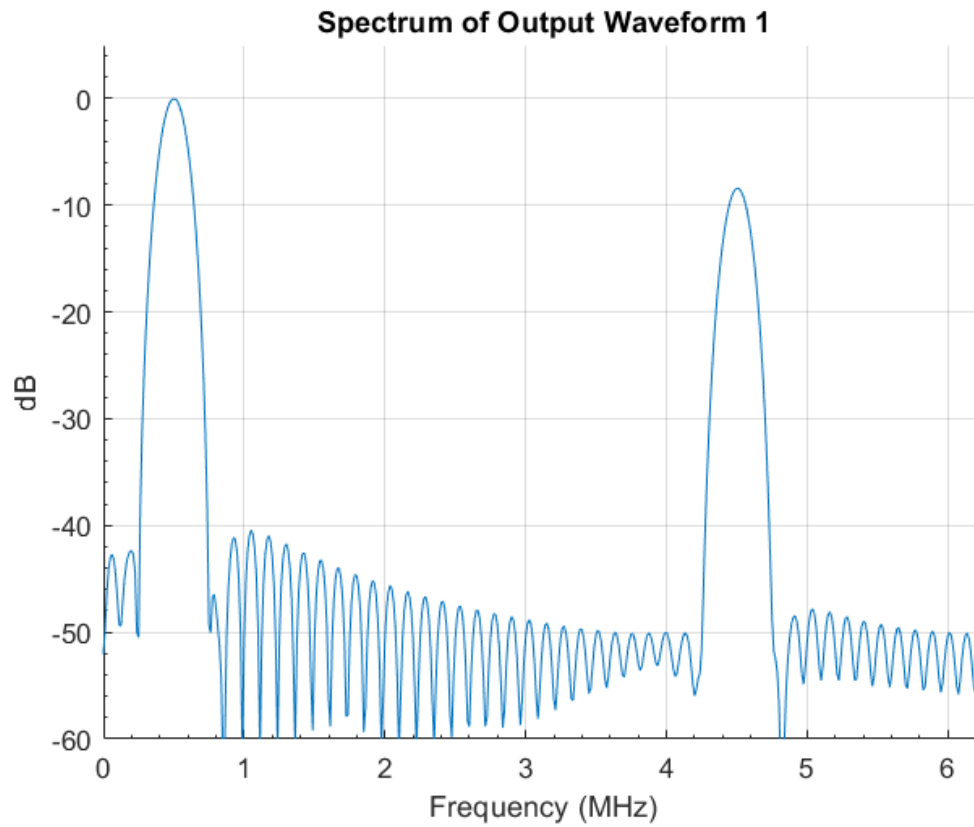


Both of these channels are low pass filtered and then decimated by 8. The 0.5 MHz component of both signals is well within the passband of the decimator and passes through unchanged. The first signal has a tone component at 4.5 MHz. This falls in the transition band

of the decimation filter. Thus the 4.5 MHz signal is attenuated (by about 5.4 dB) but partially passes through. The second signal's component at 6.5 MHz is completely in the stopband of the decimation filter and is completely removed.

Below are shown the time domain and spectrum of the output signals as read back from the hardware. Note that the x-axis scaling of the output waveforms is different that the input waveforms. This is due to the sample rate of the output being 1/8 of the sample rate of the input.



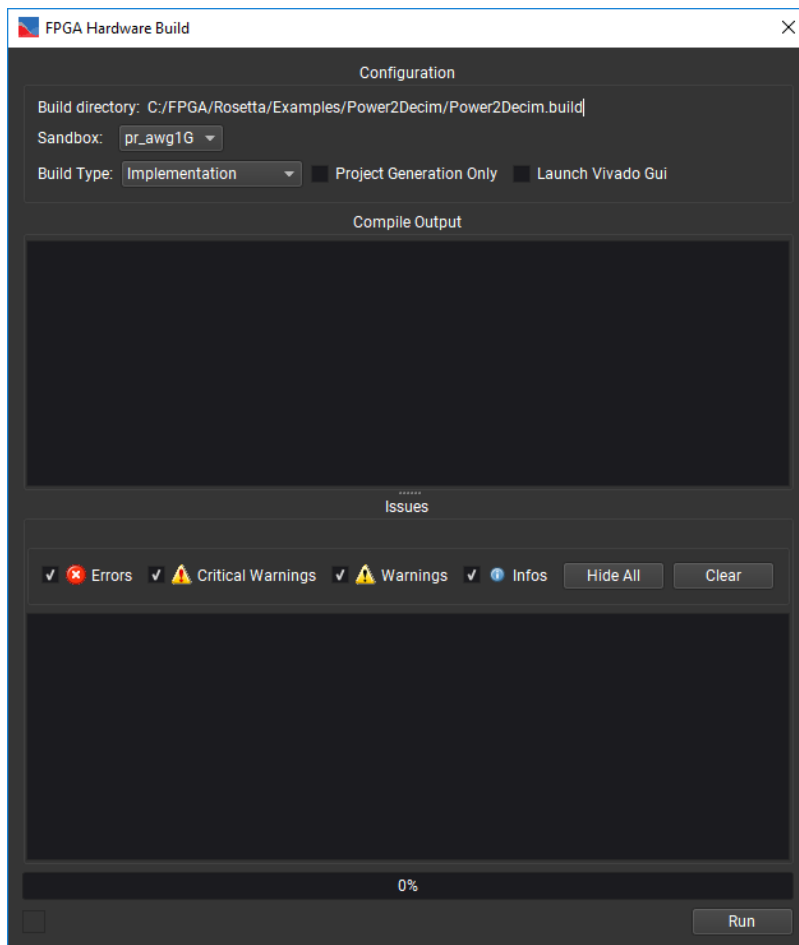


Building the Bitfile

Navigate to the examples directory under the PathWave FPGA install directory. Copy the Power2Decim directory to a location with write permissions. Open the Power2Decim.kfdk file in PathWave FPGA. The Power2Decim PathWave FPGA project is currently built for the 4 channel variable clock M3202A module.

If you have a different module or BSP, then you will need to retarget the PathWave FPGA project for your module. To retarget the project select *File→Retarget Project...* in the PathWave FPGA GUI. See Power2Decim\readme.md for more details.

To build the Power2Decim project select the *Project→Generate Bit File...* menu pick or click on the *Generate Bit File...* icon in the toolbar. Make sure the *Build Type* is set to *Implementation* and that the *Project Generation Only* and the *Launch Vivado Gui* boxes are unchecked. Then click the *Run* button to start the FPGA build process.



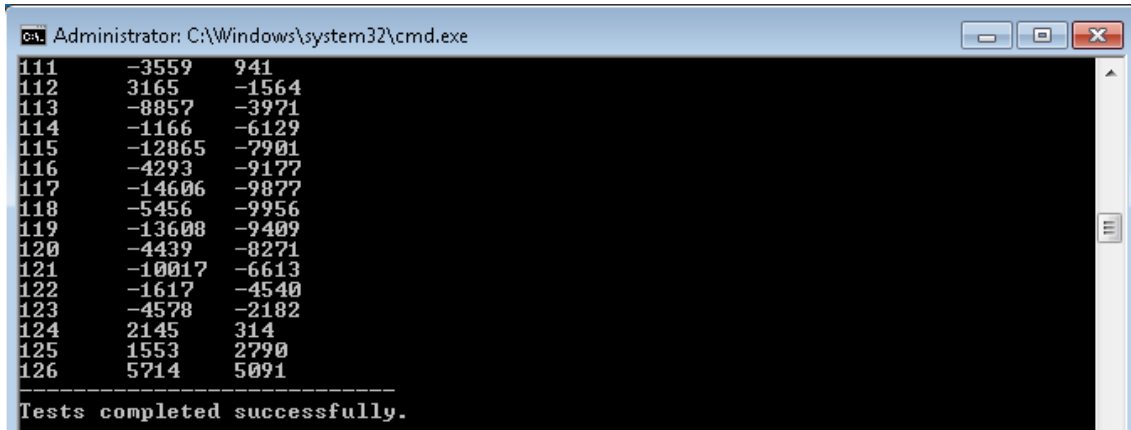
When the build is completed, there will be a directory in the project directory called *<projectName>.build*. Inside this will be one (or more) directories for each build process. They will have the project name as well as the date stamp. Inside the directory for the last build, there will be a directory called *bitfiles*. Inside that directory will be the *<projectName>.k7z* file. This file contains the necessary information for loading the routed design into the hardware module.

Running the C++ Example

Follow the steps below to run the C++ example and load the Power2Decim.k7z bitfile example onto the hardware module:

1. Navigate to the examples directory under the PathWave FPGA install directory.
2. Copy the Power2Decim project from Program Files to a location with write permissions.

3. To create the Visual Studio C++ project, follow the instructions in Power2Decim\readme.md.
4. Navigate to the build directory and open the Visual Studio solution (the default solution name is Power2Decim.sln).
5. Build the C++ example program and copy the Debug or Release folder to a PC connected to the M3XXXA module.
6. Run the C++ example.



```

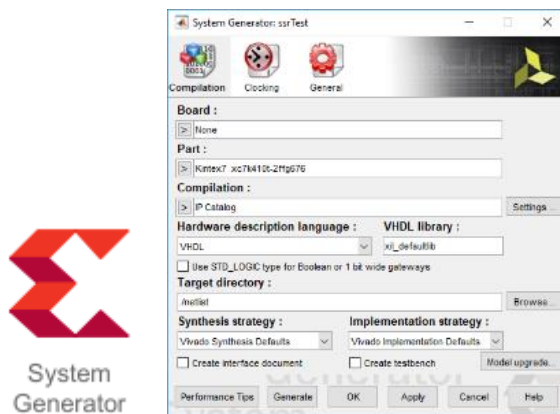
Administrator: C:\Windows\system32\cmd.exe
111      -3559      941
112      3165     -1564
113     -8857     -3971
114     -1166     -6129
115    -12865    -7901
116     -4293     -9177
117    -14606    -9877
118     -5456    -9956
119    -13608    -9409
120     -4439    -8271
121    -10017    -6613
122     -1617    -4540
123     -4578    -2182
124      2145      314
125      1553      2790
126      5714      5091
-----
Tests completed successfully.

```

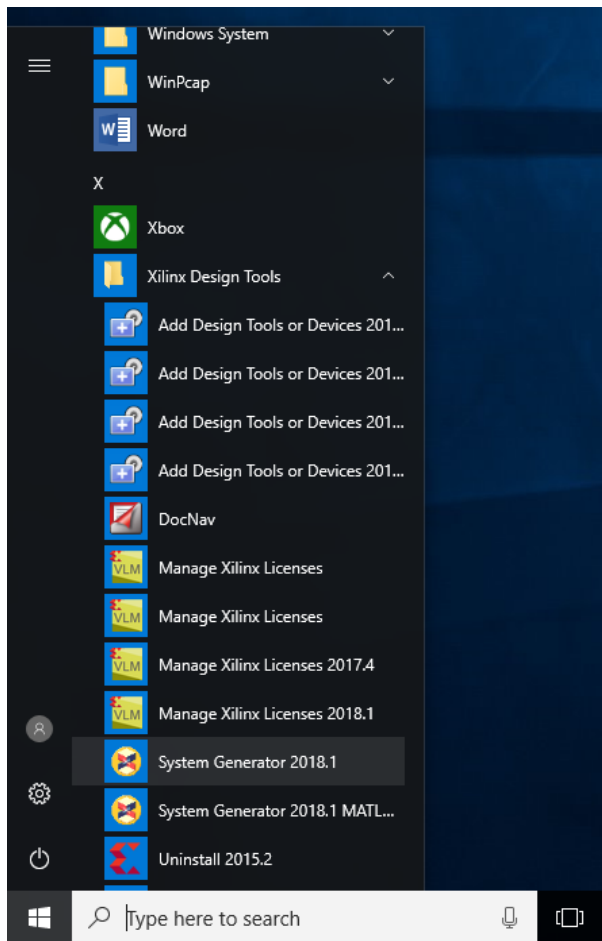
Xilinx System Generator for DSP™ Tutorial

The Xilinx System Generator for DSP™ is a Simulink library blockset that can be used for creating FPGA designs from Simulink. The System Generator library blocks instantiate Xilinx IP like filters, adders, multipliers, CORDIC, etc. The output of System Generator can be easily imported into PathWave FPGA.

Before importing your Xilinx System Generator design into PathWave FPGA, make sure there are no build errors when generating the HDL code from System Generator. For help getting started with System Generator, see the Xilinx document *Model-Based DSP Design Using System Generator* in DocNav (UG948). You can also find System Generator documentation by clicking on the *help* button in the System Generator Simulink library block.



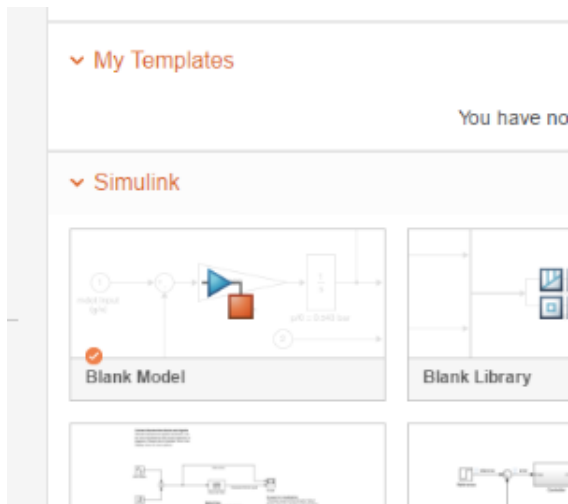
- Run System Generator from the start menu.



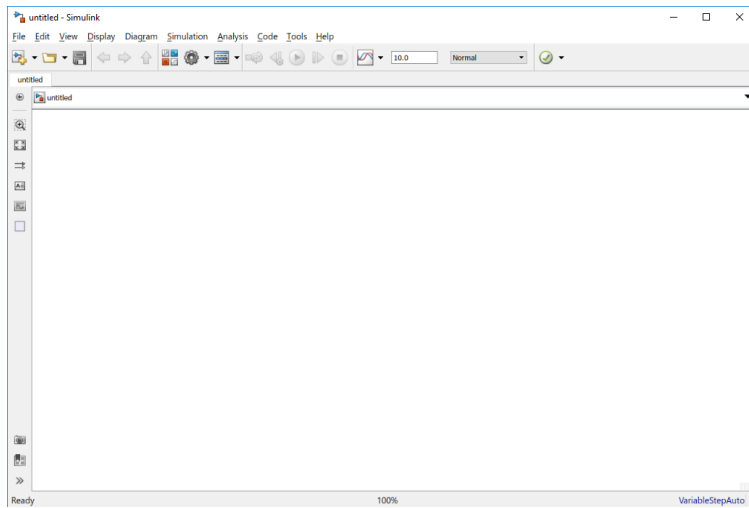
- Start Simulink by clicking on the Simulink icon on the Matlab toolbar.




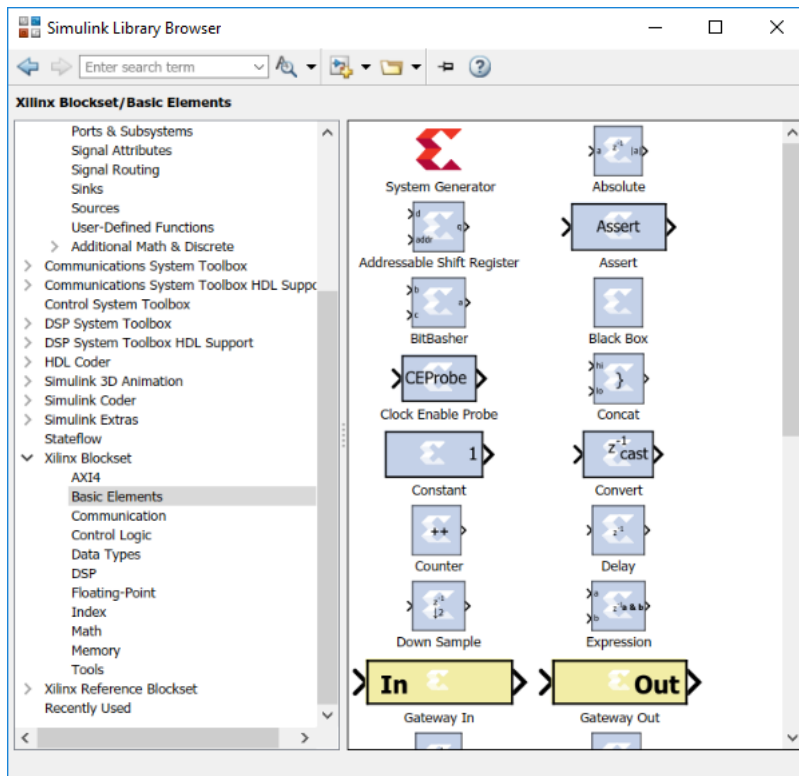
- Select Blank Model on the Simulink startup page.



- Simulink will create a blank model where users can create their designs.



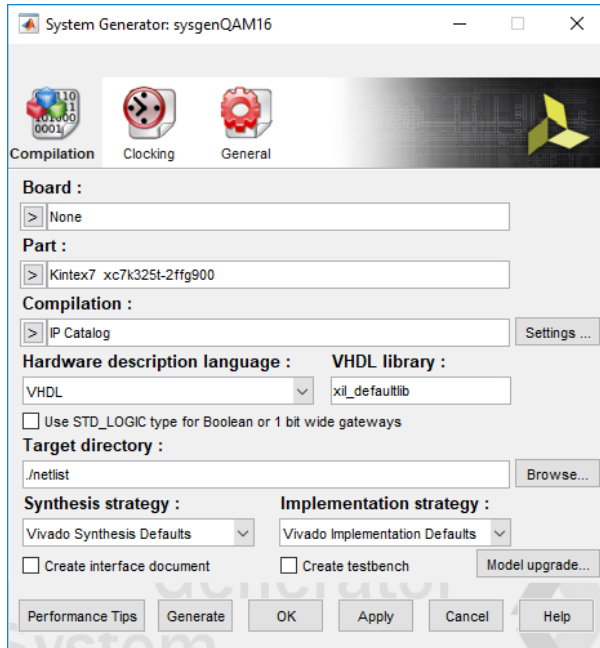
- Save the design by selecting File→Save. Make sure you save the Simulink project to a location that does not have any spaces in the filepath. Vivado will return errors when generating HDL code from the System Generator design if there are spaces in the filepath.
- Click on the Library Browser button  and navigate to Xilinx Blockset→ Basic elements



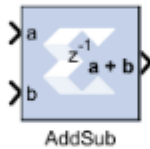
- Drag the System Generator icon on to the design canvas.
- Click on the System Generator icon in the Simulink design:



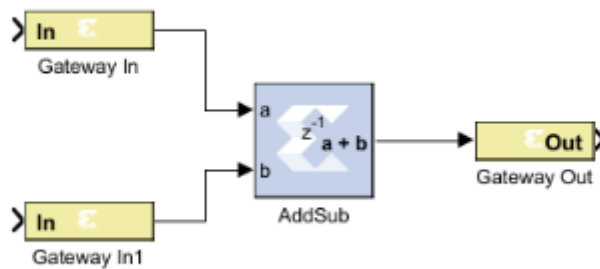
- The System Generator properties dialog will open.



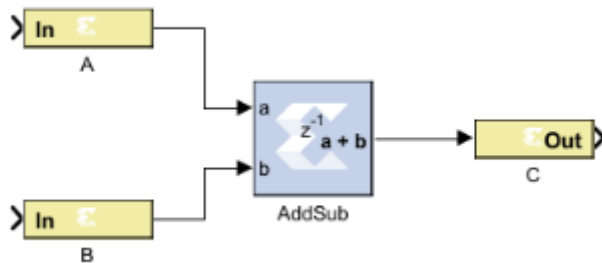
- If you are using an M3XXXA module with the k410 fpga option, set the Part to xc7k410t-2ffg676. If you are using an M3XXXA module with the k325 option, set the Part dialog box to xc7k325t-2ffg676. Set the Compilation dialog box to IP Catalog. The Hardware description language dialog box can be set to either VHDL or Verilog. This example will use VHDL. Click Ok to close the dialog window.
- In the Library browser go to Xilinx Blockset→Math and drag the AddSub block onto the design canvas. The AddSub block defaults to a latency of 1. Leave the default value set at 1.



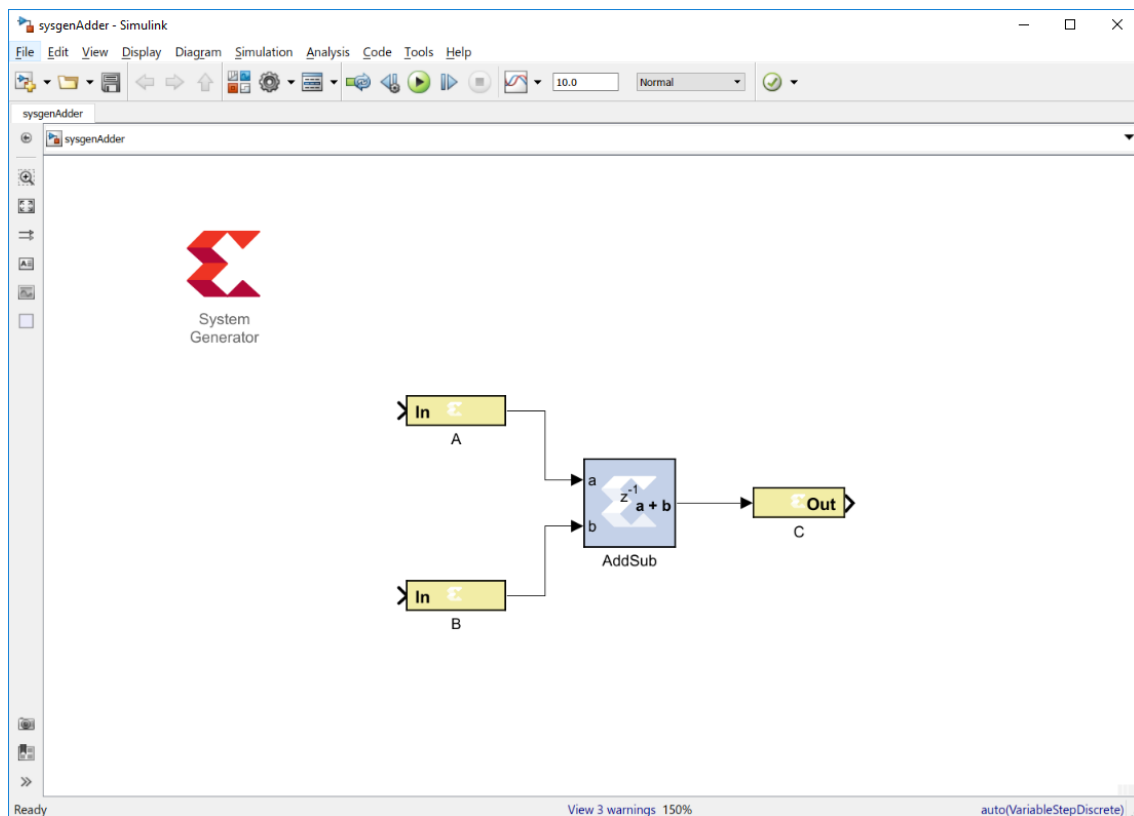
- Then in the Library Browser navigate to Xilinx Blockset→Basic Elements and drag two In ports and one Out port onto the design canvas. Connect the In ports to the a and b inputs of the AddSub block and connect the Out port to the a+b output of the AddSub block. Both the In and Out blocks default to a port width of 16. Leave the default value set at 16.



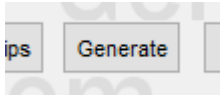
- Change the name on the upper input port to A and the name on the lower input port to B by double clicking on the text below the port. Change the name on the output port to C.



- The final design should look something like this.

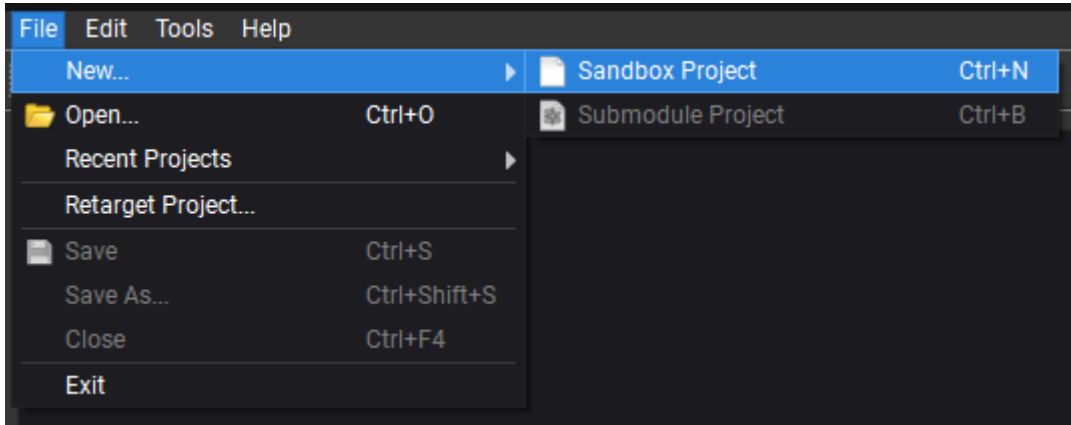


- Click on the System Generator icon and then click Generate in the System Generator dialog window. System Generator will generate the HDL code for the design.

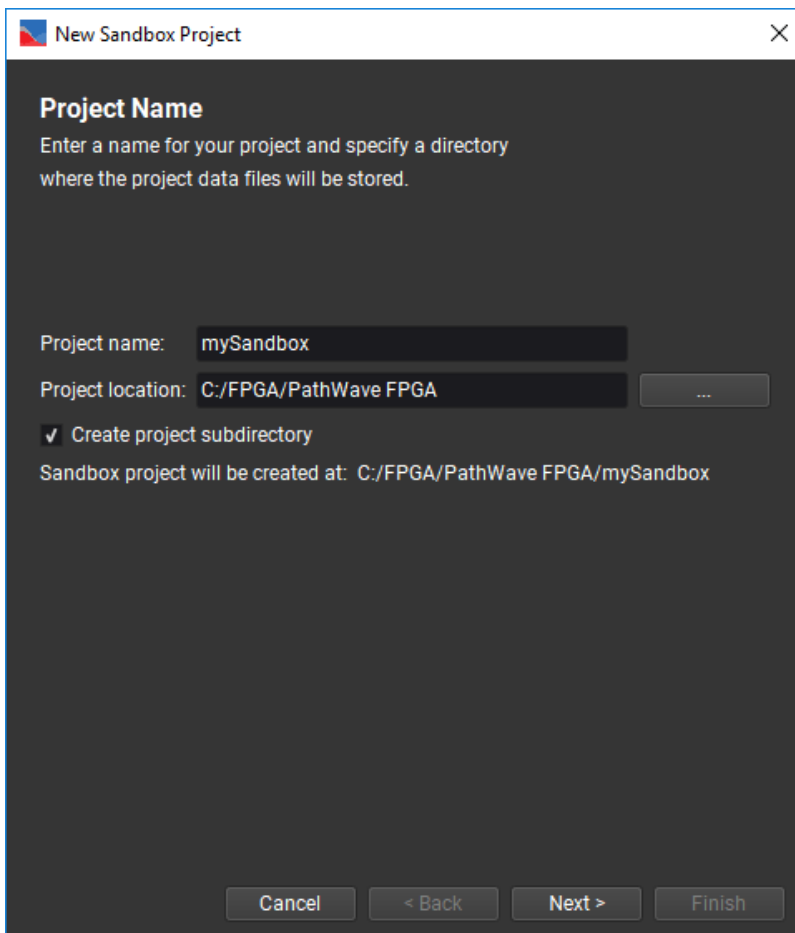


When System Generator finishes generating the HDL code, open PathWave FPGA and create a new project with the correct FPGA option (either k410 or k325).

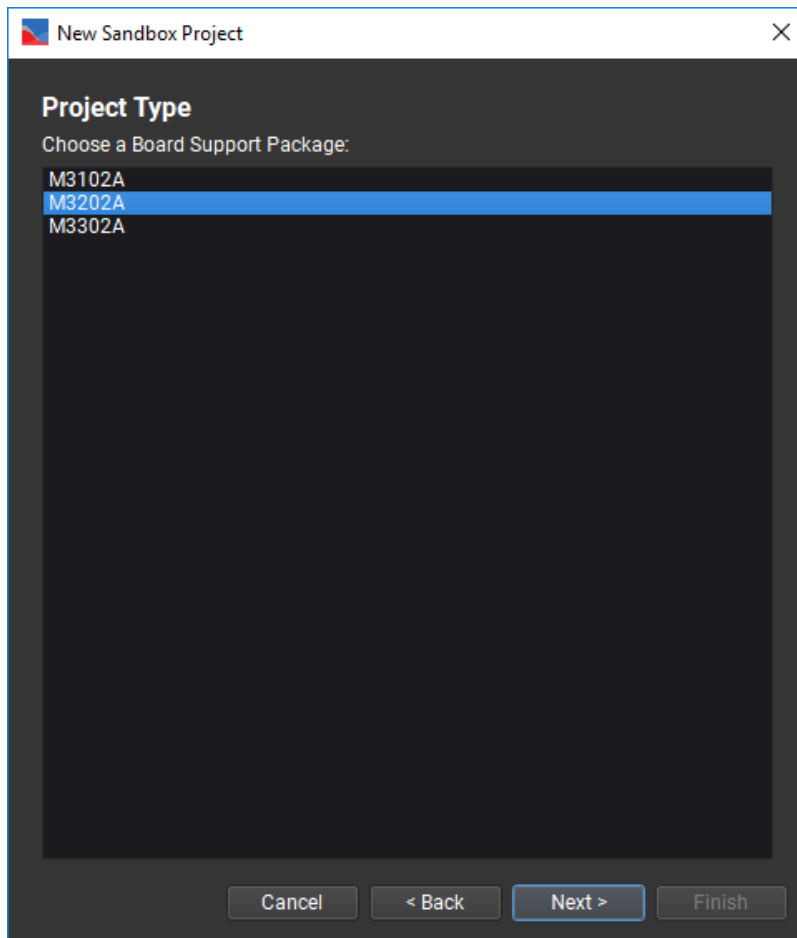
- Launch PathWave FPGA .
- Click on File→New...→Sandbox Project in the PathWave FPGA GUI.



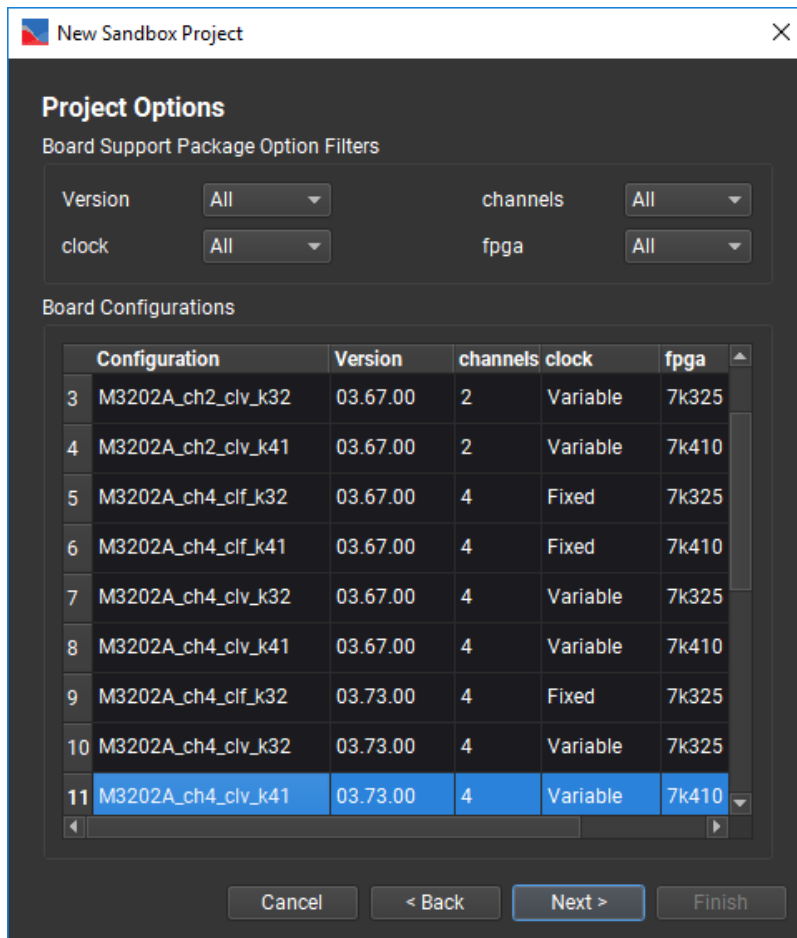
- Choose a Project name and Project location in the New Sandbox Project dialog and click Next.



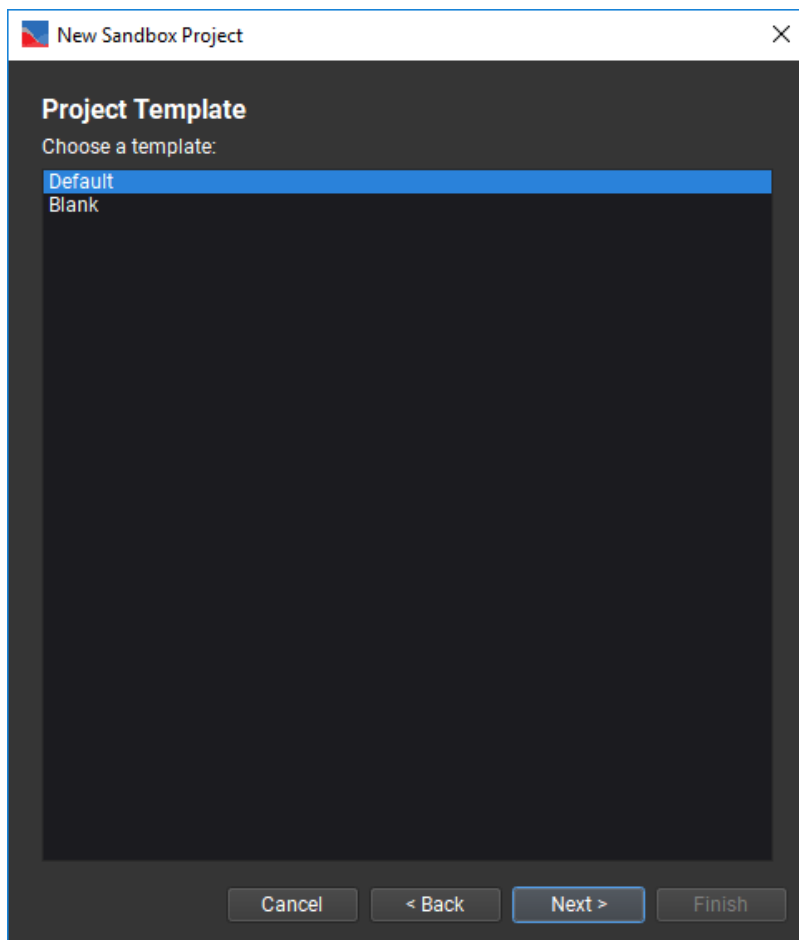
- Select a Board Support Package and click the Next button. This example uses the M3202A.



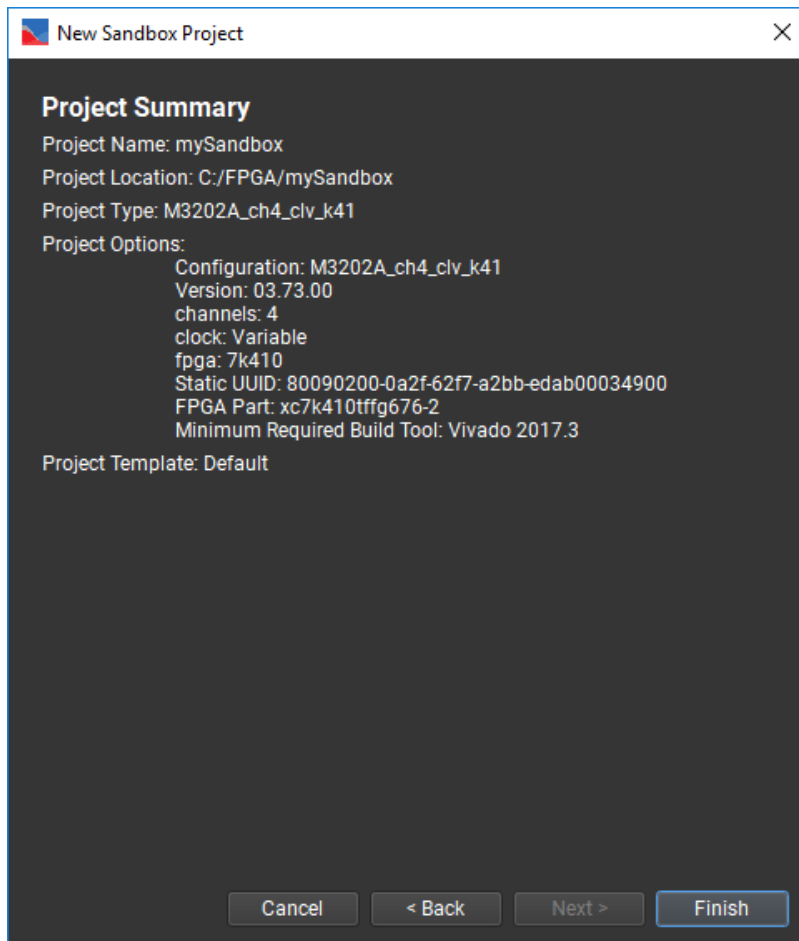
- Select a k325 or k410 configuration depending on your hardware module and click Next.



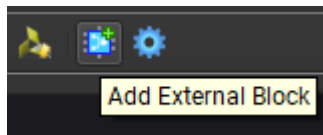
- Choose either the Default or Blank template and click Next. The Default template comes with AWG IP and the Blank template does not instantiate any IP blocks.



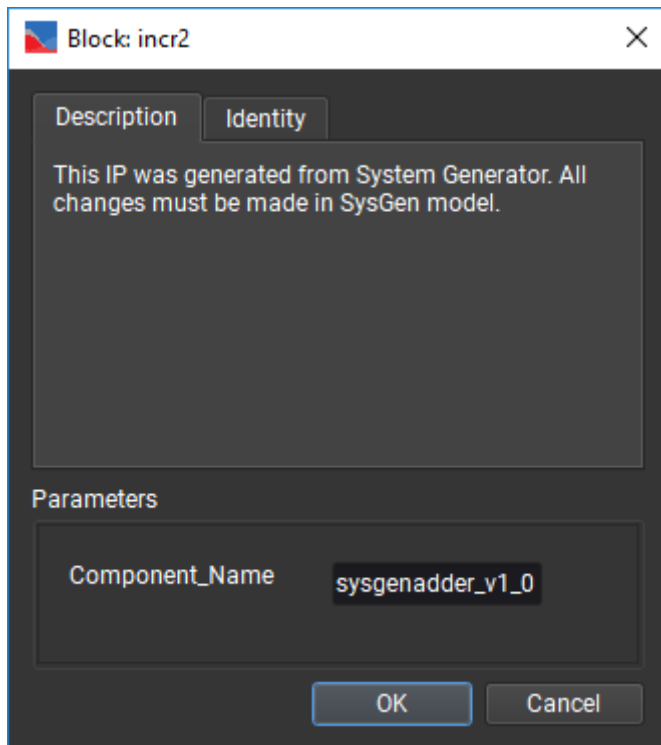
- The final page is the Project Summary page. Click Finish and PathWave FPGA will create the project.



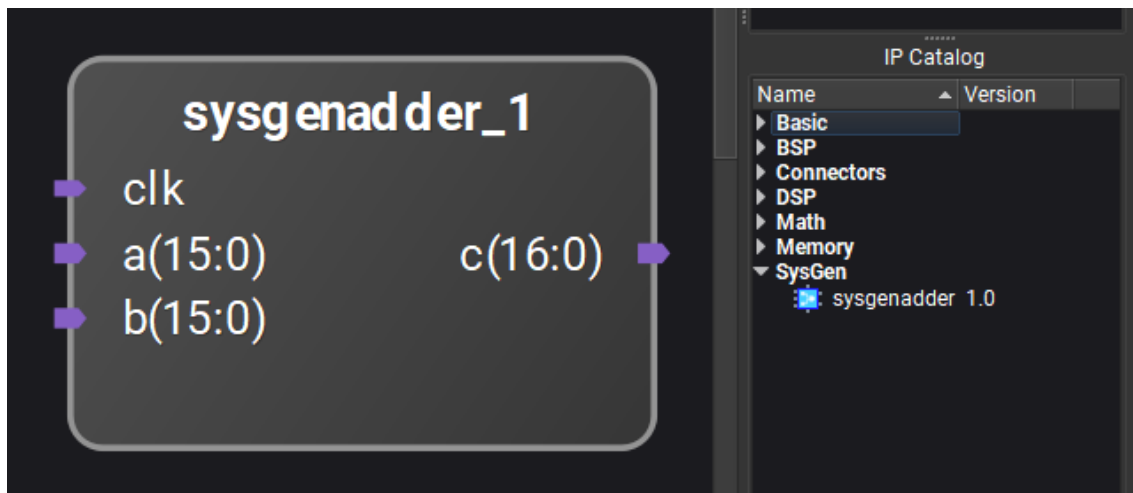
- Then click on the Add External Block button on the toolbar.



- Navigate to the SystemGenerator output directory. Go to netlist/ip and select the component.xml file. Click OK on the dialog box that shows the block description.



- The System Generator IP block will be imported into PathWave FPGA and will show up in the IP Catalog.



Appendix

- [VHDL Support](#)
- [Verilog Support](#)
- [Infer Interface Reference](#)
- [Importing IP with Invalid IP-XACT](#)

Infer Interface Reference

This section details the standard naming conventions used to infer interfaces from physical ports in an HDL file. Physical ports may be named with an arbitrary common prefix, followed by an underscore ("_"), followed by the standard port names for that interface. The physical ports may also be named as the standard port names for that interface, with no prefix. PathWave FPGA ignores the capitalization of the standard port names and prefix. The inferred interface name will usually be the common prefix of the included physical ports. The inference rules generally follow the conventions in the Xilinx document ug1118, for packaging custom IP in Vivado. Clock, reset, AXIMM, AXILite, AXIS, and PathWave FPGA mem interfaces may be inferred.

When inferring interfaces, physical ports with a fixed width of 1 are valid mappings to logical ports of width 1. The following, for example is valid:

```
-- valid when inferring interfaces
port (
    rdata: in STD_LOGIC_VECTOR ( 0 downto 0 );
);
```

Parameterized physical ports with width 1 are *not* valid mappings to logical ports of width 1; this is because the width is not guaranteed to be 1. The following, for example, is not valid when inferring interfaces:

```
-- not valid when inferring interfaces
generic (
    size : integer := 0
);
port (
    rdata: in STD_LOGIC_VECTOR ( size downto 0 );
);
```

The tables below contain the name of the interface port, whether it is required on master, and whether it is required on a slave. To infer an interface, all of the required ports on either the master or slave must be present. The checks for matching are case-insensitive.

CLOCK

Port Name	Required on Master	Required on Slave
clk	required	required

Clock interfaces may be inferred from port names of several patterns. In Xilinx UG1118 nomenclature, clocks may be matched with: `[_]clk`, `[_]clk_in`, `[_]ack`, `[_]ack_in`, or `[_]clock[_]`

NRST

PortName	Required on Master	Required on Slave
nrst	required	required

Reset interfaces may be inferred from port names of several patterns. In Xilinx UG1118 nomenclature, resets may be matched with: [*_]resetn, [*_]aresetn, [*_]rstn, or [*_]nrst. Patterns for positive active resets are not recognized.

AXIMM

Port Name	Required on Master	Required on Slave
araddr	required	required
arburst	optional	required
arcache	optional	optional
arid	optional	optional
arlen	optional	required
arlock	optional	optional
arprot	required	optional
arqos	optional	optional
arready	required	required
arregion	optional	optional
arsize	optional	required
aruser	optional	optional
arvalid	required	required
awaddr	required	required
awburst	optional	required
awcache	optional	optional
awid	optional	optional
awlen	optional	required
awlock	optional	optional
awprot	required	optional
awqos	optional	optional
awready	required	required
awregion	optional	optional
awsize	optional	required
awuser	optional	optional
awvalid	required	required

Port Name	Required on Master	Required on Slave
bid	optional	optional
bready	required	required
bresp	optional	optional
buser	optional	optional
bvalid	required	required
rdata	required	required
rid	optional	optional
rlast	optional	required
rready	required	required
rresp	optional	optional
ruser	optional	optional
rvalid	required	required
wdata	required	required
wlast	optional	optional
wready	required	required
wstrb	optional	required
wuser	optional	optional
wvalid	required	required

AXILite

Port Name	Required on Master	Required on Slave
araddr	required	required
arprot	optional	optional
arready	required	required
arvalid	required	required
awaddr	required	required
awprot	optional	optional
awready	required	required
awvalid	required	required
bready	required	required
bresp	optional	optional
bvalid	required	required
rdata	required	required

Port Name	Required on Master	Required on Slave
rready	required	required
rresp	optional	optional
rvalid	required	required
wdata	required	required
wready	required	required
wstrb	optional	required
wvalid	required	required

AXIS

Port Name	Required on Master	Required on Slave
tdata	optional	optional
tdest	optional	optional
tid	optional	optional
tkeep	optional	optional
tlast	optional	optional
tready	optional	optional
tstrb	optional	optional
tuser	optional	optional
tvalid	required	required

MEM

Port Name	Required on Master	Required on Slave
address	required	required
rddata	required	required
rden	required	required
wdata	required	required
wren	required	required

Importing IP with Invalid IP-XACT

When importing IP, Pathwave FPGA can use IP-XACT files to determine the modules ports and interfaces. However, some third party tools can generate invalid IP-XACT files. These are IP-XACT files that violate the IP-XACT specification. An example of this is if the IP-XACT file uses names that include invalid characters such as embedded spaces. Pathwave FPGA will generate errors when trying to parse these IP-XACT files.

To use one of these IP blocks, the best solution would be to obtain valid IP-XACT files. If this is not possible, the alternative is to create valid IP-XACT using Pathwave FPGA's IP Packager and replacing the invalid IP-XACT with the valid IP-XACT. Creating this IP-XACT will require knowledge of the IP block's ports and interface structure. If the IP block is from Vivado's IP Catalog, then the "Files" section of the IP-XACT should point to the *.xci file generated by Vivado. If the IP block is HDL, then the "Files" section of the IP-XACT should include the HDL as well as any submodules necessary to build it.

Note that Vivado generates the older, incompatible version of IP-XACT, and hence Vivado's IP-XACT can not be edited in the Pathwave FPGA's IP Packager.

VHDL Support

This page describes the supported VHDL types and constructs when importing a VHDL file into PathWave FPGA. These limitations apply to the following flows:

- [IP Packager](#), when using the "Autofill from File" or "Load from File" action.
- [Imported User IP](#)

It is recommended that you create IP-XACT for any VHDL IP that does not meet the conditions described in this section.

Generics

All generics are treated as user-configurable parameters by PathWave FPGA.

The supported datatypes for generics are:

- **bit**
- **boolean**
- **natural** - treated as integer, but with minimum boundary set to 0
- **positive** - treated as integer, but with minimum boundary set to 1
- **integer**
- **string**

The supported operators for the default values of integer type generics are:

- **+** : addition
- **-** : subtraction
- ***** : multiplication
- **/** : division

Ports

All ports are treated as *std_logic* or *std_logic_vector* type by PathWave FPGA. The supported datatypes are:

- **std_logic**
- **std_logic_vector**
- **bit** - treated as *std_logic*
- **bit_vector** - treated as *std_logic_vector*, with the same range
- **boolean** - treated as *std_logic*

- **natural** - treated as *std_logic_vector(30 downto 0)*
- **positive** - treated as *std_logic_vector(30 downto 0)*
- **integer** - treated as *std_logic_vector(31 downto 0)*
- **character** - treated as *std_logic_vector(7 downto 0)*

Port ranges can use generics and the supported operators described above. See Known Issues below for limitations on port boundaries.

Known Issues

- The value range of an Integer datatype of a port is ignored. Directly importing such a file in PathWave FPGA will be completed successfully, however, the synthesis of any design that contains that IP will fail. A workaround is to create an IP-XACT file for the VHDL file using the [IP Packager](#). Then, in the Physical Ports tab, modify the width to match the actual width required.
- Some VHDL errors are ignored by PathWave FPGA when importing VHDL, but will fail during synthesis. Vivado is the authority on whether a VHDL file is valid, not PathWave FPGA.
- For vector ports with a 'downto' range, the right boundary must be literal '0'. For a 'to' range, the left boundary must be literal '0'.
- Constants or datatypes imported from another package cannot be used in the entity declaration.
- When Kactus2 is used for creating IP-XACT for a VHDL file, the VHDL entity declaration must end with "end <entity_name>" and not "end entity."
- Arrays are not supported. They may or may not load into the schematic properly, but they will not build properly.

Verilog Support

This page describes known issues when importing a Verilog file into PathWave FPGA. These limitations apply to the following flows:

- [IP Packager](#), when using the "Autofill from File" or "Load from File" action.
- [Imported User IP](#)

It is recommended that you create IP-XACT for any Verilog IP that does not meet the conditions described in this section.

Parameters

All parameters are treated as user-configurable parameters by PathWave FPGA.

The **parameter** keyword is supported, but the **localparam** keyword is not. Local parameters are permitted, but they cannot be used in a port definition.

Parameters are always treated as 32 bit integers. It is valid to declare an integer type, or give a range declaration, but it will still be treated as a 32 bit signed integer. For example, in "parameter [1:0] myParam = 5", the parameter has the value 5 instead of being truncated to 1.

For a parameter to be automatically inferred from a Verilog source file (e.g. when using Autofill from file in the IP Packager), the parameter should be defined in the module name declaration. For example, "module modname #(parameter a=5, b=6)' would detect the

parameters a and b. If the parameters are defined in the body of the module (after the port definitions), they will not be automatically inferred and should be entered manually.

Expressions

Most Verilog expressions and functions are supported.

All Verilog math functions are supported except **ln**.

The following operators are not supported: bit select (**[]**), concatenation(**{ }**), and replication(**{{ }}**)

Reduction operators are supported, but they are always evaluated from a 64 bit signed integer. For example, "&1'x1" returns 0 because "&'x0000000000000001" is 0.

Known Issues

Importing Verilog IP into PathWave FPGA has a number of known limitations. It is recommended that you create IP-XACT for any Verilog IP that does not meet the following conditions. Note that only module declarations, port and parameter definitions and 'endmodule' are checked. A violation of the following conditions will produce a "Syntax Error" message when importing Verilog IP:

- Module declarations must include at least one port definition.
- Ports and parameters cannot have the same name differing only by case (e.g. "myPort" and "myport").
- Tasks and functions are not supported because their ports are misinterpreted as part of the module's interface.
- Output registers cannot be assigned an initial value in the same statement where it is defined, such as `output reg myReg = 0;`
- Definition of port attributes is not supported, such as `(* attribute definition *) input portName,`
- Parameters and port definitions in a module declaration may not be conditionally included using ``ifdef / `endif` statements and they cannot use any preprocessor variables.
- Expressions are limited to 32-bit signed integers. For example, `'hFFFF_FFFF` is treated as -1 instead of 4294967295.
- Size constants in expressions are ignored. For example, `"4'd65"` is treated as 65 instead of being truncated to 1.
- Arrays will fail to parse and will not load.

Legal

Portions of this software are licensed by third parties including open source terms and conditions.

7-zip

PathWave FPGA uses parts of 7-Zip, which is licensed under the GNU LGPL license. For more information or to receive a copy of the source code for 7-Zip, visit <http://support.keysight.com>.

License for use and distribution

~~~~~

7-Zip Copyright (C) 1999-2016 Igor Pavlov.

Licenses for files are:

- 1) 7z.dll: GNU LGPL + unRAR restriction
- 2) All other files: GNU LGPL

The GNU LGPL + unRAR restriction means that you must follow both GNU LGPL rules and unRAR restriction rules.

#### Note:

You can use 7-Zip on any computer, including a computer in a commercial organization. You don't need to register or pay for 7-Zip.

#### GNU LGPL information

-----

This library is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation; either version 2.1 of the License, or (at your option) any later version.

This library is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You can receive a copy of the GNU Lesser General Public License from <http://www.gnu.org/>

#### unRAR restriction

-----

The decompression engine for RAR archives was developed using source code of unRAR program.

All copyrights to original unRAR code are owned by Alexander Roshal.

The license for original unRAR code has the following restriction:

The unRAR sources cannot be used to re-create the RAR compression algorithm, which is proprietary. Distribution of modified unRAR sources in separate form or as a part of other software is permitted, provided that it is clearly stated in the documentation and source comments that the code may not be used to develop a RAR (WinRAR) compatible archiver.

--

Igor Pavlov

## bzip2

PathWave FPGA uses bzip2 v1.0.6, used with permission. For more information, visit <https://spdx.org/licenses/bzip2-1.0.6.html>.

## Doxygen

PathWave FPGA uses Doxygen 1.8.13, which is licensed under the GNU LGPL license. For more information or to receive a copy of the source code for Doxygen, visit <http://support.keysight.com>

### Doxygen license

Copyright © 1997-2018 by Dimitri van Heesch.

Permission to use, copy, modify, and distribute this software and its documentation under the terms of the GNU General Public License is hereby granted. No representations are made about the suitability of this software for any purpose. It is provided "as is" without express or implied warranty. See the GNU General Public License for more details.

Documents produced by doxygen are derivative works derived from the input used in their production; they are not affected by this license.

## Inja

PathWave FPGA uses Inja. For more information, visit <https://github.com/pantor/inja>

### MIT License

Copyright (c) 2018 Ibersch

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

---

Copyright (c) 2009-2018 FIRST  
All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

\* Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

\* Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

\* Neither the name of the FIRST nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY FIRST AND CONTRIBUTORS ``AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL FIRST OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

## Lua

PathWave FPGA uses parts of Lua 5.3.4.

Copyright © 1994–2017 [Lua.org](http://lua.org), PUC-Rio.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NON-INFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

## Qt

PathWave FPGA uses Qt 5.7.0 and 5.6.2, licensed under the terms of GNU LGPLv3. For more information or to receive a copy of the source code for Qt, visit <http://support.keysight.com>.

The Qt Toolkit is Copyright (C) 2015 The Qt Company Ltd.

Contact: <http://www.qt.io/licensing/>

You may use, distribute and copy the Qt GUI Toolkit under the terms of GNU Lesser General Public License version 3, which is displayed below. This license makes reference to the version 3 of the GNU General Public License, which you can find below.

## Xerces-C++

PathWave FPGA uses Xerces-C++ 3.2.0, licensed under the terms of Apache License v2.0, which is displayed below. For more information, visit <https://xerces.apache.org/xerces-c/>.

```
=====
=
== NOTICE file corresponding to section 4(d) of the Apache License, ==
== Version 2.0, in this case for the Apache Xerces distribution. ==
=====
=

This product includes software developed by
The Apache Software Foundation (http://www.apache.org/).

Portions of this software were originally based on the following:
- software copyright (c) 1999, IBM Corporation., http://www.ibm.com.
```

## zlib

PathWave FPGA uses zlib 1.2.11, used by permission. For more information, visit [https://www.zlib.net/zlib\\_license.html](https://www.zlib.net/zlib_license.html).

## Apache License v2.0

### Apache License

Version 2.0, January 2004

<http://www.apache.org/licenses/>

#### TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

##### 1. Definitions.

"License" shall mean the terms and conditions for use, reproduction, and distribution as defined by Sections 1 through 9 of this document.

"Licensor" shall mean the copyright owner or entity authorized by the copyright owner that is granting the License.

"Legal Entity" shall mean the union of the acting entity and all other entities that control, are controlled by, or are under common control with that entity. For the purposes of this definition, "control" means (i) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (ii) ownership of fifty percent (50%) or more of the outstanding shares, or (iii) beneficial ownership of such entity.

"You" (or "Your") shall mean an individual or Legal Entity exercising permissions granted by this License.

"Source" form shall mean the preferred form for making modifications, including but not limited to software source code, documentation source, and configuration files.

"Object" form shall mean any form resulting from mechanical transformation or translation of a Source form, including but not limited to compiled object code, generated documentation, and conversions to other media types.

"Work" shall mean the work of authorship, whether in Source or Object form, made available under the License, as indicated by a copyright notice that is included in or attached to the work (an example is provided in the Appendix below).

"Derivative Works" shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

"Contribution" shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, "submitted" means any form of electronic, verbal, or written communication sent to the Licensor or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as "Not a Contribution."

"Contributor" shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:

1. You must give any other recipients of the Work or Derivative Works a copy of this License; and
2. You must cause any modified files to carry prominent notices stating that You changed the files; and
3. You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and
4. If the Work includes a "NOTICE" text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution

notices cannot be construed as modifying the License.

You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise, any Contribution intentionally submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions. Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement you may have executed with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be liable to You for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing the Work or Derivative Works thereof, You may choose to offer, and charge a fee for, acceptance of support, warranty, indemnity, or other liability obligations and/or rights consistent with this License. However, in accepting such obligations, You may act only on Your own behalf and on Your sole responsibility, not on behalf of any other Contributor, and only if You agree to indemnify, defend, and hold each Contributor harmless for any liability incurred by, or claims asserted against, such Contributor by reason of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS

## APPENDIX: HOW TO APPLY THE APACHE LICENSE TO YOUR WORK

To apply the Apache License to your work, attach the following boilerplate notice, with the fields enclosed by brackets "[]" replaced with your own identifying information. (Don't include the brackets!) The text should be enclosed in the appropriate comment syntax for the file format. We also recommend that a file or class name and description of purpose be included on the same "printed page" as the copyright notice for easier identification within third-party archives.

```
Copyright [yyyy] [name of copyright owner]
```

```
Licensed under the Apache License, Version 2.0 (the "License");  
you may not use this file except in compliance with the License.  
You may obtain a copy of the License at
```

```
http://www.apache.org/licenses/LICENSE-2.0
```

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.

See the License for the specific language governing permissions and limitations under the License.

## GNU GPLv3

### GNU GENERAL PUBLIC LICENSE

Version 3, 29 June 2007

Copyright © 2007 Free Software Foundation, Inc. <<https://fsf.org/>>

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

#### Preamble

The GNU General Public License is a free, copyleft license for software and other kinds of works.

The licenses for most software and other practical works are designed to take away your freedom to share and change the works. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change all versions of a program--to make sure it remains free software for all its users. We, the Free Software Foundation, use the GNU General Public License for most of our software; it applies also to any other work released this way by its authors. You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for them if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs, and that you know you can do these things.

To protect your rights, we need to prevent others from denying you these rights or asking you to surrender the rights. Therefore, you have certain responsibilities if you distribute copies of the software, or if you modify it: responsibilities to respect the freedom of others.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must pass on to the recipients the same freedoms that you received. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

Developers that use the GNU GPL protect your rights with two steps: (1) assert copyright on the software, and (2) offer you this License giving you legal permission to copy, distribute and/or modify it.

For the developers' and authors' protection, the GPL clearly explains that there is no warranty for this free software. For both users' and authors' sake, the GPL requires that modified versions be marked as changed, so that their problems will not be attributed erroneously to authors of previous versions.

Some devices are designed to deny users access to install or run modified versions of the software inside them, although the manufacturer can do so. This is fundamentally incompatible with the aim of protecting users' freedom to change the software. The systematic pattern of such abuse occurs in the area of products for individuals to use, which is precisely where it is most unacceptable. Therefore, we have designed this version of the GPL to prohibit the practice for those products. If such problems arise substantially in other domains, we stand ready to extend this provision to those domains in future versions of the GPL, as needed to protect the freedom of users.



Finally, every program is threatened constantly by software patents. States should not allow patents to restrict development and use of software on general-purpose computers, but in those that do, we wish to avoid the special danger that patents applied to a free program could make it effectively proprietary. To prevent this, the GPL assures that patents cannot be used to render the program non-free.

The precise terms and conditions for copying, distribution and modification follow.

## TERMS AND CONDITIONS

### ***0. Definitions.***

“This License” refers to version 3 of the GNU General Public License.

“Copyright” also means copyright-like laws that apply to other kinds of works, such as semiconductor masks.

“The Program” refers to any copyrightable work licensed under this License. Each licensee is addressed as “you”. “Licensees” and “recipients” may be individuals or organizations.

To “modify” a work means to copy from or adapt all or part of the work in a fashion requiring copyright permission, other than the making of an exact copy. The resulting work is called a “modified version” of the earlier work or a work “based on” the earlier work.

A “covered work” means either the unmodified Program or a work based on the Program.

To “propagate” a work means to do anything with it that, without permission, would make you directly or secondarily liable for infringement under applicable copyright law, except executing it on a computer or modifying a private copy. Propagation includes copying, distribution (with or without modification), making available to the public, and in some countries other activities as well.

To “convey” a work means any kind of propagation that enables other parties to make or receive copies. Mere interaction with a user through a computer network, with no transfer of a copy, is not conveying.

An interactive user interface displays “Appropriate Legal Notices” to the extent that it includes a convenient and prominently visible feature that (1) displays an appropriate copyright notice, and (2) tells the user that there is no warranty for the work (except to the extent that warranties are provided), that licensees may convey the work under this License, and how to view a copy of this License. If the interface presents a list of user commands or options, such as a menu, a prominent item in the list meets this criterion.

### ***1. Source Code.***

The “source code” for a work means the preferred form of the work for making modifications to it. “Object code” means any non-source form of a work.

A “Standard Interface” means an interface that either is an official standard defined by a recognized standards body, or, in the case of interfaces specified for a particular programming language, one that is widely used among developers working in that language.

The “System Libraries” of an executable work include anything, other than the work as a whole, that (a) is included in the normal form of packaging a Major Component, but which is not part of that Major Component, and (b) serves only to enable use of the work with that Major Component, or to implement a Standard Interface for which an implementation is available to the public in source code form. A “Major Component”, in this context, means a major essential component (kernel, window system, and so on) of the specific operating system (if any) on which the executable work runs, or a compiler used to produce the work, or an object code interpreter used to run it.

The “Corresponding Source” for a work in object code form means all the source code needed to generate, install, and (for an executable work) run the object code and to modify the work, including scripts to control those activities. However, it does not include the work’s System Libraries, or general-purpose tools or generally available free programs which are used unmodified in performing those activities but which are not part of the work. For example,

Corresponding Source includes interface definition files associated with source files for the work, and the source code for shared libraries and dynamically linked subprograms that the work is specifically designed to require, such as by intimate data communication or control flow between those subprograms and other parts of the work.

The Corresponding Source need not include anything that users can regenerate automatically from other parts of the Corresponding Source.

The Corresponding Source for a work in source code form is that same work.

## ***2. Basic Permissions.***

All rights granted under this License are granted for the term of copyright on the Program, and are irrevocable provided the stated conditions are met. This License explicitly affirms your unlimited permission to run the unmodified Program. The output from running a covered work is covered by this License only if the output, given its content, constitutes a covered work. This License acknowledges your rights of fair use or other equivalent, as provided by copyright law.

You may make, run and propagate covered works that you do not convey, without conditions so long as your license otherwise remains in force. You may convey covered works to others for the sole purpose of having them make modifications exclusively for you, or provide you with facilities for running those works, provided that you comply with the terms of this License in conveying all material for which you do not control copyright. Those thus making or running the covered works for you must do so exclusively on your behalf, under your direction and control, on terms that prohibit them from making any copies of your copyrighted material outside their relationship with you.

Conveying under any other circumstances is permitted solely under the conditions stated below. Sublicensing is not allowed; section 10 makes it unnecessary.

## ***3. Protecting Users' Legal Rights From Anti-Circumvention Law.***

No covered work shall be deemed part of an effective technological measure under any applicable law fulfilling obligations under article 11 of the WIPO copyright treaty adopted on 20 December 1996, or similar laws prohibiting or restricting circumvention of such measures.

When you convey a covered work, you waive any legal power to forbid circumvention of technological measures to the extent such circumvention is effected by exercising rights under this License with respect to the covered work, and you disclaim any intention to limit operation or modification of the work as a means of enforcing, against the work's users, your or third parties' legal rights to forbid circumvention of technological measures.

## ***4. Conveying Verbatim Copies.***

You may convey verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice; keep intact all notices stating that this License and any non-permissive terms added in accord with section 7 apply to the code; keep intact all notices of the absence of any warranty; and give all recipients a copy of this License along with the Program.

You may charge any price or no price for each copy that you convey, and you may offer support or warranty protection for a fee.

## ***5. Conveying Modified Source Versions.***

You may convey a work based on the Program, or the modifications to produce it from the Program, in the form of source code under the terms of section 4, provided that you also meet all of these conditions:

- a) The work must carry prominent notices stating that you modified it, and giving a relevant date.
- b) The work must carry prominent notices stating that it is released under this License and any conditions added under section 7. This requirement modifies the requirement in section 4 to "keep intact all notices".

- c) You must license the entire work, as a whole, under this License to anyone who comes into possession of a copy. This License will therefore apply, along with any applicable section 7 additional terms, to the whole of the work, and all its parts, regardless of how they are packaged. This License gives no permission to license the work in any other way, but it does not invalidate such permission if you have separately received it.
- d) If the work has interactive user interfaces, each must display Appropriate Legal Notices; however, if the Program has interactive interfaces that do not display Appropriate Legal Notices, your work need not make them do so.

A compilation of a covered work with other separate and independent works, which are not by their nature extensions of the covered work, and which are not combined with it such as to form a larger program, in or on a volume of a storage or distribution medium, is called an “aggregate” if the compilation and its resulting copyright are not used to limit the access or legal rights of the compilation's users beyond what the individual works permit. Inclusion of a covered work in an aggregate does not cause this License to apply to the other parts of the aggregate.

## **6. Conveying Non-Source Forms.**

You may convey a covered work in object code form under the terms of sections 4 and 5, provided that you also convey the machine-readable Corresponding Source under the terms of this License, in one of these ways:

- a) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by the Corresponding Source fixed on a durable physical medium customarily used for software interchange.
- b) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by a written offer, valid for at least three years and valid for as long as you offer spare parts or customer support for that product model, to give anyone who possesses the object code either (1) a copy of the Corresponding Source for all the software in the product that is covered by this License, on a durable physical medium customarily used for software interchange, for a price no more than your reasonable cost of physically performing this conveying of source, or (2) access to copy the Corresponding Source from a network server at no charge.
- c) Convey individual copies of the object code with a copy of the written offer to provide the Corresponding Source. This alternative is allowed only occasionally and noncommercially, and only if you received the object code with such an offer, in accord with subsection 6b.
- d) Convey the object code by offering access from a designated place (gratis or for a charge), and offer equivalent access to the Corresponding Source in the same way through the same place at no further charge. You need not require recipients to copy the Corresponding Source along with the object code. If the place to copy the object code is a network server, the Corresponding Source may be on a different server (operated by you or a third party) that supports equivalent copying facilities, provided you maintain clear directions next to the object code saying where to find the Corresponding Source. Regardless of what server hosts the Corresponding Source, you remain obligated to ensure that it is available for as long as needed to satisfy these requirements.
- e) Convey the object code using peer-to-peer transmission, provided you inform other peers where the object code and Corresponding Source of the work are being offered to the general public at no charge under subsection 6d.

A separable portion of the object code, whose source code is excluded from the Corresponding Source as a System Library, need not be included in conveying the object code work.

A “User Product” is either (1) a “consumer product”, which means any tangible personal property which is normally used for personal, family, or household purposes, or (2) anything designed or sold for incorporation into a dwelling. In determining whether a product is a consumer product, doubtful cases shall be resolved in favor of coverage. For a particular product received by a particular user, “normally used” refers to a typical or common use of that class of product, regardless of the status of the particular user or of the way in which the particular user actually uses, or expects or is expected to use, the product. A product is a

consumer product regardless of whether the product has substantial commercial, industrial or non-consumer uses, unless such uses represent the only significant mode of use of the product.

“Installation Information” for a User Product means any methods, procedures, authorization keys, or other information required to install and execute modified versions of a covered work in that User Product from a modified version of its Corresponding Source. The information must suffice to ensure that the continued functioning of the modified object code is in no case prevented or interfered with solely because modification has been made.

If you convey an object code work under this section in, or with, or specifically for use in, a User Product, and the conveying occurs as part of a transaction in which the right of possession and use of the User Product is transferred to the recipient in perpetuity or for a fixed term (regardless of how the transaction is characterized), the Corresponding Source conveyed under this section must be accompanied by the Installation Information. But this requirement does not apply if neither you nor any third party retains the ability to install modified object code on the User Product (for example, the work has been installed in ROM).

The requirement to provide Installation Information does not include a requirement to continue to provide support service, warranty, or updates for a work that has been modified or installed by the recipient, or for the User Product in which it has been modified or installed. Access to a network may be denied when the modification itself materially and adversely affects the operation of the network or violates the rules and protocols for communication across the network.

Corresponding Source conveyed, and Installation Information provided, in accord with this section must be in a format that is publicly documented (and with an implementation available to the public in source code form), and must require no special password or key for unpacking, reading or copying.

## **7. Additional Terms.**

“Additional permissions” are terms that supplement the terms of this License by making exceptions from one or more of its conditions. Additional permissions that are applicable to the entire Program shall be treated as though they were included in this License, to the extent that they are valid under applicable law. If additional permissions apply only to part of the Program, that part may be used separately under those permissions, but the entire Program remains governed by this License without regard to the additional permissions.

When you convey a copy of a covered work, you may at your option remove any additional permissions from that copy, or from any part of it. (Additional permissions may be written to require their own removal in certain cases when you modify the work.) You may place additional permissions on material, added by you to a covered work, for which you have or can give appropriate copyright permission.

Notwithstanding any other provision of this License, for material you add to a covered work, you may (if authorized by the copyright holders of that material) supplement the terms of this License with terms:

- a) Disclaiming warranty or limiting liability differently from the terms of sections 15 and 16 of this License; or
- b) Requiring preservation of specified reasonable legal notices or author attributions in that material or in the Appropriate Legal Notices displayed by works containing it; or
- c) Prohibiting misrepresentation of the origin of that material, or requiring that modified versions of such material be marked in reasonable ways as different from the original version; or
- d) Limiting the use for publicity purposes of names of licensors or authors of the material; or
- e) Declining to grant rights under trademark law for use of some trade names, trademarks, or service marks; or
- f) Requiring indemnification of licensors and authors of that material by anyone who conveys the material (or modified versions of it) with contractual assumptions of liability to

the recipient, for any liability that these contractual assumptions directly impose on those licensors and authors.

All other non-permissive additional terms are considered “further restrictions” within the meaning of section 10. If the Program as you received it, or any part of it, contains a notice stating that it is governed by this License along with a term that is a further restriction, you may remove that term. If a license document contains a further restriction but permits relicensing or conveying under this License, you may add to a covered work material governed by the terms of that license document, provided that the further restriction does not survive such relicensing or conveying.

If you add terms to a covered work in accord with this section, you must place, in the relevant source files, a statement of the additional terms that apply to those files, or a notice indicating where to find the applicable terms.

Additional terms, permissive or non-permissive, may be stated in the form of a separately written license, or stated as exceptions; the above requirements apply either way.

## **8. Termination.**

You may not propagate or modify a covered work except as expressly provided under this License. Any attempt otherwise to propagate or modify it is void, and will automatically terminate your rights under this License (including any patent licenses granted under the third paragraph of section 11).

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, you do not qualify to receive new licenses for the same material under section 10.

## **9. Acceptance Not Required for Having Copies.**

You are not required to accept this License in order to receive or run a copy of the Program. Ancillary propagation of a covered work occurring solely as a consequence of using peer-to-peer transmission to receive a copy likewise does not require acceptance. However, nothing other than this License grants you permission to propagate or modify any covered work. These actions infringe copyright if you do not accept this License. Therefore, by modifying or propagating a covered work, you indicate your acceptance of this License to do so.

## **10. Automatic Licensing of Downstream Recipients.**

Each time you convey a covered work, the recipient automatically receives a license from the original licensors, to run, modify and propagate that work, subject to this License. You are not responsible for enforcing compliance by third parties with this License.

An “entity transaction” is a transaction transferring control of an organization, or substantially all assets of one, or subdividing an organization, or merging organizations. If propagation of a covered work results from an entity transaction, each party to that transaction who receives a copy of the work also receives whatever licenses to the work the party's predecessor in interest had or could give under the previous paragraph, plus a right to possession of the Corresponding Source of the work from the predecessor in interest, if the predecessor has it or can get it with reasonable efforts.

You may not impose any further restrictions on the exercise of the rights granted or affirmed under this License. For example, you may not impose a license fee, royalty, or other charge for exercise of rights granted under this License, and you may not initiate litigation (including a

cross-claim or counterclaim in a lawsuit) alleging that any patent claim is infringed by making, using, selling, offering for sale, or importing the Program or any portion of it.

## **11. Patents.**

A “contributor” is a copyright holder who authorizes use under this License of the Program or a work on which the Program is based. The work thus licensed is called the contributor's “contributor version”.

A contributor's “essential patent claims” are all patent claims owned or controlled by the contributor, whether already acquired or hereafter acquired, that would be infringed by some manner, permitted by this License, of making, using, or selling its contributor version, but do not include claims that would be infringed only as a consequence of further modification of the contributor version. For purposes of this definition, “control” includes the right to grant patent sublicenses in a manner consistent with the requirements of this License.

Each contributor grants you a non-exclusive, worldwide, royalty-free patent license under the contributor's essential patent claims, to make, use, sell, offer for sale, import and otherwise run, modify and propagate the contents of its contributor version.

In the following three paragraphs, a “patent license” is any express agreement or commitment, however denominated, not to enforce a patent (such as an express permission to practice a patent or covenant not to sue for patent infringement). To “grant” such a patent license to a party means to make such an agreement or commitment not to enforce a patent against the party.

If you convey a covered work, knowingly relying on a patent license, and the Corresponding Source of the work is not available for anyone to copy, free of charge and under the terms of this License, through a publicly available network server or other readily accessible means, then you must either (1) cause the Corresponding Source to be so available, or (2) arrange to deprive yourself of the benefit of the patent license for this particular work, or (3) arrange, in a manner consistent with the requirements of this License, to extend the patent license to downstream recipients. “Knowingly relying” means you have actual knowledge that, but for the patent license, your conveying the covered work in a country, or your recipient's use of the covered work in a country, would infringe one or more identifiable patents in that country that you have reason to believe are valid.

If, pursuant to or in connection with a single transaction or arrangement, you convey, or propagate by procuring conveyance of, a covered work, and grant a patent license to some of the parties receiving the covered work authorizing them to use, propagate, modify or convey a specific copy of the covered work, then the patent license you grant is automatically extended to all recipients of the covered work and works based on it.

A patent license is “discriminatory” if it does not include within the scope of its coverage, prohibits the exercise of, or is conditioned on the non-exercise of one or more of the rights that are specifically granted under this License. You may not convey a covered work if you are a party to an arrangement with a third party that is in the business of distributing software, under which you make payment to the third party based on the extent of your activity of conveying the work, and under which the third party grants, to any of the parties who would receive the covered work from you, a discriminatory patent license (a) in connection with copies of the covered work conveyed by you (or copies made from those copies), or (b) primarily for and in connection with specific products or compilations that contain the covered work, unless you entered into that arrangement, or that patent license was granted, prior to 28 March 2007.

Nothing in this License shall be construed as excluding or limiting any implied license or other defenses to infringement that may otherwise be available to you under applicable patent law.

## **12. No Surrender of Others' Freedom.**

If conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot convey a covered work so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not convey it at all. For example, if you agree to terms that obligate you to collect a royalty for

further conveying from those to whom you convey the Program, the only way you could satisfy both those terms and this License would be to refrain entirely from conveying the Program.

### **13. Use with the GNU Affero General Public License.**

Notwithstanding any other provision of this License, you have permission to link or combine any covered work with a work licensed under version 3 of the GNU Affero General Public License into a single combined work, and to convey the resulting work. The terms of this License will continue to apply to the part which is the covered work, but the special requirements of the GNU Affero General Public License, section 13, concerning interaction through a network will apply to the combination as such.

### **14. Revised Versions of this License.**

The Free Software Foundation may publish revised and/or new versions of the GNU General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies that a certain numbered version of the GNU General Public License “or any later version” applies to it, you have the option of following the terms and conditions either of that numbered version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of the GNU General Public License, you may choose any version ever published by the Free Software Foundation.

If the Program specifies that a proxy can decide which future versions of the GNU General Public License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Program.

Later license versions may give you additional or different permissions. However, no additional obligations are imposed on any author or copyright holder as a result of your choosing to follow a later version.

### **15. Disclaimer of Warranty.**

THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

### **16. Limitation of Liability.**

IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

### **17. Interpretation of Sections 15 and 16.**

If the disclaimer of warranty and limitation of liability provided above cannot be given local legal effect according to their terms, reviewing courts shall apply local law that most closely approximates an absolute waiver of all civil liability in connection with the Program, unless a warranty or assumption of liability accompanies a copy of the Program in return for a fee.

END OF TERMS AND CONDITIONS

## How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively state the exclusion of warranty; and each file should have at least the “copyright” line and a pointer to where the full notice is found.

```
<one line to give the program's name and a brief idea of what it
does.>
Copyright (C) <year> <name of author>

This program is free software: you can redistribute it and/or
modify
it under the terms of the GNU General Public License as published
by
the Free Software Foundation, either version 3 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program. If not, see
<https://www.gnu.org/licenses/>.
```

Also add information on how to contact you by electronic and paper mail.

If the program does terminal interaction, make it output a short notice like this when it starts in an interactive mode:

```
<program> Copyright (C) <year> <name of author>
This program comes with ABSOLUTELY NO WARRANTY; for details type
`show w'.
This is free software, and you are welcome to redistribute it
under certain conditions; type `show c' for details.
```

The hypothetical commands `show w' and `show c' should show the appropriate parts of the General Public License. Of course, your program's commands might be different; for a GUI interface, you would use an “about box”.

You should also get your employer (if you work as a programmer) or school, if any, to sign a “copyright disclaimer” for the program, if necessary. For more information on this, and how to apply and follow the GNU GPL, see [<https://www.gnu.org/licenses/>](https://www.gnu.org/licenses/).

The GNU General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Lesser General Public License instead of this License. But first, please read [<https://www.gnu.org/licenses/why-not-lgpl.html>](https://www.gnu.org/licenses/why-not-lgpl.html).

## GNU LESSER GENERAL PUBLIC LICENSE

Version 3, 29 June 2007

Copyright © 2007 Free Software Foundation, Inc. [<https://fsf.org/>](https://fsf.org/)

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.



This version of the GNU Lesser General Public License incorporates the terms and conditions of version 3 of the GNU General Public License, supplemented by the additional permissions listed below.

## 0. Additional Definitions.

As used herein, “this License” refers to version 3 of the GNU Lesser General Public License, and the “GNU GPL” refers to version 3 of the GNU General Public License.

“The Library” refers to a covered work governed by this License, other than an Application or a Combined Work as defined below.

An “Application” is any work that makes use of an interface provided by the Library, but which is not otherwise based on the Library. Defining a subclass of a class defined by the Library is deemed a mode of using an interface provided by the Library.

A “Combined Work” is a work produced by combining or linking an Application with the Library. The particular version of the Library with which the Combined Work was made is also called the “Linked Version”.

The “Minimal Corresponding Source” for a Combined Work means the Corresponding Source for the Combined Work, excluding any source code for portions of the Combined Work that, considered in isolation, are based on the Application, and not on the Linked Version.

The “Corresponding Application Code” for a Combined Work means the object code and/or source code for the Application, including any data and utility programs needed for reproducing the Combined Work from the Application, but excluding the System Libraries of the Combined Work.

## 1. Exception to Section 3 of the GNU GPL.

You may convey a covered work under sections 3 and 4 of this License without being bound by section 3 of the GNU GPL.

## 2. Conveying Modified Versions.

If you modify a copy of the Library, and, in your modifications, a facility refers to a function or data to be supplied by an Application that uses the facility (other than as an argument passed when the facility is invoked), then you may convey a copy of the modified version:

- a) under this License, provided that you make a good faith effort to ensure that, in the event an Application does not supply the function or data, the facility still operates, and performs whatever part of its purpose remains meaningful, or
- b) under the GNU GPL, with none of the additional permissions of this License applicable to that copy.

## 3. Object Code Incorporating Material from Library Header Files.

The object code form of an Application may incorporate material from a header file that is part of the Library. You may convey such object code under terms of your choice, provided that, if the incorporated material is not limited to numerical parameters, data structure layouts and accessors, or small macros, inline functions and templates (ten or fewer lines in length), you do both of the following:

- a) Give prominent notice with each copy of the object code that the Library is used in it and that the Library and its use are covered by this License.
- b) Accompany the object code with a copy of the GNU GPL and this license document.

## 4. Combined Works.

You may convey a Combined Work under terms of your choice that, taken together, effectively do not restrict modification of the portions of the Library contained in the Combined Work and reverse engineering for debugging such modifications, if you also do each of the following:

- a) Give prominent notice with each copy of the Combined Work that the Library is used in it and that the Library and its use are covered by this License.
- b) Accompany the Combined Work with a copy of the GNU GPL and this license document.
- c) For a Combined Work that displays copyright notices during execution, include the copyright notice for the Library among these notices, as well as a reference directing the user to the copies of the GNU GPL and this license document.
- d) Do one of the following:
  - 0) Convey the Minimal Corresponding Source under the terms of this License, and the Corresponding Application Code in a form suitable for, and under terms that permit, the user to recombine or relink the Application with a modified version of the Linked Version to produce a modified Combined Work, in the manner specified by section 6 of the GNU GPL for conveying Corresponding Source.
  - 1) Use a suitable shared library mechanism for linking with the Library. A suitable mechanism is one that (a) uses at run time a copy of the Library already present on the user's computer system, and (b) will operate properly with a modified version of the Library that is interface-compatible with the Linked Version.
- e) Provide Installation Information, but only if you would otherwise be required to provide such information under section 6 of the GNU GPL, and only to the extent that such information is necessary to install and execute a modified version of the Combined Work produced by recombining or relinking the Application with a modified version of the Linked Version. (If you use option 4d0, the Installation Information must accompany the Minimal Corresponding Source and Corresponding Application Code. If you use option 4d1, you must provide the Installation Information in the manner specified by section 6 of the GNU GPL for conveying Corresponding Source.)

## 5. Combined Libraries.

You may place library facilities that are a work based on the Library side by side in a single library together with other library facilities that are not Applications and are not covered by this License, and convey such a combined library under terms of your choice, if you do both of the following:

- a) Accompany the combined library with a copy of the same work based on the Library, uncombined with any other library facilities, conveyed under the terms of this License.
- b) Give prominent notice with the combined library that part of it is a work based on the Library, and explaining where to find the accompanying uncombined form of the same work.

## 6. Revised Versions of the GNU Lesser General Public License.

The Free Software Foundation may publish revised and/or new versions of the GNU Lesser General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Library as you received it specifies that a certain numbered version of the GNU Lesser General Public License “or any later version” applies to it, you have the option of following the terms and conditions either of that published version or of any later version published by the Free Software Foundation. If the Library as you received it does not specify a version number of the GNU Lesser General Public

License, you may choose any version of the GNU Lesser General Public License ever published by the Free Software Foundation.

If the Library as you received it specifies that a proxy can decide whether future versions of the GNU Lesser General Public License shall apply, that proxy's public statement of acceptance of any version is permanent authorization for you to choose that version for the Library.