PathWave
FPGA 2019

User's Guide

KEYSIGHT
TECHNOLOGIES

Notice

Table of Contents

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

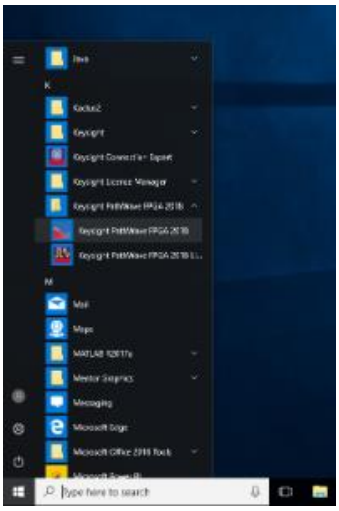
Contents

Overview

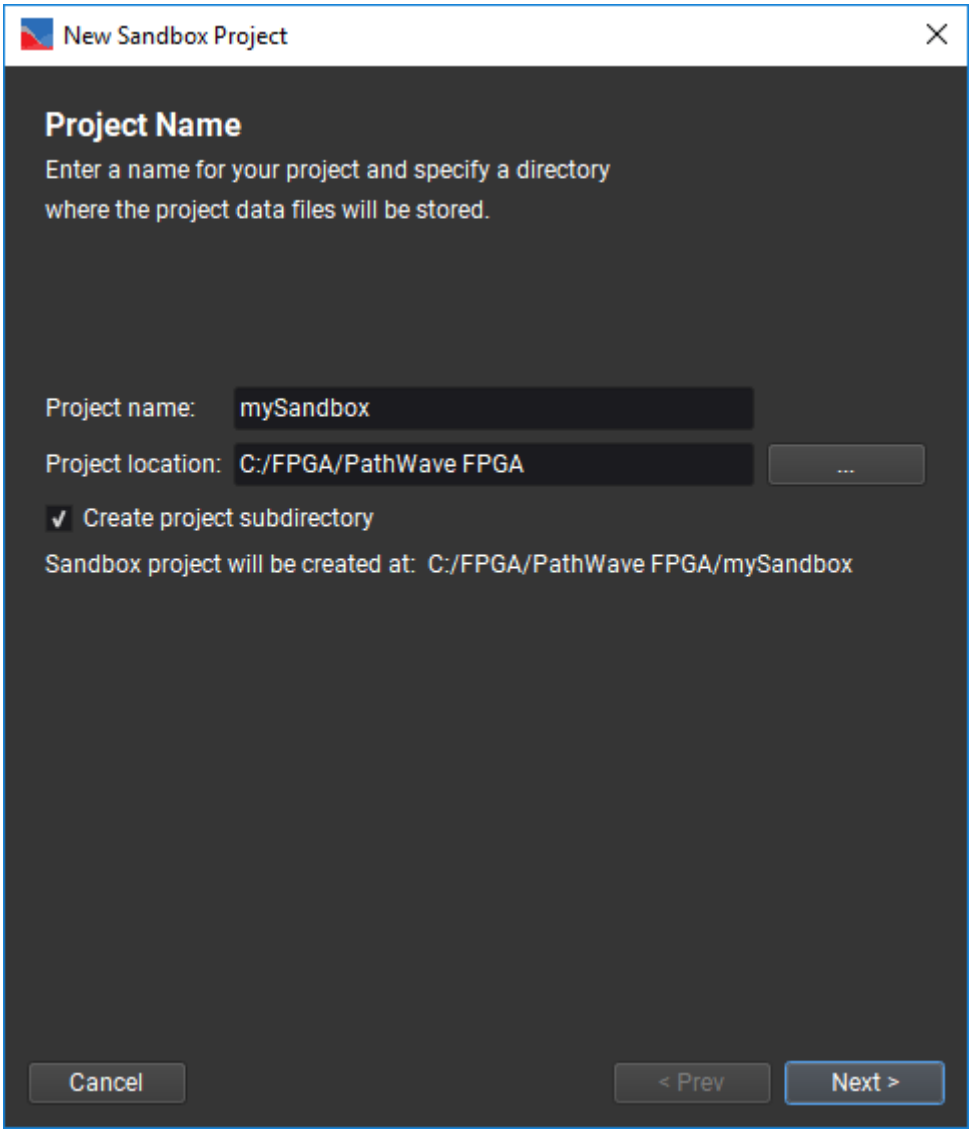
PathWave FPGA is a graphical environment that provides a way to rapidly develop FPGA designs on Keysight Open FPGA hardware. An IP library is provided which includes Logic/Math, Memory, and DSP blocks that can be included in an FPGA design. Vivado IP blocks or custom HDL IP can also be imported and the 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:

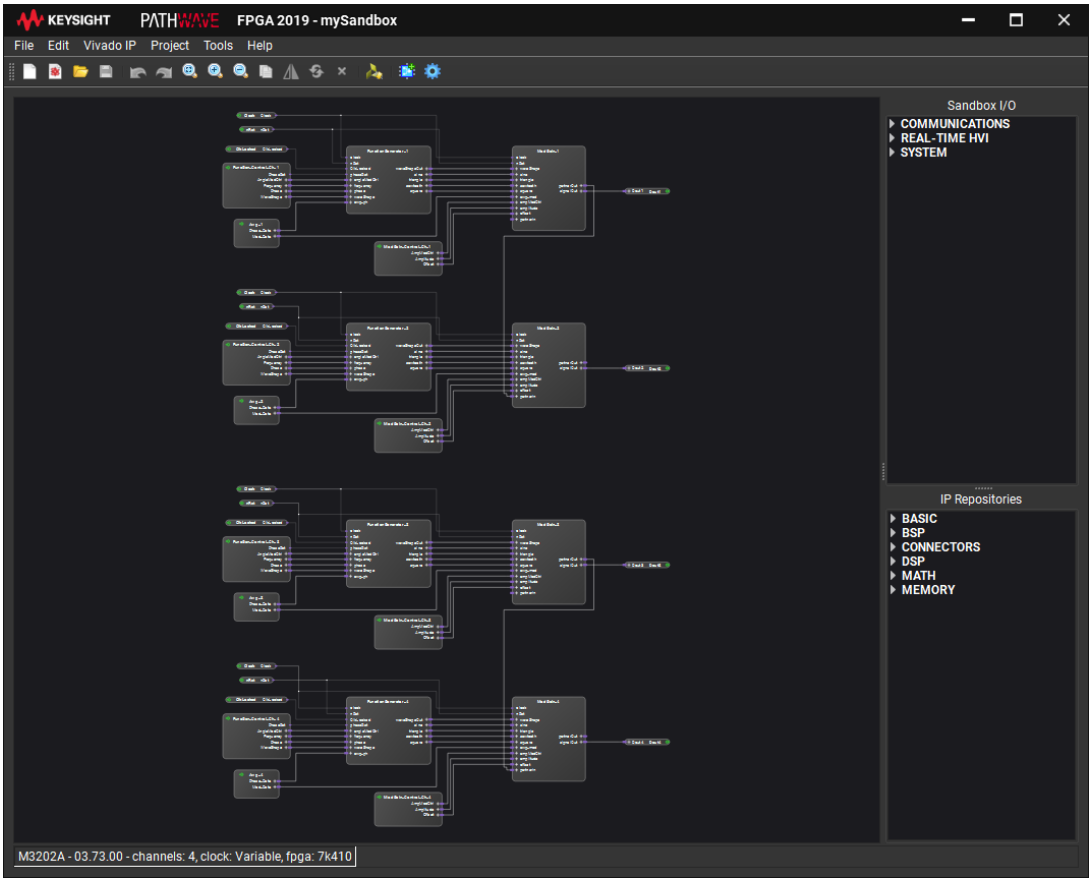
1. Start PathWave FPGA



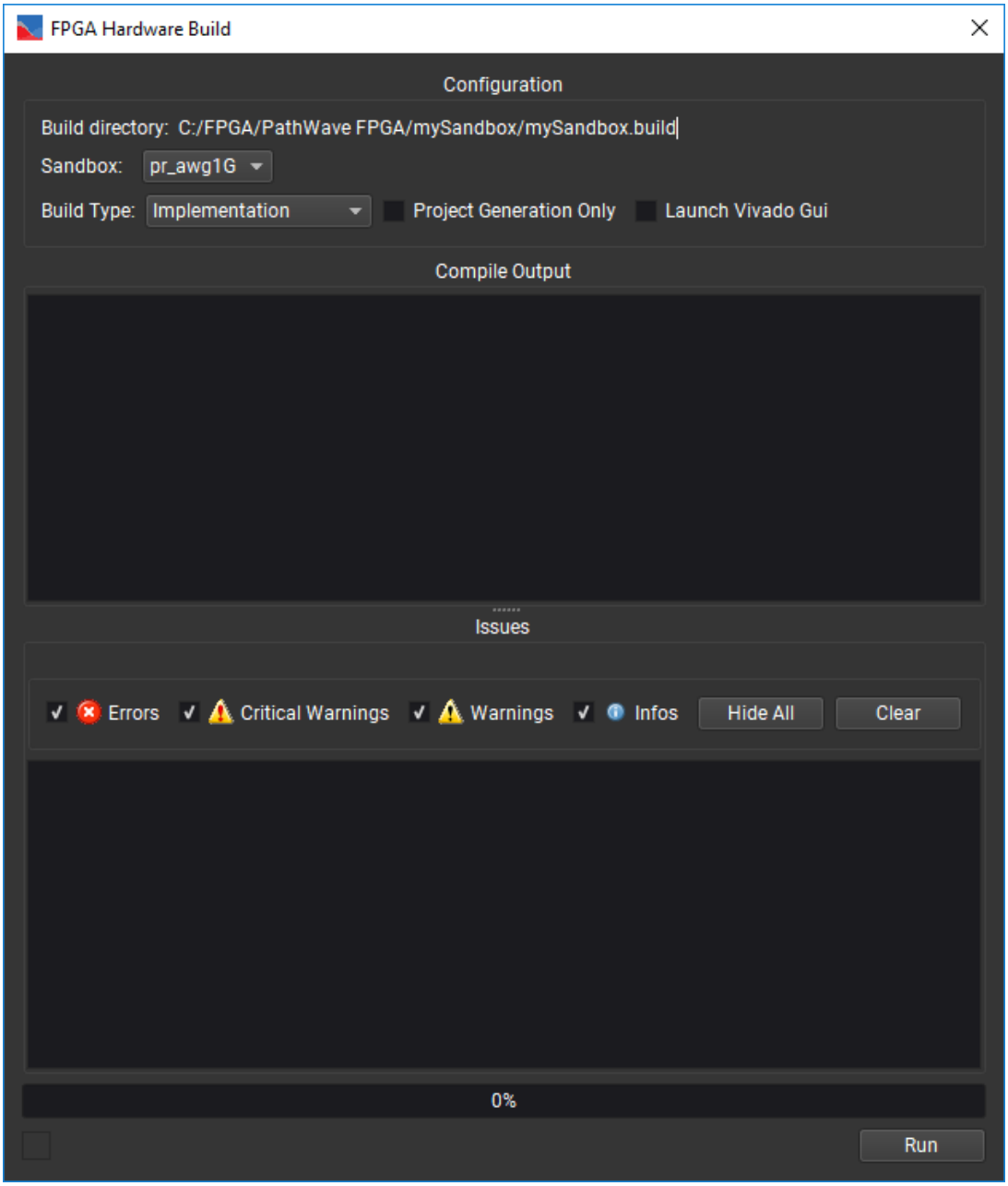
2. Create a new project with the PathWave FPGA New Project Wizard



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

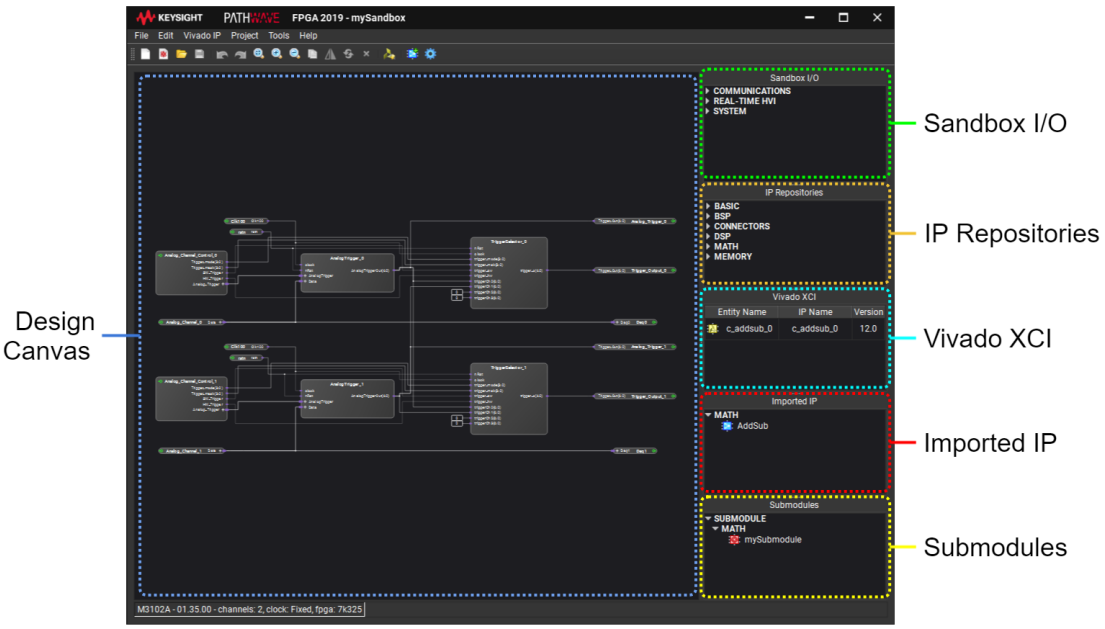


4. Compile the design into a bit image



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, add an external block, export to VHDL, create a template, configure settings, and exit.
Edit	Includes options to undo an operation, redo an operation, and select all.
Vivado IP	Includes an option to launch the Vivado IP tool.
Project	Includes and option to generate FPGA firmware.
Tools	Includes the IP Packager.
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.

Menu/Icon/Pane	Description
	Fit schematic in window.
	Zoom in.
	Zoom out.
	Copy.
	Flip.
	Redraw connections.
	Remove.
	Launch the Vivado IP tool.
	Add external block.
	Generate the firmware for the project.
Sandbox I/O	Sandbox I/O 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 Repositories	IP repositories that are built-in or custom.
Vivado XCI	Vivado XCI (Xilinx Core Instance) created either by launching the Vivado IP tool or importing Vivado XCI. Note, only visible if you have imported a Vivado XCI file.
Imported IP	Imported User IP from many different sources including: VHDL, Verilog, IP-XACT, Vivado Projects (XPR). Note, only visible if you have imported IP.
Submodules	Submodules created by PathWave FPGA. Note, only visible if you have created or added a submodule.

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

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

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	Using the shortcut menu (right-click a block), you can perform the following operations: • 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

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 PatheWave 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.

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.

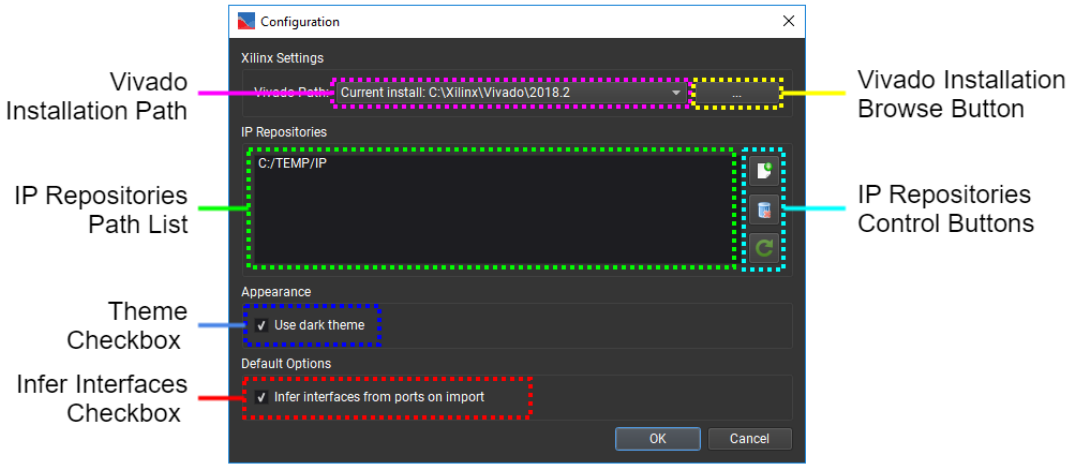
You may want to retarget a submodule to work with different hardware, or remove the targeted hardware altogether to make a generic submodule. A generic submodule can be shared with projects targeting different BSPs, but will not have access to the BSP IP.

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.

Configuring PathWave FPGA

The Configuration dialog provides some options for configuring *PathWave FPGA*. You can specify the Vivado path, IP repositories, and the appearance of the interface. Select **File > 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 that 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.

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

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

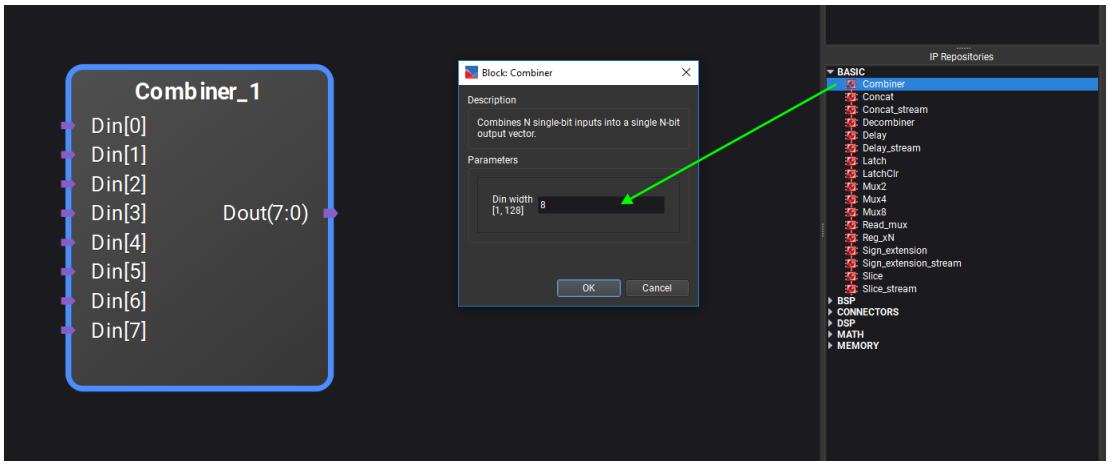
Adding Blocks

A hardware project is created by combining blocks from the panes displayed on the right side of the user interface. These are grouped under:

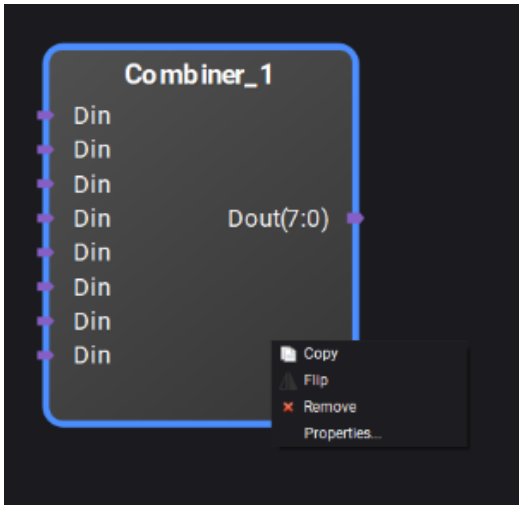
- Sandbox I/O
- IP Repositories
- Vivado XCI (Xilinx Core Instance)
- Imported User IP
- PathWave FPGA IP Repository
- PathWave FPGA Submodule

When a hardware project is opened, sandbox I/O and IP repositories that are available for the particular board support package are shown in the panes on the right. The blocks can be selected, dragged into the project, configured, and connected to other blocks in the project.

For example:



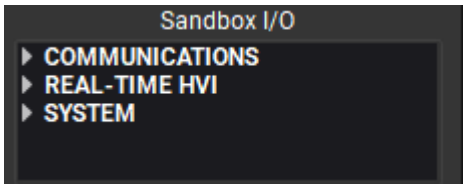
The selected block can be configured and saved.
If you select a block and right-click on it, the following options are available:



- **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.

Sandbox I/O

To communicate between the sandbox and the static region, you need to instantiate a sandbox I/O block from the Sandbox I/O pane. Each board support package provides a unique set of sandbox I/O blocks that are specific for the instrument. The sandbox I/O blocks are grouped based on the function of their connections to the "outside world". The interfaces of a sandbox are collapsed, in order to show the different categories of sandbox I/O:



Apart from categorizing, some sandbox I/O blocks can 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:

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. Launch PathWave FPGA.
2. Open/Create a project you wish to edit.
3. With the project open, in the Sandbox I/O 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**.
4. 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 group of registers with a contiguous address space. Each register in a Register Bank is editable by the user. Below are the major sections of the Register Bank Dialog.

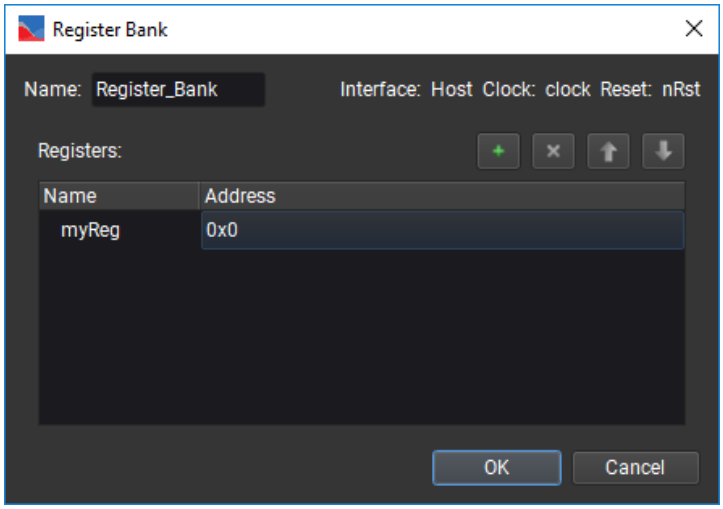


Figure 1: Register Bank Dialog when opened into a new project.

There are 5 main areas to inspect on 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). If the name is not valid, it will be converted to a valid and unique name.

2. Memory Mapped Components - This is the main portion of the Register Bank Dialog. You can edit registers that are contained within the Register Bank here.
   a. Name - This column represents the name of a register. Double left click on the register name to change from the default name. A register name must be unique within the bank, and have valid HDL syntax (see Naming Conventions).
      i. If the Register Dialog detects an issue with the name of a register, it will turn the text red and display a tool tip stating the reason for the failure.
   b. Address - This column represents the byte offset address of a register. The user is not allowed to directly edit this field, it is for informational use only.
   c. Reordering Registers - It is possible to reorder registers in the Register Bank by selecting one, then clicking and dragging it to the location you wish it to go. This changes the address field of the moved register and updates addresses of other registers affected by the move.
3. Add/Remove/Reorder - This section of the dialog is used for manipulating the number and order of registers present in the Register Bank.
   a. The user can add registers to the design if no issues are detected inside the Register Bank. The button will be disabled, when an issue is detected.
   b. The user can remove registers at any point. Any currently selected registers will be removed from the Register Bank.
      i. Another way to remove registers is to use the "Delete" key.
   c. A selected register may be reordered by clicking the up or down arrow.
4. OK/Cancel - This section of the dialog is used to exit the dialog. Clicking OK will create a Register Bank that can be placed on the schematic, while cancel closes 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.

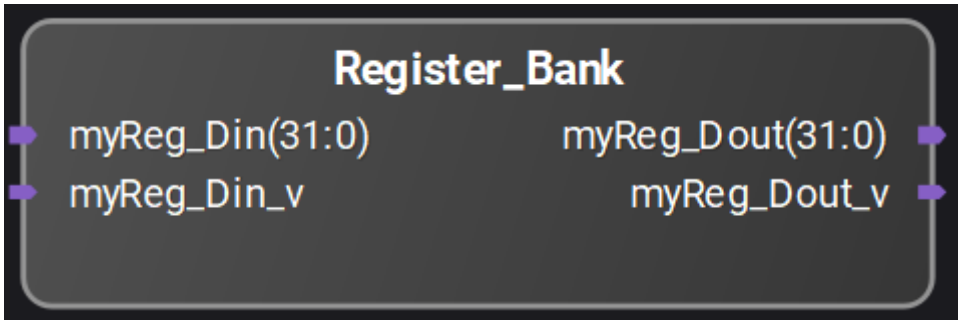


Figure 2: Register Bank block when placed onto the schematic.

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.

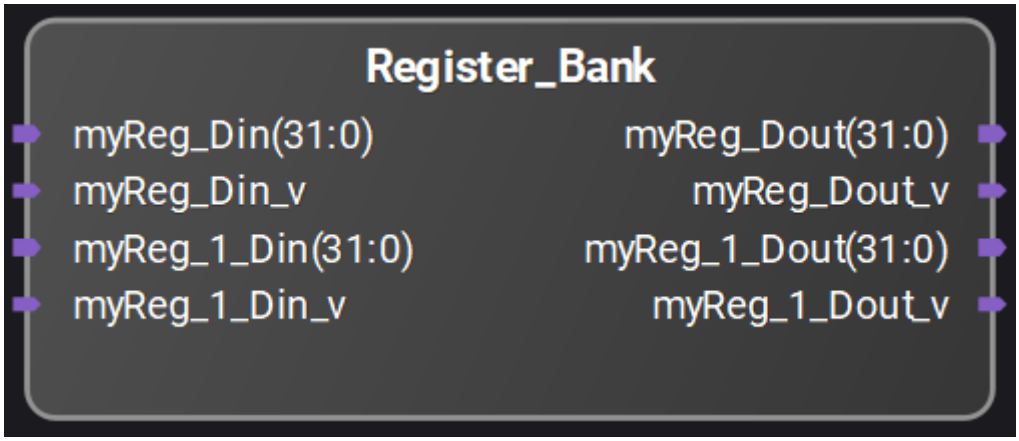


Figure 3: Register Bank block that has two RW registers in it.

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".

IP Repositories

IP repositories are libraries of blocks that are loaded into PathWave FPGA. There are three 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 user-defined list of directories that include IP definitions. These directories can be defined in the Settings dialog (File → Settings). Important: A project should be reloaded, in order for the added IP to be loaded. To load an IP repository, use the Settings Dialog. To learn how to create an IP repository, refer to the IP Developers Guide.

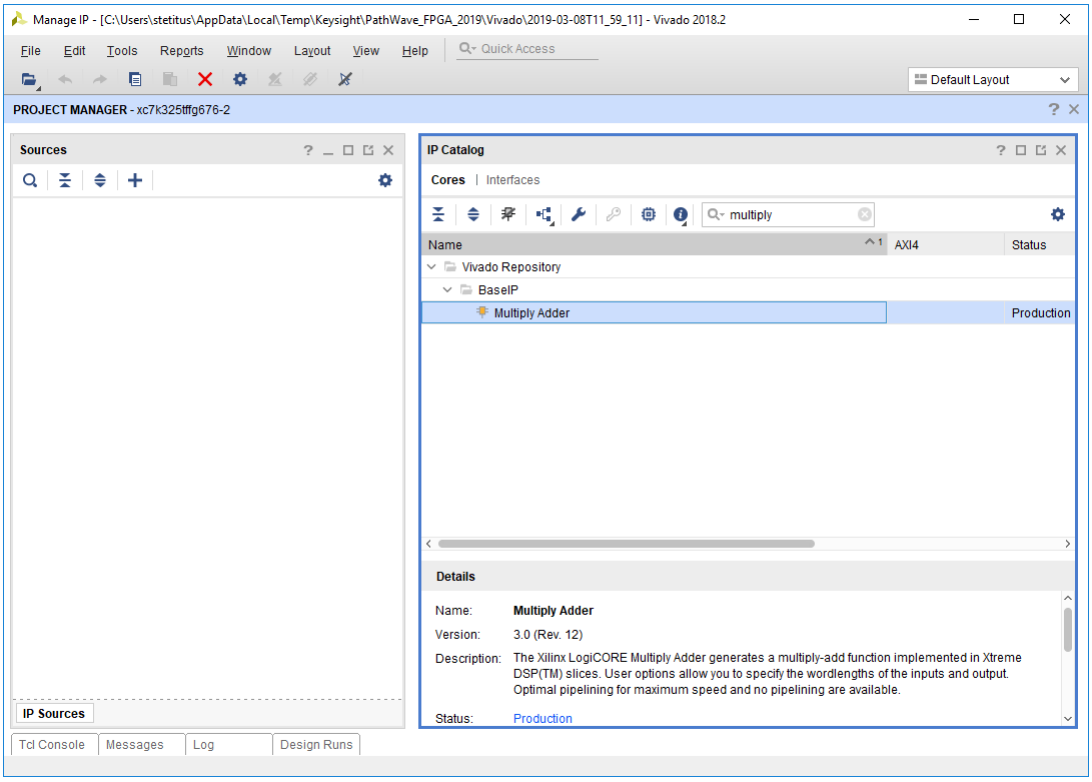
IP will be found recursively in each repository location. All valid IP will be added into the library blocks. If any problems are encountered with loading, a dialog will popup to display the errors. Xilinx Vivado IP is excluded from this search.

Vivado XCI (Xilinx Core Instance)

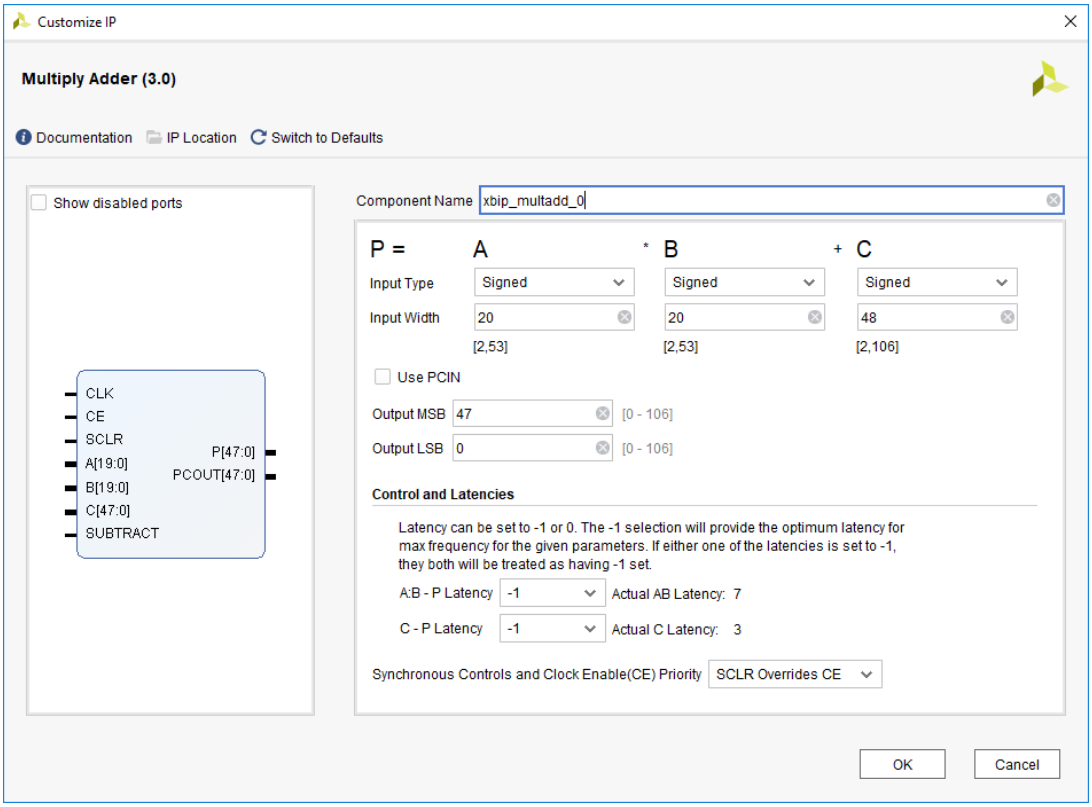
Invoking Vivado IP tool

PathWave FPGA allows you to import Vivado IPs from the Xilinx Vivado IP Catalog and integrate them into your project.

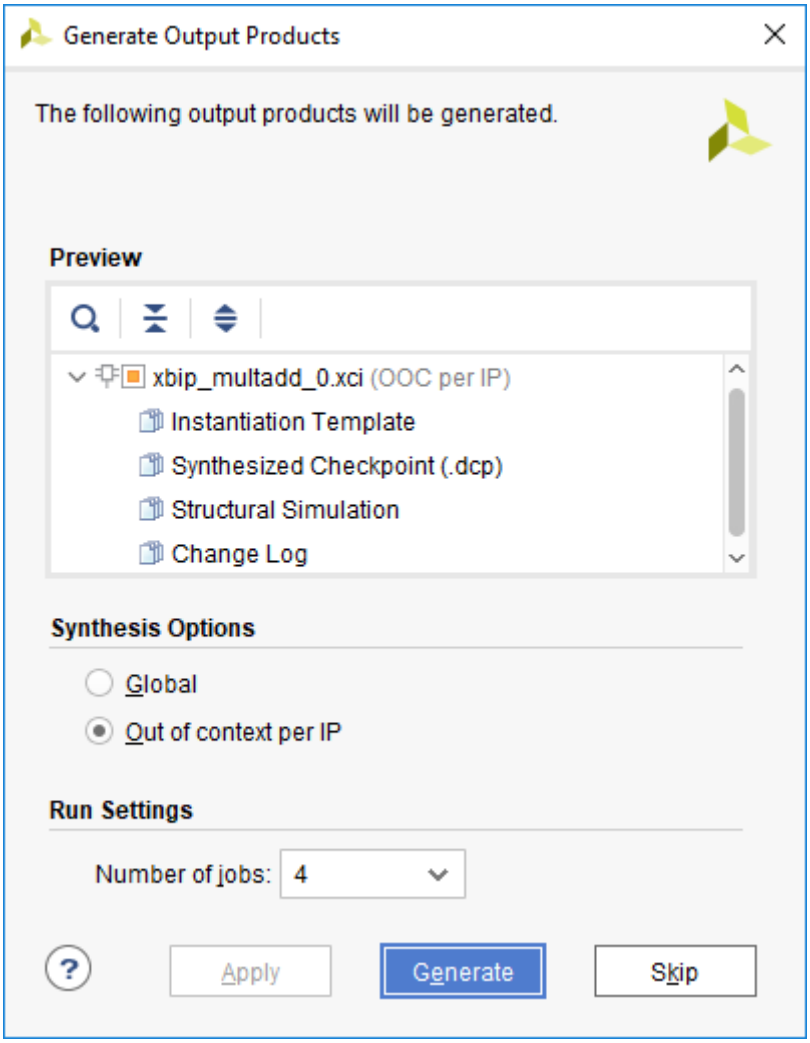
1. Click on the **Launch Vivado IP Tool** button on the main toolbar.
2. Select a Vivado IP block from the IP Catalog and double-click it.



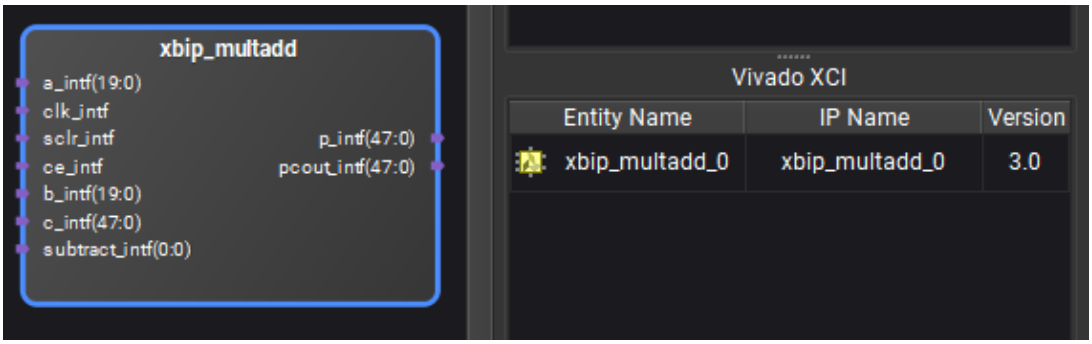
3. Configure the IP properties and then press OK.



4. Click the Skip button. PathWave FPGA always regenerates Vivado IP during bitfile generation, so the output products created by clicking Generate are not needed.



5. If you need any other Vivado IP, repeat steps 2-4 to generate them. When you are done, close Vivado.
6. PathWave FPGA will show the configured IP in the Vivado XCI section of the library. 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.

Imported User IP

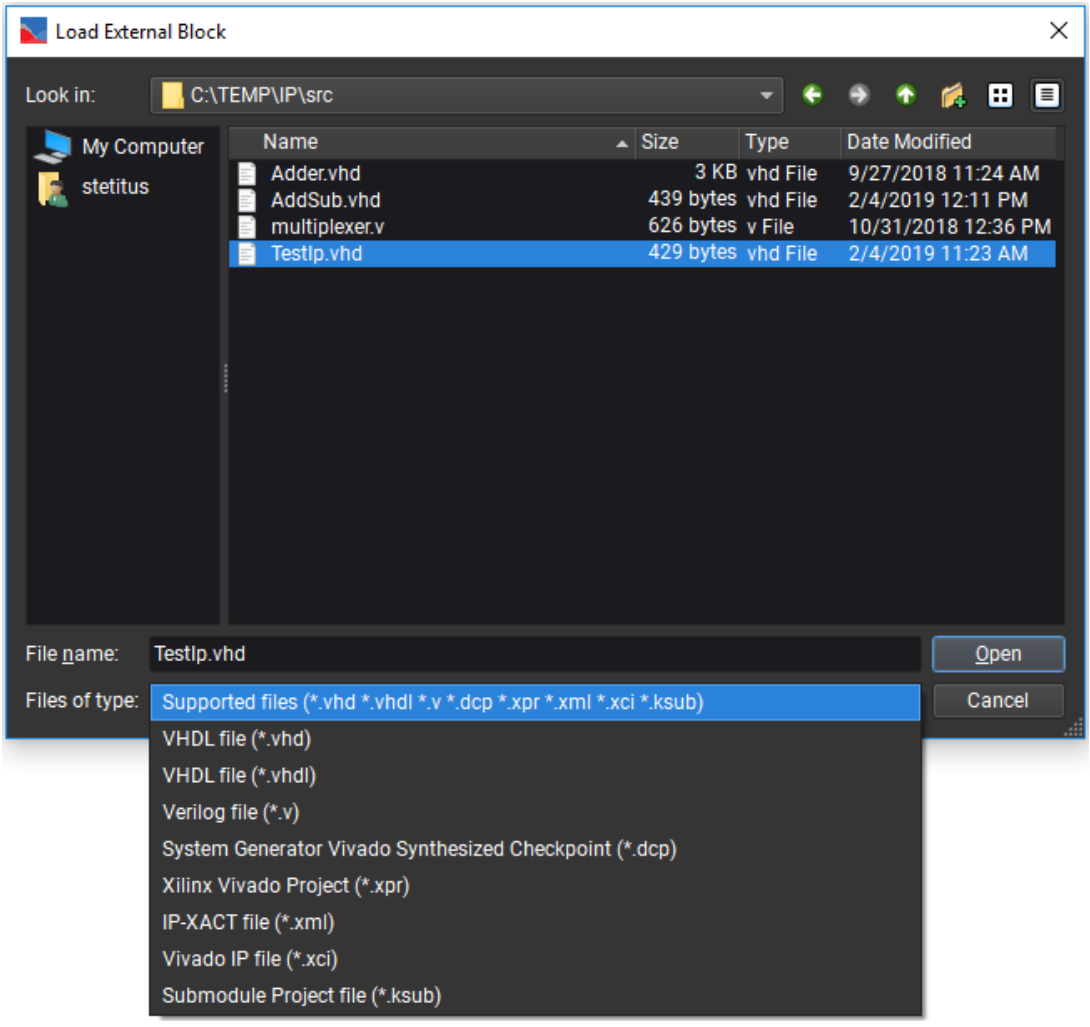
In addition to IP developed using the Library tools, the PathWave FPGA software allows importing and integration of custom IP into a project. User IP is developed using external FPGA tools; the PathWave FPGA software is not intended for developing IP from scratch. However, once the user has created an 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, *.vhdl)
- Verilog source files (*.v).
- Xilinx Vivado projects (*.xpr).
- System Generator Vivado Synthesized Checkpoints (*.dcp).
- IP-XACT files (*.xml).
- Vivado IP files (*.xci)
- PathWave FPGA Submodules

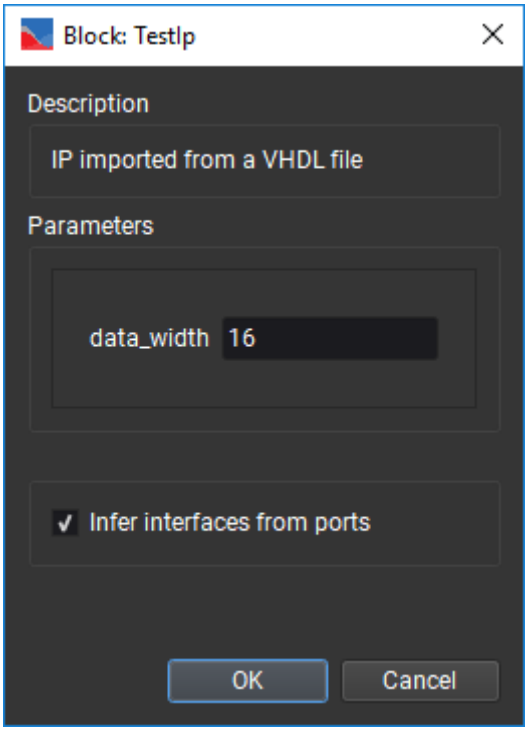
To import a user IP:

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.



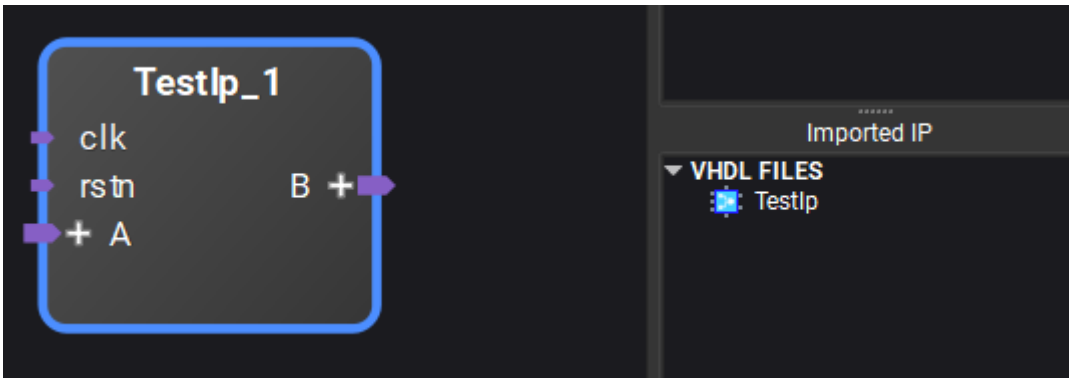
2. Navigate to select the file to be imported into the project. Click **Open** to import the file.

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 User IP External Block region 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.

PathWave FPGA IP Repository

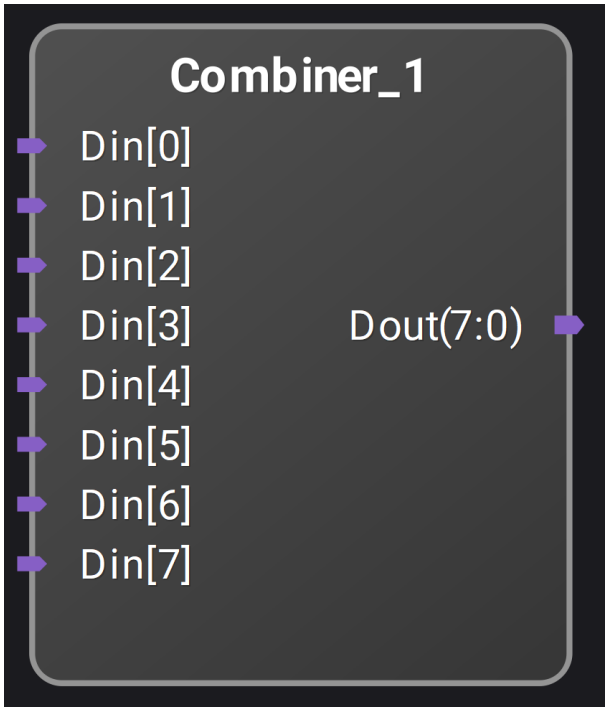
PathWave FPGA includes some IP blocks that a user can incorporate into their FPGA design. The IP blocks are categorized into different libraries so that similar blocks are grouped together. Below is a description of the IP blocks included in PathWave FPGA.

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.

- Shift_stream
- DSP
  - Combine1toN
  - Complex2Real / Real2Complex
  - DecimateBy5
  - DecimateBy5 Complex
  - InterpolateBy5
  - InterpolateBy5 Complex
  - Lo
  - Lo5_dc
  - Lo5_uc
  - Power2Decimator
  - Power2Interpolator
- Memory
  - DualPortRam
  - DualPortRam_stream
  - Mem_mux_2x
  - Mem_mux_4x
  - Streamer blocks (2 channels at 32 bits/channel) – Streamer32x2/Streamer32x2b

Basic IP blocks

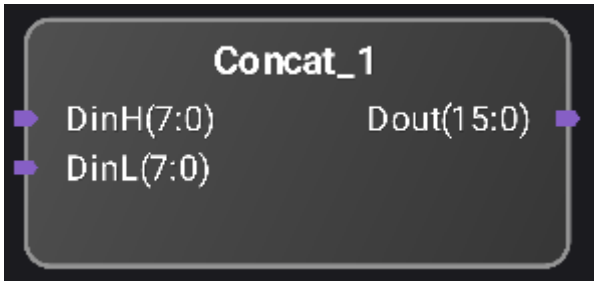
Combiner



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 64. 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.

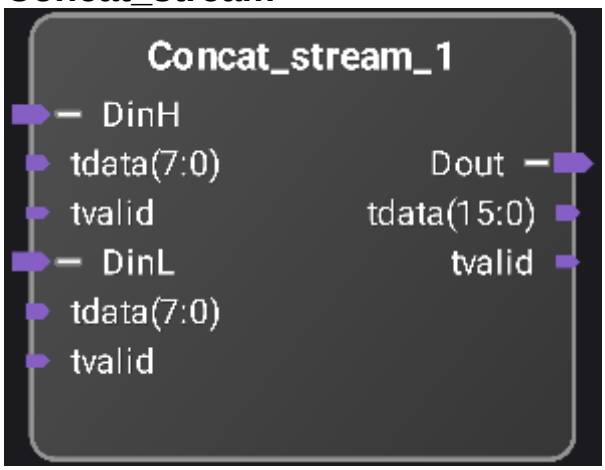
This module does not introduce extra delay.

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.

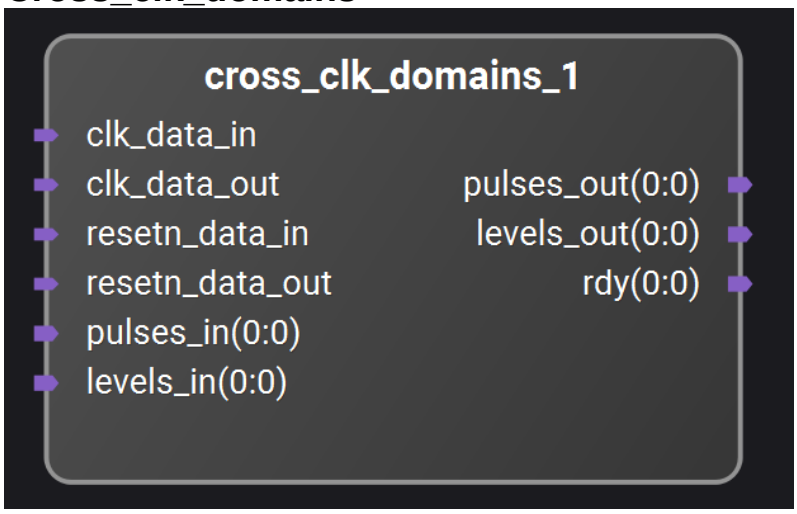
Note that both streaming inputs must assert and deassert tvalid at the same time.

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.

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.

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

'resetn_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.

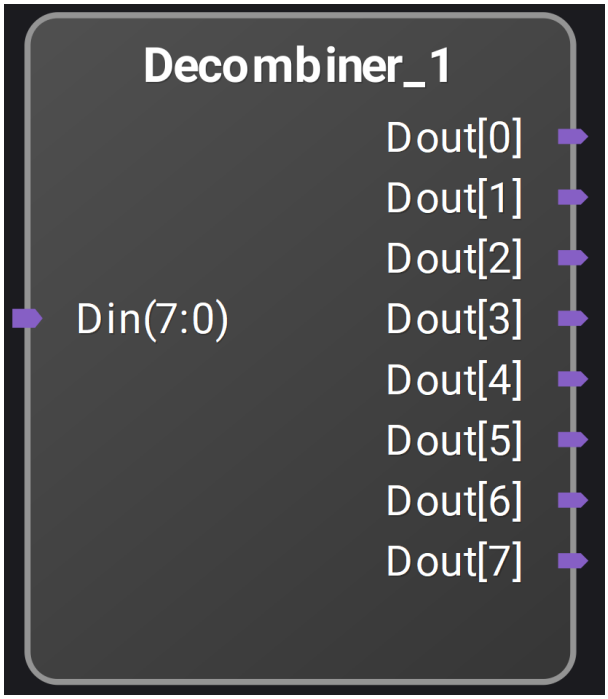
Parameters

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

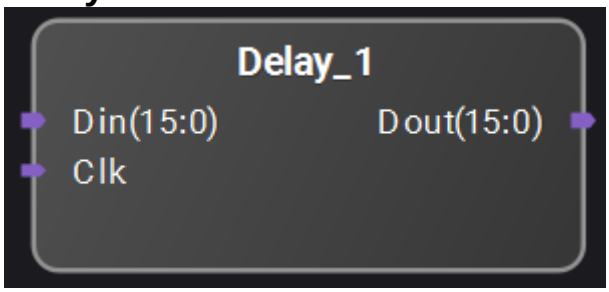


Converts a single N-bit input vector into N single-bit output signals.

Parameters

Din width: Sets the Din data width. Variable from 1 to 64. Default is 8.

Delay



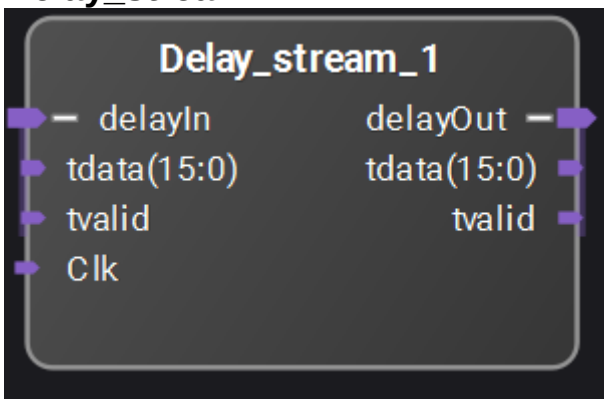
Delays input N cycles.

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.

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.

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

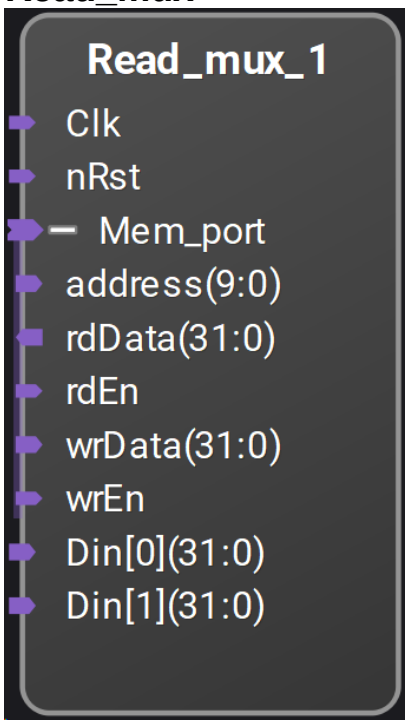
These are 2 to 1, 4 to 1, and 8 to 1 multiplexers. The value of the Sel input determines which of the various In ports connect to Result.

These are combinatorial.

Parameter

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

Read_mux

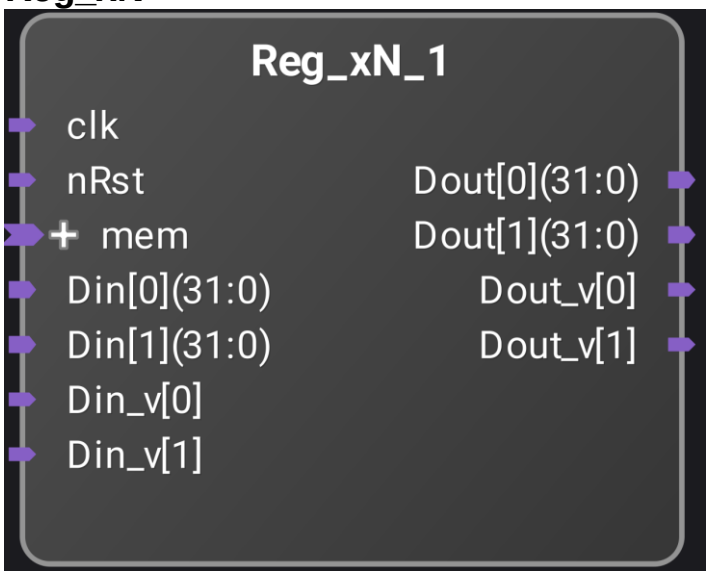


Read data from multiple sources.

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.

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 64. Default is 2.

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

sign_extension

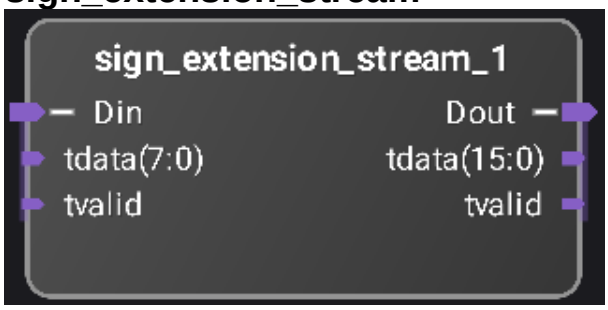
Sign extends the input vector.

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 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.

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.

Connectors

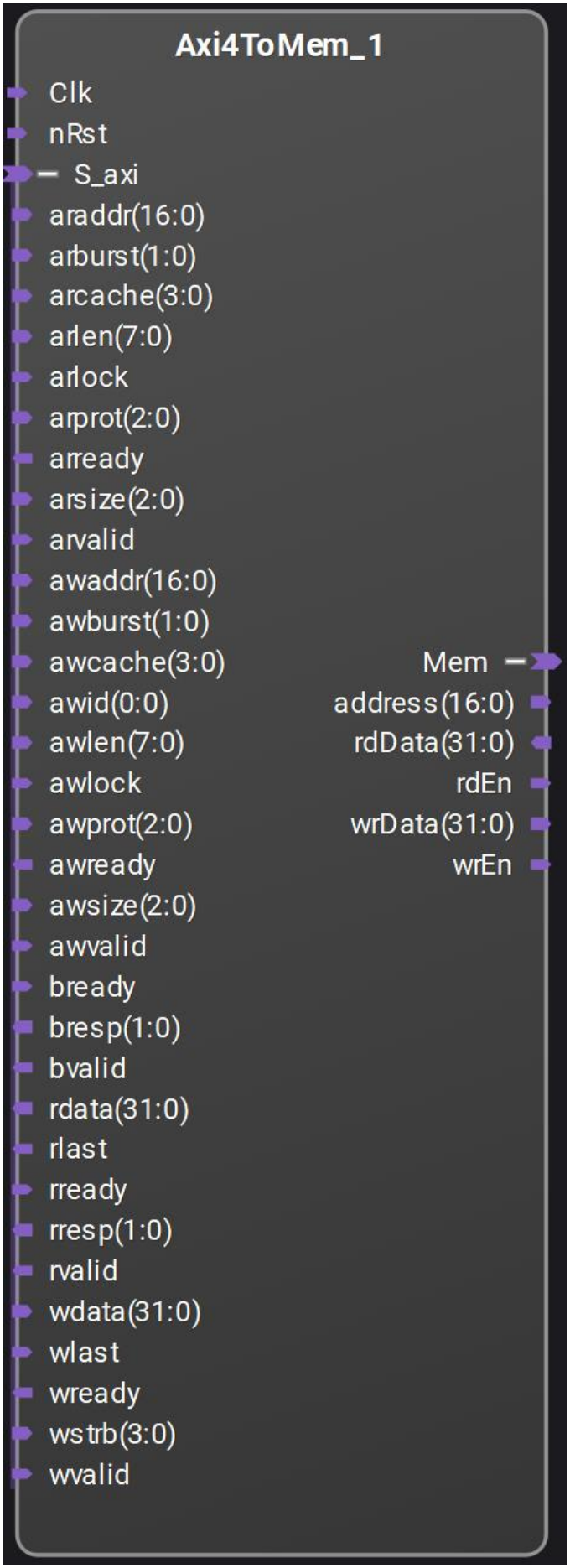
Axi4liteToMem

Converts Axi4Lite slave interface to PC_Mem master interface.

Parameters

Address width: Sets the AXI interface and Mem interface address width. Default is 8.

Axi4Tomem


Axi4ToMem_1
Clk
nRst
S_axi
araddr(16:0)
arburst(1:0)
arcache(3:0)
arlen(7:0)
arlock
arprot(2:0)
arready
arsize(2:0)
arvalid
awaddr(16:0)
awburst(1:0)
awcache(3:0)
awid(0:0)
awlen(7:0)
awlock
awprot(2:0)
awready
awsize(2:0)
awvalid
bready
bresp(1:0)
bvalid
rdata(31:0)
rlast
rready
rresp(1:0)
rvalid
wdata(31:0)
wlast
wready
wstrb(3:0)
wvalid
Mem
address(16:0)
rdData(31:0)
rdEn
wrData(31:0)
wrEn

Converts Axi4MM slave interface to PC_Mem master interface.

Parameters
Address width: Sets the AXI interface and Mem interface address width. Default is 8.

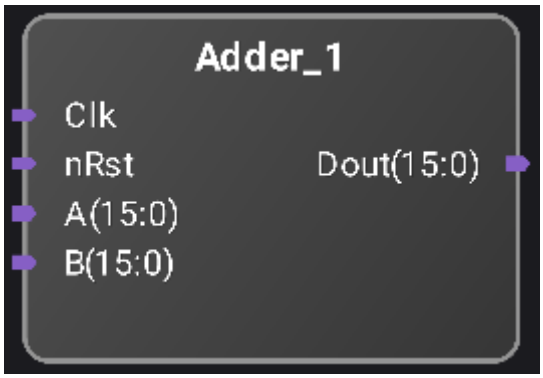
AXIStream_Broadcaster

Broadcasts AXI4 streaming data from one master to multiple slaves.

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.

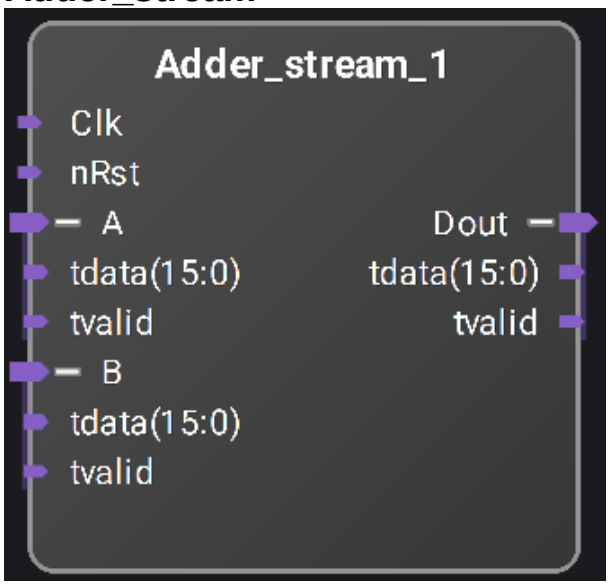
Adder implementation: Selects saturate or full precision adder modes. Default is Saturate.

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.

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

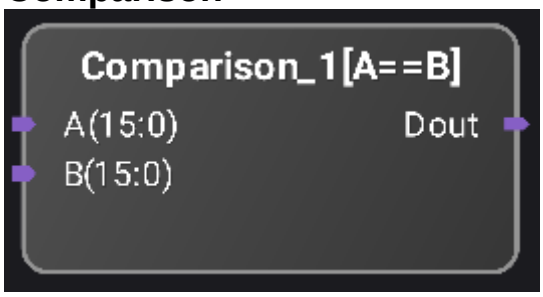
input width: Sets the bus width of the A and B inputs. Default is 16.

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.

Comparison



Comparisons between inputs A and B.

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

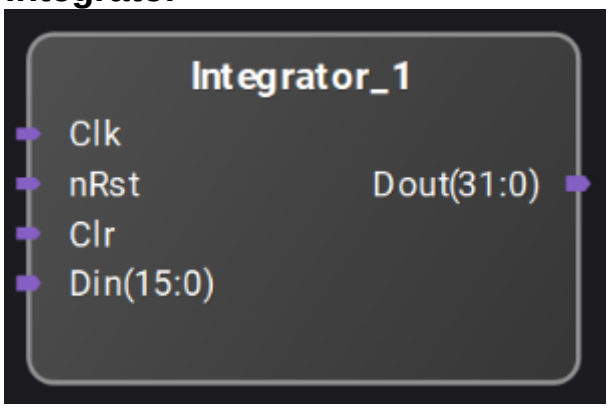
Parameters

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.

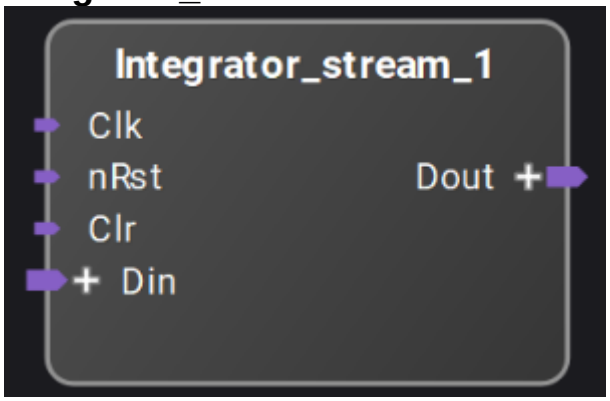
output_width: Sets the bus width of the internal accumulator and the output. Variable from 1 to 1024. Default is 32.

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 oly when the 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.

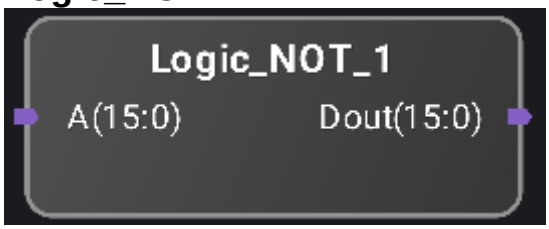
output_width: Sets the bus width of the internal accumulator and the output. Variable from 1 to 1024. Default is 32.

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

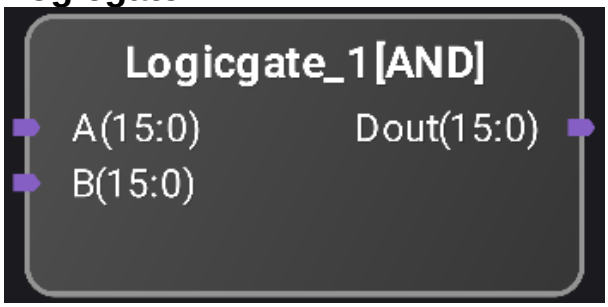


Logic NOT operation.

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.

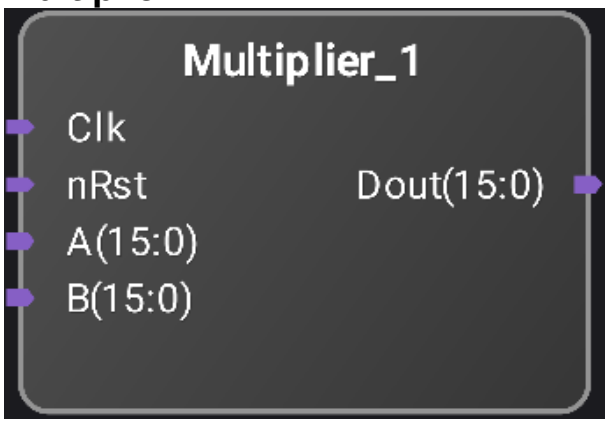
The operation parameter determines which logical operation is performed from AND, OR, XOR, NAND, NOR, and XNOR.

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, output length is the sum of both inputs lengths minus 1.

This block adds a delay of 1 cycle.

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

When the Dout width is less than Input A width + Input B width, Dout will consist of the lower bits of the product.

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.

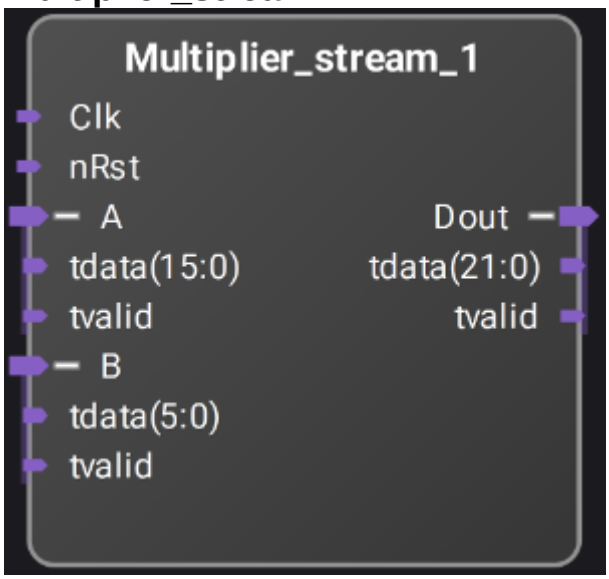
B signed: Select when the B input data is signed. Default is unsigned.

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.

Multiplier_stream



Multiplier (DSP core).

Input lengths and signedness are configurable.

When both inputs are signed, output length is the sum of both inputs lengths minus 1.

This block adds a minimum delay of 1 cycle.

Pipeline increases the total delay by an additional clock cycle.

When the Dout tdata width is less than Input A tdata width + Input B tdata width, Dout will consist of the lower bits of the product.

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.

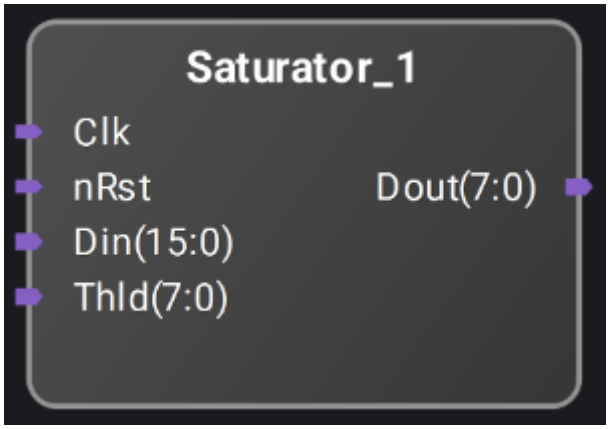
B signed: Select when the B input data is signed. Default is unsigned.

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.

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

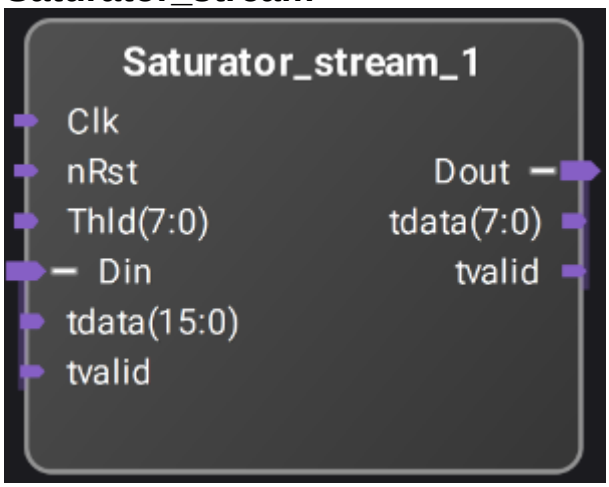
Din signed: Select when the data on the Din input is signed. Default is signed.

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.

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

Saturator_stream



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

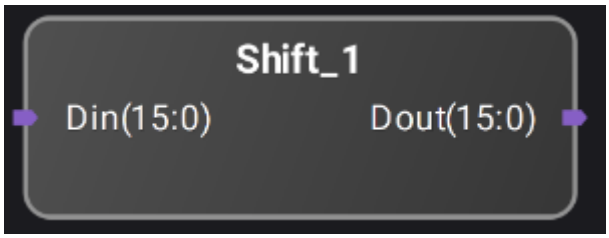
Din signed: Select when the data on the Din input is signed. Default is signed.

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.

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

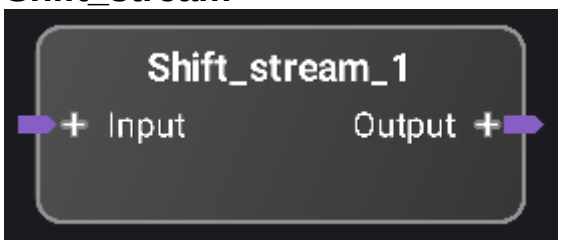
bus width: Sets the data width of the Din and Dout ports. Variable between 1 and 1024. Default is 16.

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.

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.

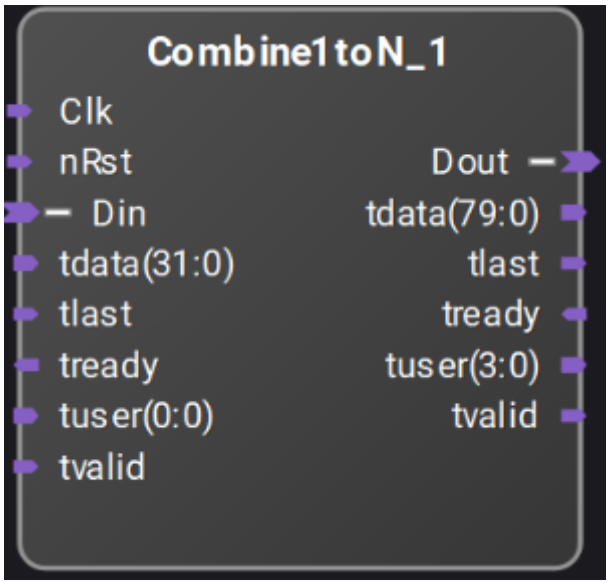
shift direction: Sets the direction to shift the Din data. Possible options are Left shift or Right shift. Default is Left shift.

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.

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

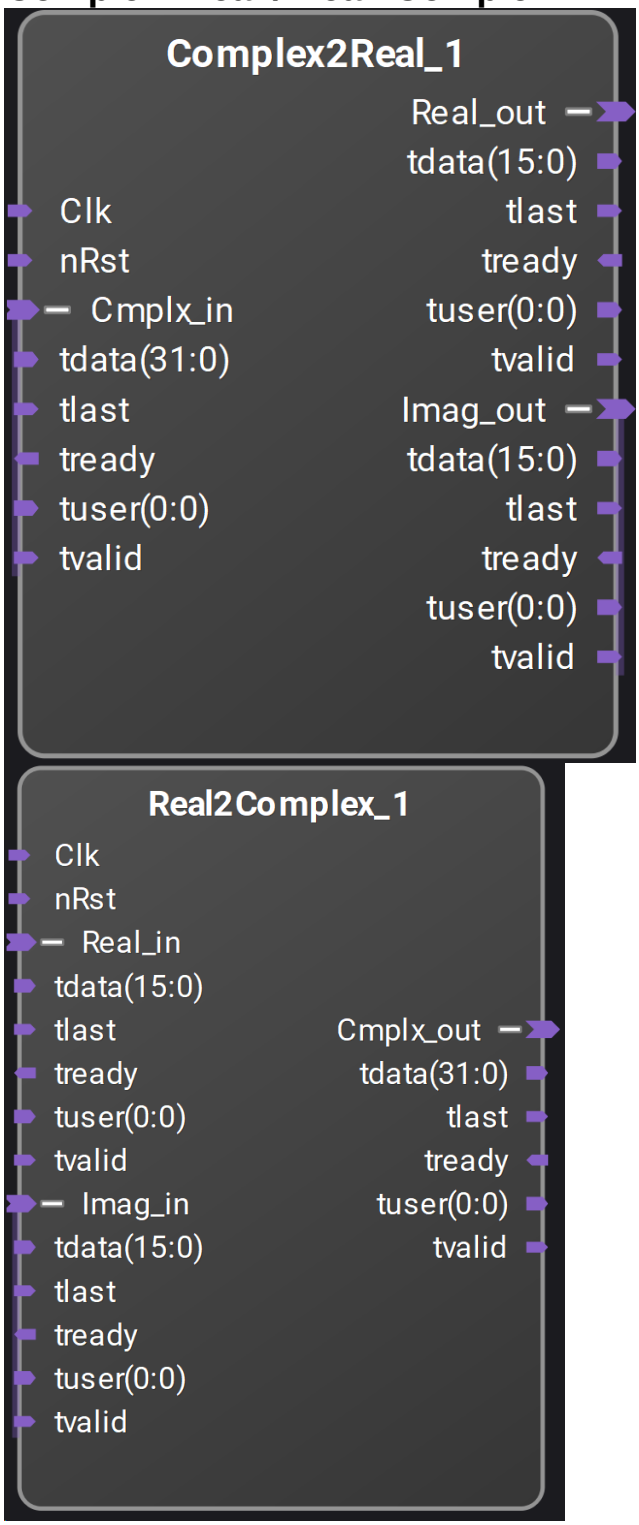
Tdata size: This sets the data width of Din_tdata. Dout_tdata will be N or N+1/2 times this value in width.

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.

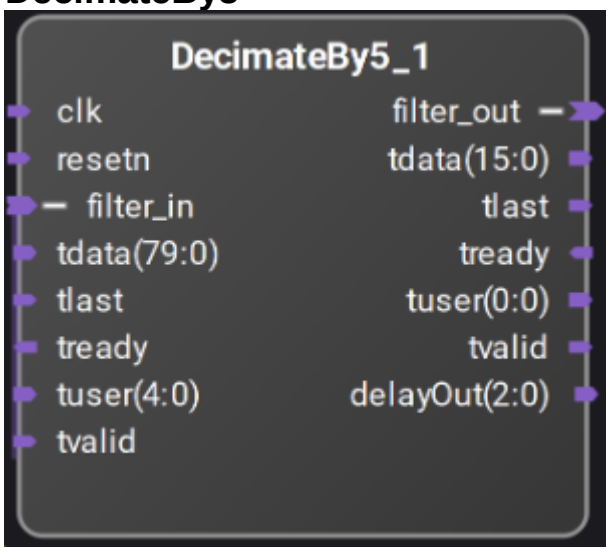
Parameters

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.

DecimateBy5



Decimate 5x, supersampled streaming input by a factor of 5. Decimation is achieved using a polyphase, FIR filter.

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.

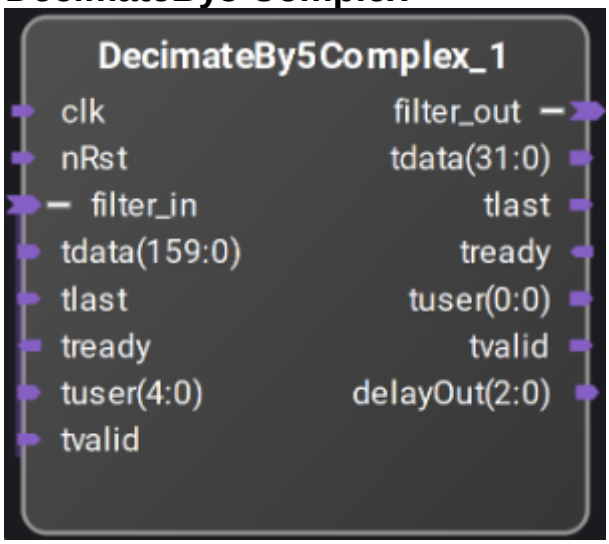
filter_out_tuser will be asserted to indicate the corresponding sample of the output stream. If *User Data Width* is greater than one, then the tuser input will have 5 * *User Data Width* bits and the tuser output will have *User Data Width* bits.

Parameters

Data Width: This sets the input and output sample widths. Note the input tdata width is 5 x *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.

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 16 bits of the output are real output data and the upper 16 bits of the output are imaginary output data.

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.

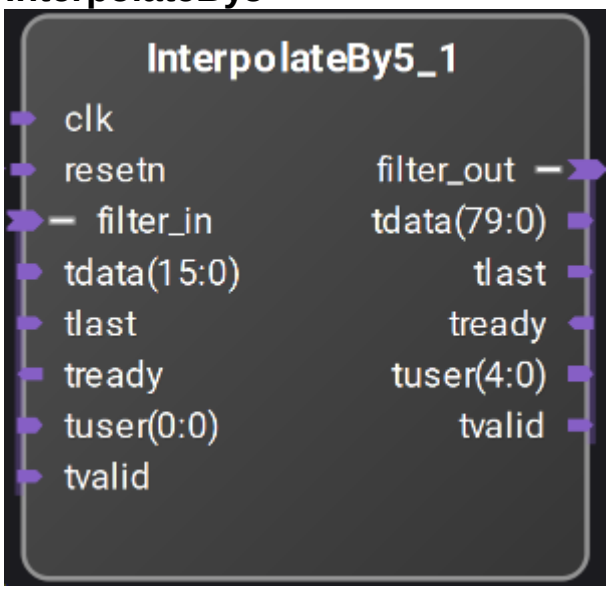
filter_out_tuser will be asserted to indicate the corresponding sample of the output stream. If *User Data Width* is greater than one, then the tuser input will have 5 * *User Data Width* bits and the tuser output will have *User Data Width* bits.

Parameters

Data Width: This sets the input and output sample widths. Note the filter_in_tdata width is 10 x *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.

InterpolateBy5



Interpolate an input stream by a factor of 5. Interpolation is achieved using an oversampled, FIR filter.

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.

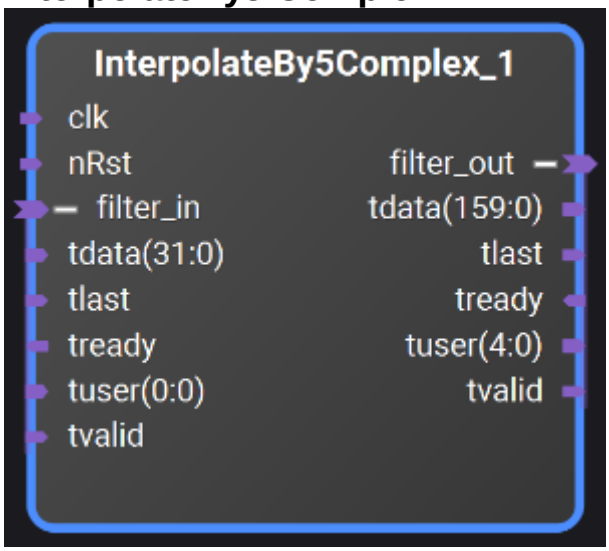
If *User Data Width* is greater than one, then the tuser input will have *User Data Width* bits and the tuser output will have *5 * User Data Width* bits.

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.

InterpolateBy5 Complex



Interpolate a complex input stream by a factor of 5. Interpolation is achieved using an oversampled, FIR filter.

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.

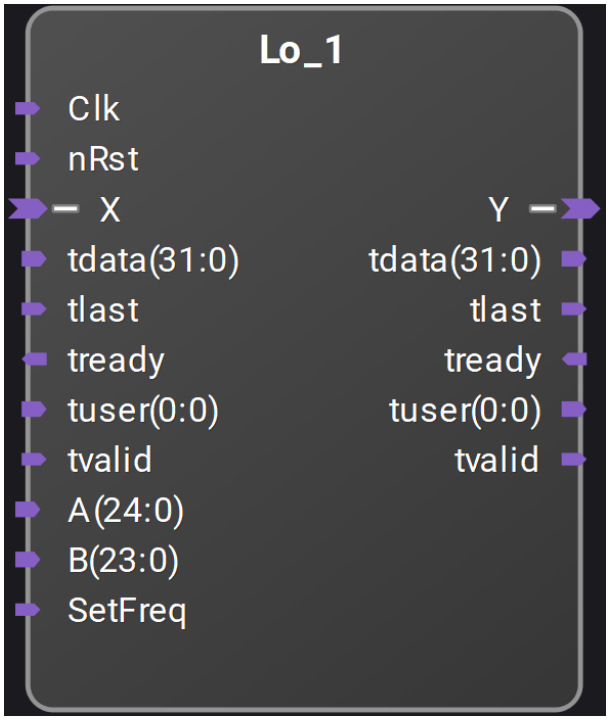
If *User Data Width* is greater than one, then the tuser input will have *User Data Width* bits and the tuser output will have *5 * User Data Width* bits.

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.

Lo



Parameterized Local Oscillator. It handles supersampled or non-supersampled data, and either the input and/or the output can be real or complex.

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<=T<2S).
The LO frequency is given by $f = f_s * (A+B/5^{10})/((S/T)*2^{25})$. 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 <= f/f_s <= 1/2. Values of A and B that are outside this range will give incorrect results.

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.

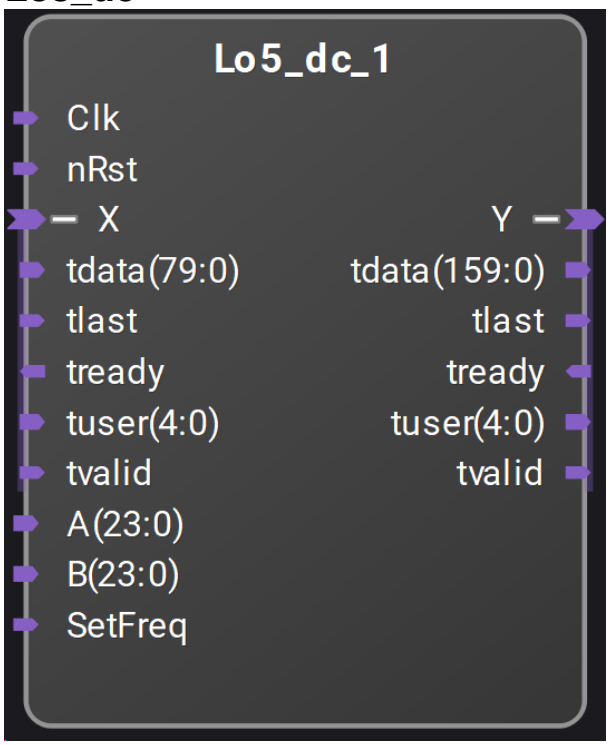
Complex Output: If set, then the output data is complex. If cleared, then only the real part of the output data is generated.

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).

Lo5_dc



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.

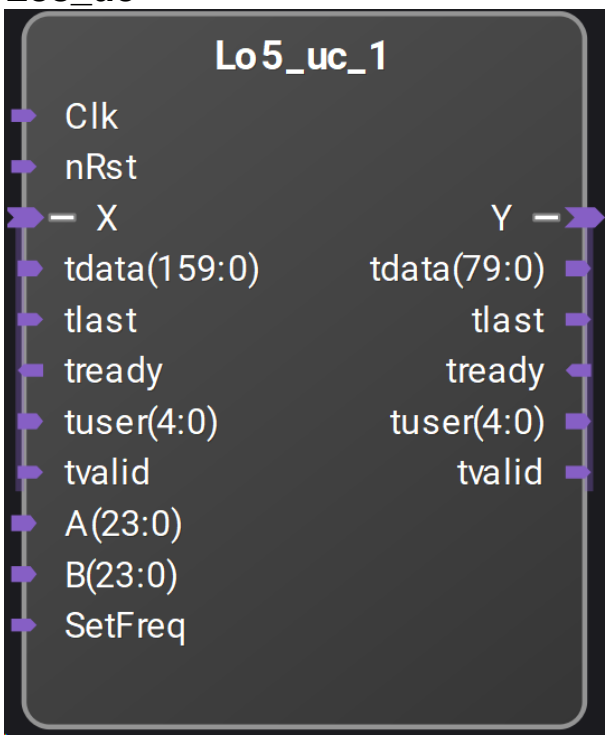
The LO frequency is given by $f = f_s * (A+B/5^{10})/(5*2^{22})$. For a sample rate, f_s, of 1 Gs/s, this results in an even decimal frequency resolution of 0.01 Hz.

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



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.

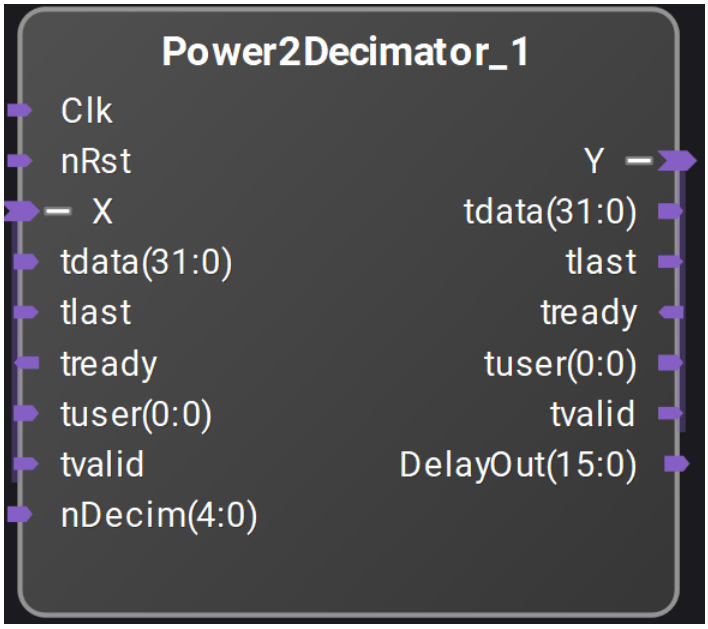
The LO frequency is given by $f = f_s * (A+B/5^{10})/(5*2^{22})$. For a sample rate, f_s, of 1 Gs/s, this results in an even decimal frequency resolution of 0.01 Hz.

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



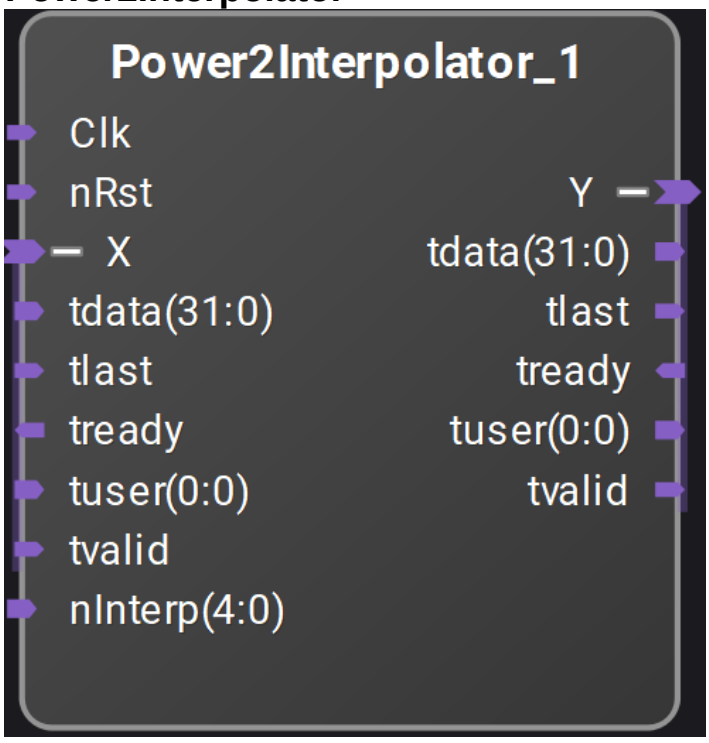
This is a power of two decimation filter that operates on complex data. It accepts complex data at up to one sample per clock. It filters and decimates the data by 2^N, where N=0...16.

Parameters

Tdata size: This sets the data width of the samples. Since both the input and output are complex, the width of the tdata busses are twice this value.

Tuser size: This sets the tuser bits per sample.

Power2Interpolator



This is a power of two interpolation filter that operates on complex data. It accepts complex data and interpolates and filters the data by 2^N, where N=0...16, generating up to one complex output sample per clock.

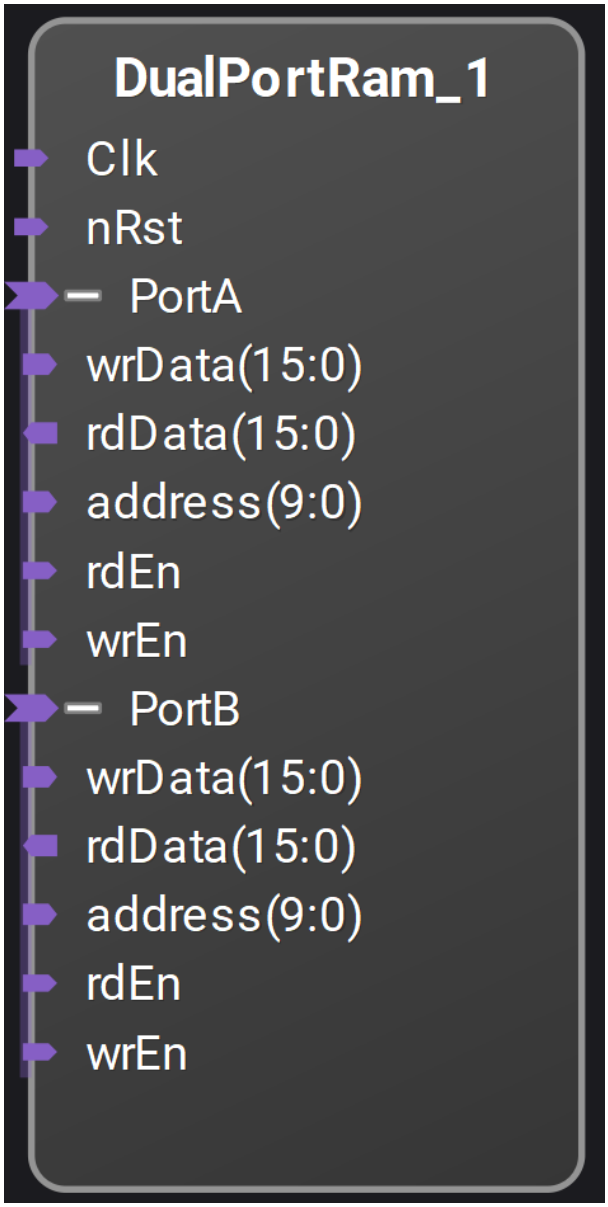
Parameters

Tdata size: This sets the data width of the samples. Since both the input and output are complex, the width of the tdata busses are twice this value.

Tuser size: This sets the tuser bits per sample.

Memory

DualPortRam



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

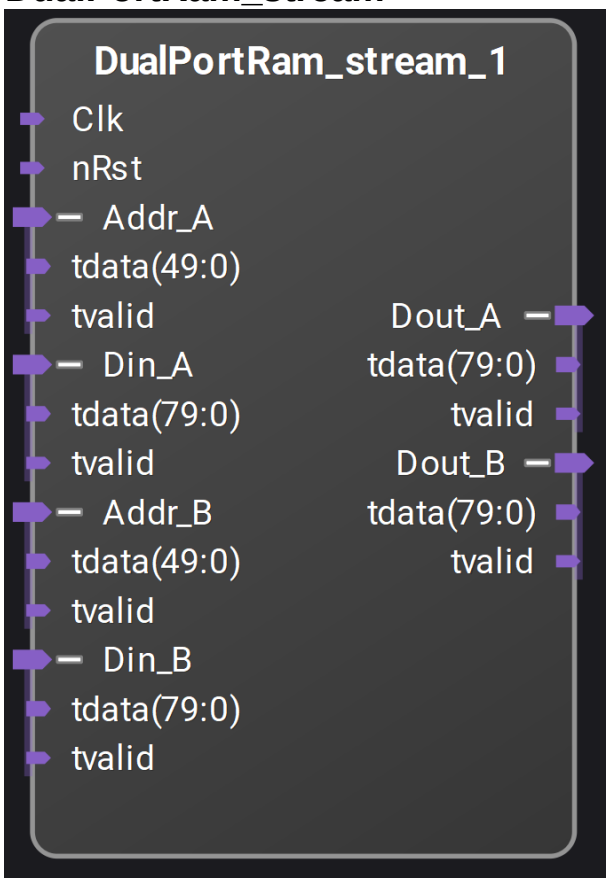
Read latency is 1 cycle.

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.

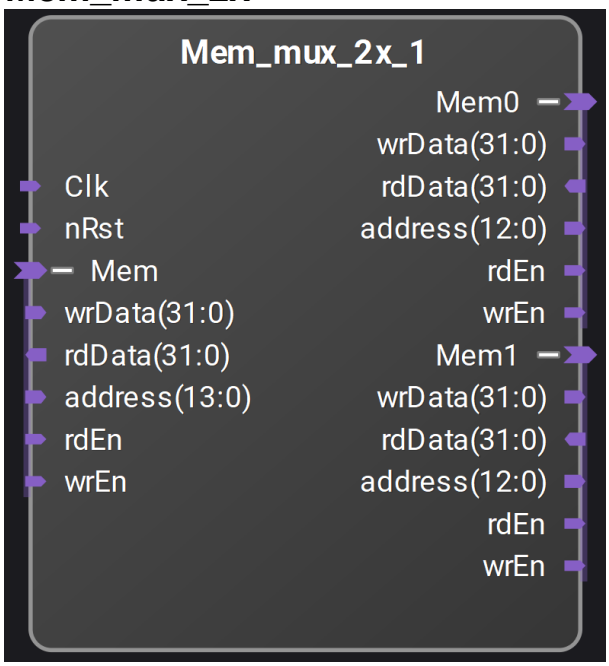
Parameters

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^(Slave Address Width)

Output address space size = Input address space size / 2

MEM0 offset = 0.

MEM1 offset = Output address space size.

Parameters

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^(Slave Address Width)

Output address space size = Input address space size / 4

MEM0 offset = 0.

MEM1 offset = 1*Output address space size.

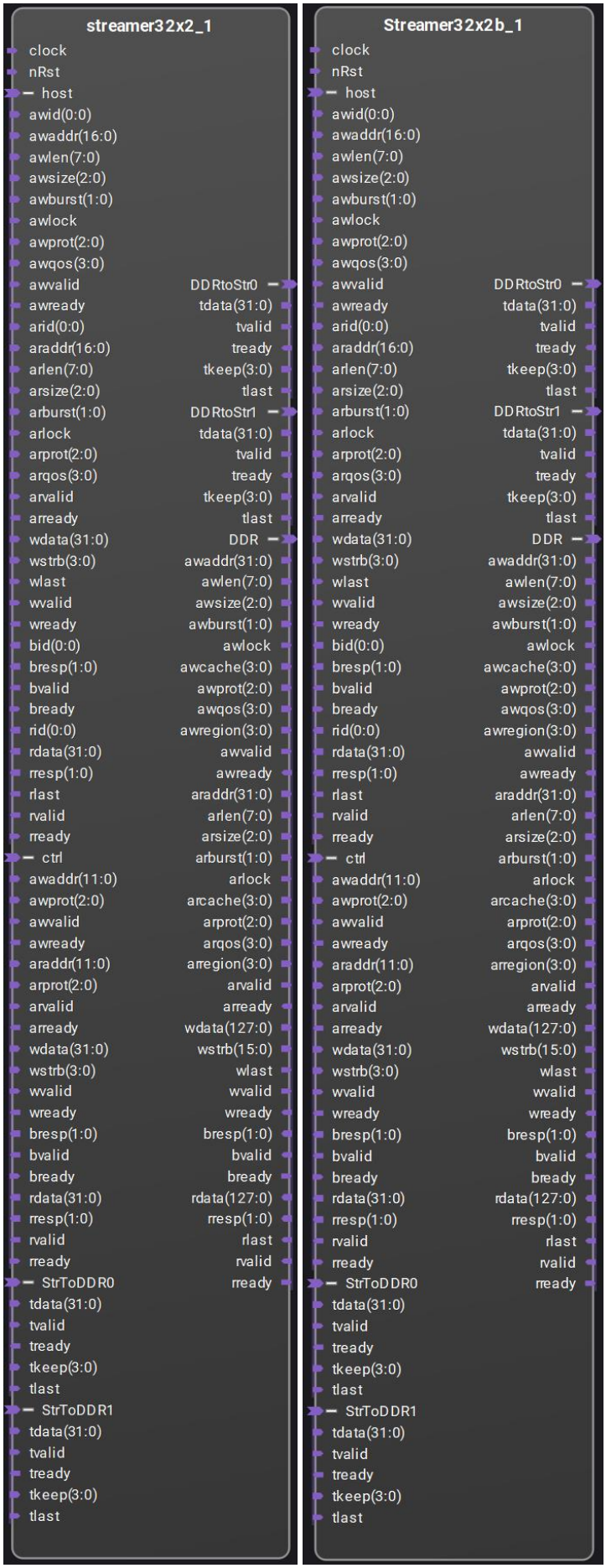
MEM2 offset = 2*Output address space size.

MEM3 offset = 3*Output address space size.

Parameters

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

Streamer blocks (2 channels at 32 bits/channel) – Streamer32x2/Streamer32x2b

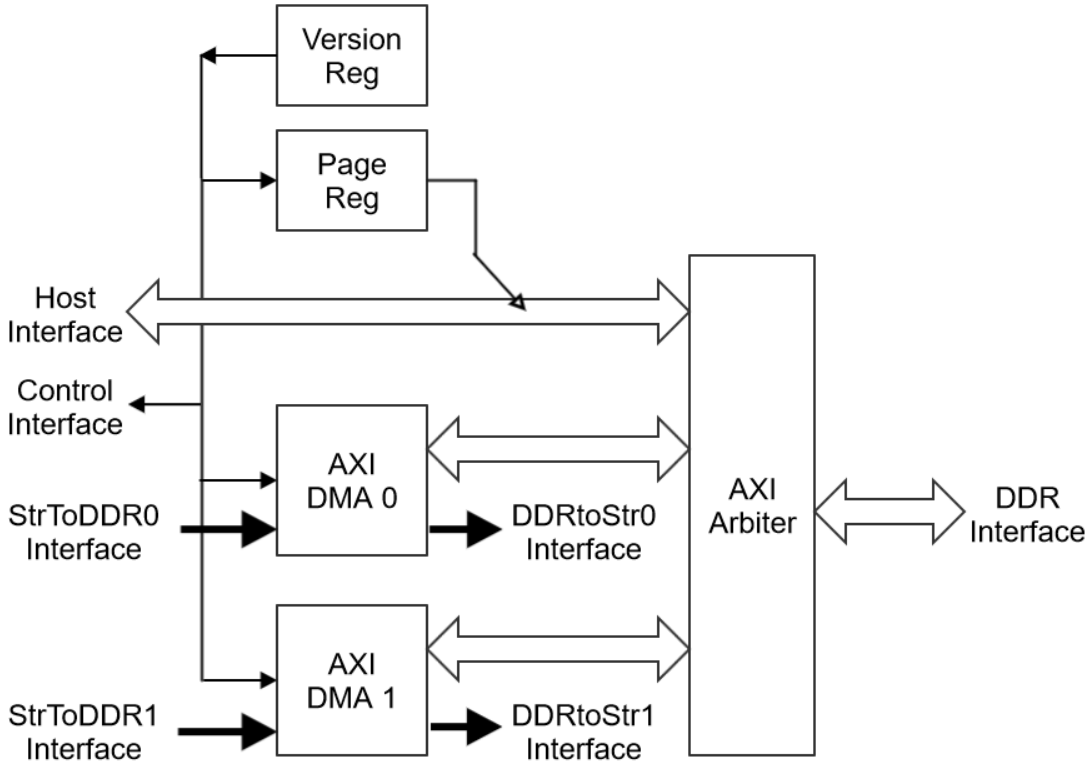

streamer32x2_1
clock
nRst
host
awid(0:0)
awaddr(16:0)
awlen(7:0)
awsize(2:0)
awburst(1:0)
awlock
awprot(2:0)
awqos(3:0)
awvalid
awready
arid(0:0)
araddr(16:0)
arlen(7:0)
arsize(2:0)
arburst(1:0)
arlock
arprot(2:0)
arqos(3:0)
arvalid
arready
wdata(31:0)
wstrb(3:0)
wlast
wvalid
wready
bid(0:0)
bresp(1:0)
bvalid
bready
rid(0:0)
rdata(31:0)
rresp(1:0)
rlast
rvalid
rready
ctrl
awaddr(11:0)
awprot(2:0)
awvalid
awready
araddr(11:0)
arprot(2:0)
arvalid
arready
wdata(31:0)
wstrb(3:0)
wvalid
wready
bresp(1:0)
bvalid
bready
rdata(31:0)
rresp(1:0)
rvalid
rready
StrToDDR0
tdata(31:0)
tvalid
tready
tkeep(3:0)
tlast
StrToDDR1
tdata(31:0)
tvalid
tready
tkeep(3:0)
tlast
DDRtoStr0
tdata(31:0)
tvalid
tready
tkeep(3:0)
tlast
DDRtoStr1
tdata(31:0)
tvalid
tready
tkeep(3:0)
tlast
DDR
awaddr(31:0)
awlen(7:0)
awsize(2:0)
awburst(1:0)
awlock
awcache(3:0)
awprot(2:0)
awqos(3:0)
awregion(3:0)
awvalid
awready
araddr(31:0)
arlen(7:0)
arsize(2:0)
arburst(1:0)
arlock
arcache(3:0)
arprot(2:0)
arqos(3:0)
arregion(3:0)
arvalid
arready
wdata(127:0)
wstrb(15:0)
wlast
wvalid
wready
bresp(1:0)
bvalid
bready
rdata(127:0)
rresp(1:0)
rlast
rvalid
rready
Streamer32x2b_1
clock
nRst
host
awid(0:0)
awaddr(16:0)
awlen(7:0)
awsize(2:0)
awburst(1:0)
awlock
awprot(2:0)
awqos(3:0)
awvalid
awready
arid(0:0)
araddr(16:0)
arlen(7:0)
arsize(2:0)
arburst(1:0)
arlock
arprot(2:0)
arqos(3:0)
arvalid
arready
wdata(31:0)
wstrb(3:0)
wlast
wvalid
wready
bid(0:0)
bresp(1:0)
bvalid
bready
rid(0:0)
rdata(31:0)
rresp(1:0)
rlast
rvalid
rready
ctrl
awaddr(11:0)
awprot(2:0)
awvalid
awready
araddr(11:0)
arprot(2:0)
arvalid
arready
wdata(31:0)
wstrb(3:0)
wvalid
wready
bresp(1:0)
bvalid
bready
rdata(31:0)
rresp(1:0)
rvalid
rready
StrToDDR0
tdata(31:0)
tvalid
tready
tkeep(3:0)
tlast
StrToDDR1
tdata(31:0)
tvalid
tready
tkeep(3:0)
tlast
DDRtoStr0
tdata(31:0)
tvalid
tready
tkeep(3:0)
tlast
DDRtoStr1
tdata(31:0)
tvalid
tready
tkeep(3:0)
tlast
DDR
awaddr(31:0)
awlen(7:0)
awsize(2:0)
awburst(1:0)
awlock
awcache(3:0)
awprot(2:0)
awqos(3:0)
awregion(3:0)
awvalid
awready
araddr(31:0)
arlen(7:0)
arsize(2:0)
arburst(1:0)
arlock
arcache(3:0)
arprot(2:0)
arqos(3:0)
arregion(3:0)
arvalid
arready
wdata(127:0)
wstrb(15:0)
wlast
wvalid
wready
bresp(1:0)
bvalid
bready
rdata(127:0)
rresp(1:0)
rlast
rvalid
rready

NOTE: The Streamer32x2 IP block uses the Vivado 2017.3 AXI DMA IP block. This IP block has a 23 bit transfer length register. This means the largest DMA transfer size allowed is 8 Mbyte. If a larger transfer size is needed, use the Streamer32x2b IP block. The Streamer32x2b IP block uses the Vivado 2018.1 AXI DMA IP block that has a 26 bit transfer length register. This allows DMA transfers up to 64 Mbyte.

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 the RSP 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

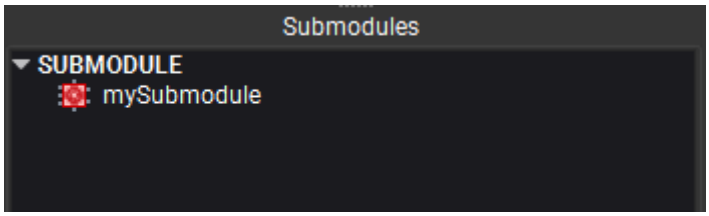
The version register is used to identify the version and configuration of the Streamer32x2/Streamer32x2b IP block.

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

PathWave FPGA Submodule

PathWave FPGA submodules allow you to define your design hierarchically. In addition, you can share submodules in IP repositories.

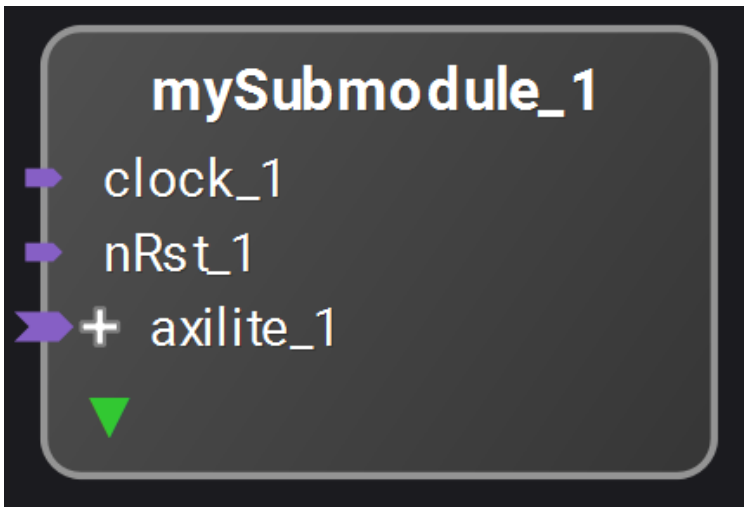
The submodules that can be added to your design are displayed in the Submodule pane.



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.

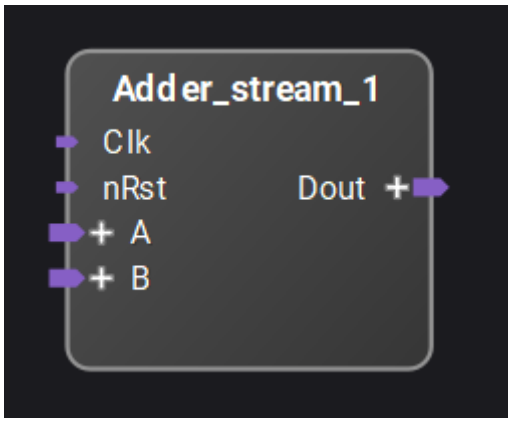
Submodules may also be added to a project by selecting **Project > Add External Block...** and navigating to the desired submodule project file with the `.ksub` filename extension.

Submodules can be visually distinguished from other blocks in the canvas with a small green triangle in the bottom left corner of the block.



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).

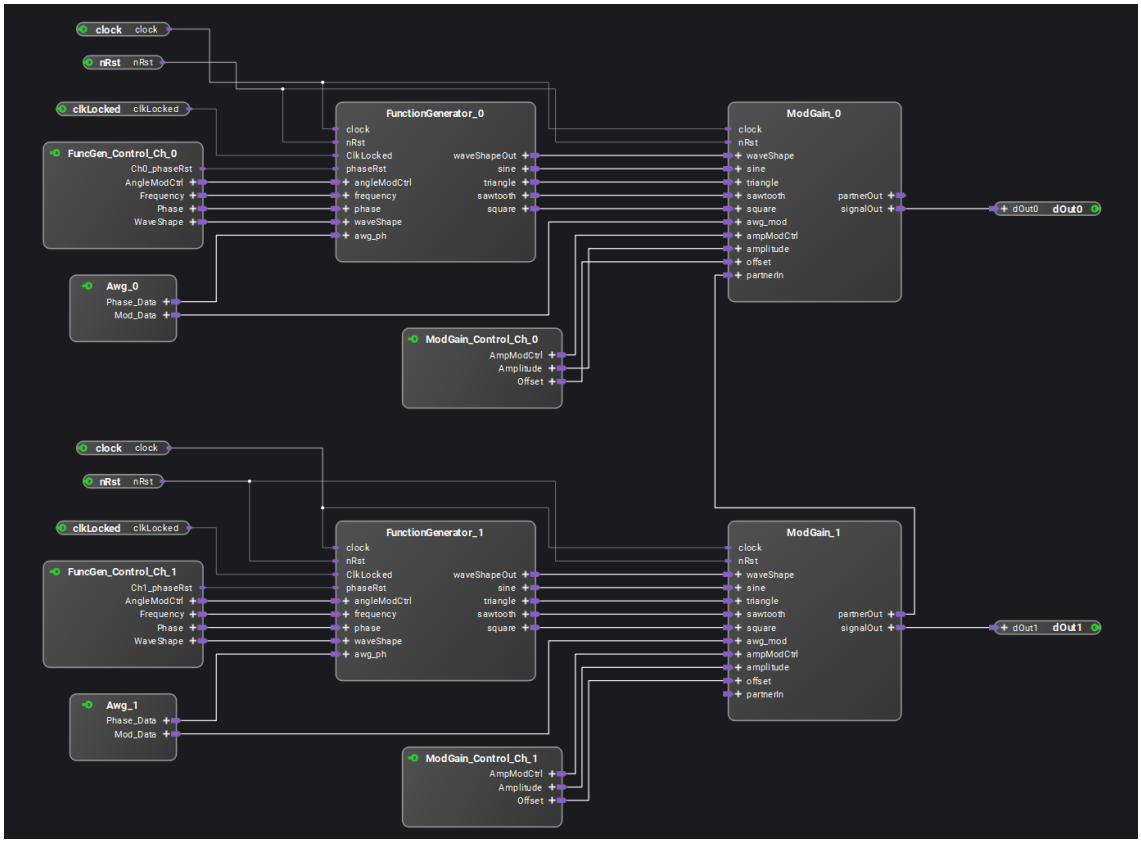
This block has two ports (small connectors), and the other connectors are interfaces (larger connectors). The ports can represent one bit of data or a vector of bits. If the port represents a vector of bits, the size can be identified next to its name.

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.

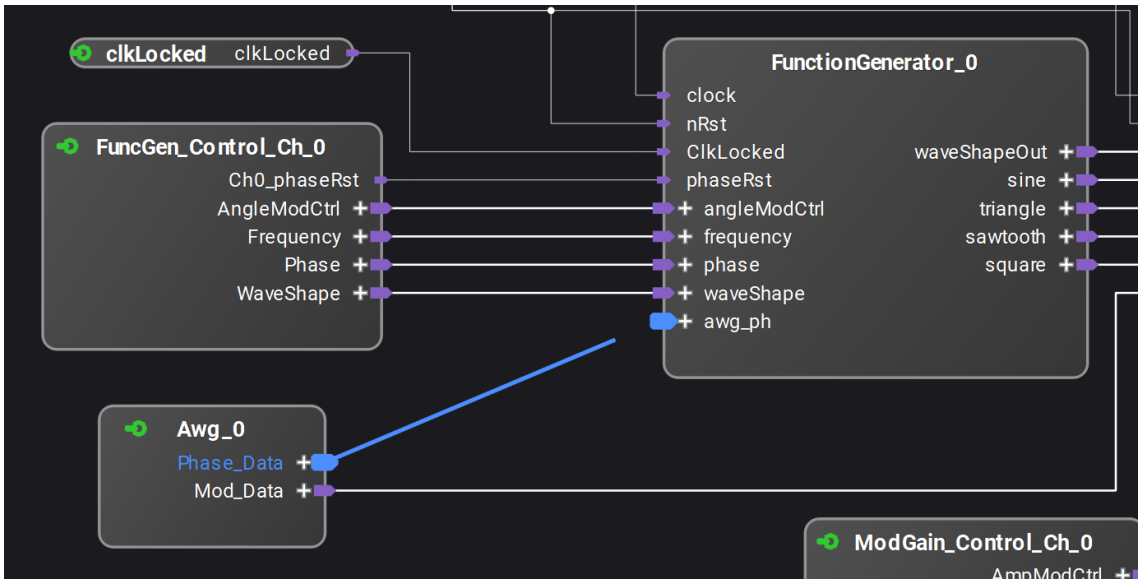
Connecting an Output Port to an Input Port

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.

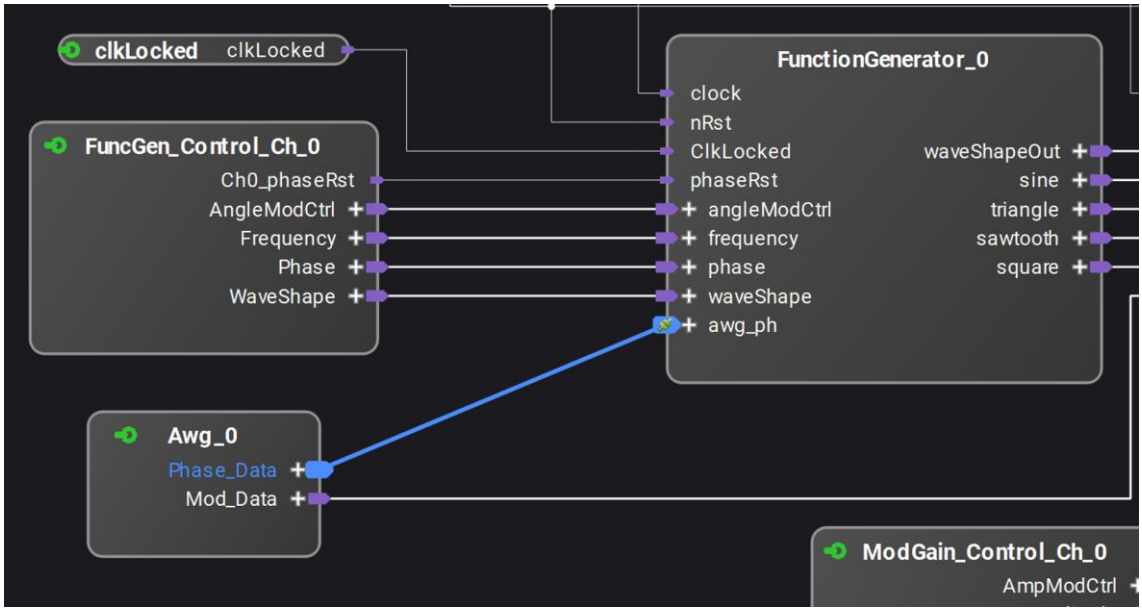


Connections can be created according to connection rules. For more information, refer Connection Rules.

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.



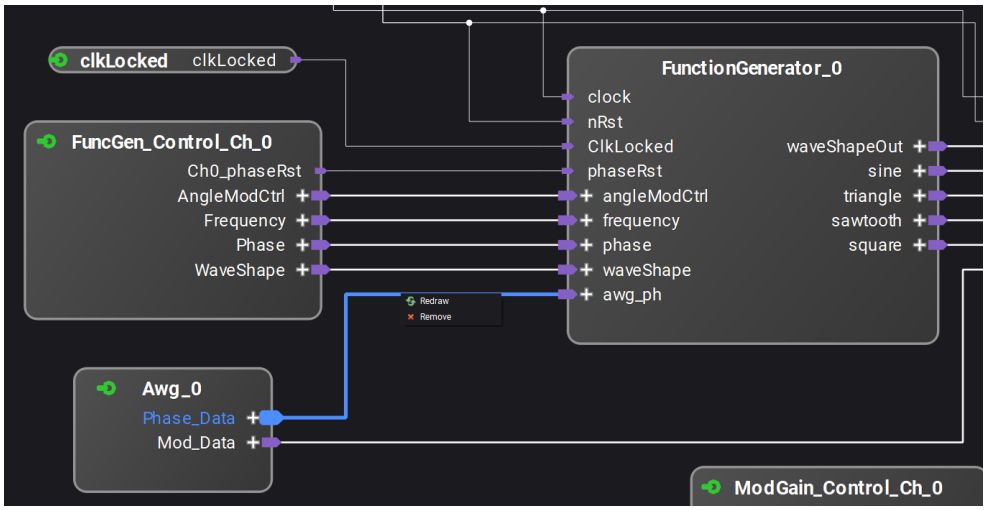
For finishing the connection, the end of the connection line is dragged by the mouse to a compatible target connector. In this case, the mouse icon changes to the green connection icon.



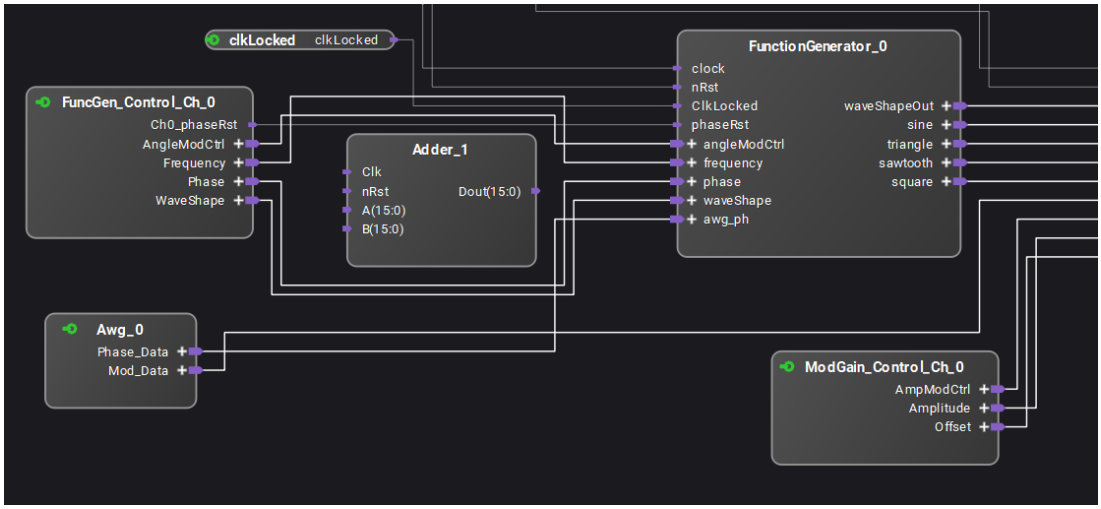
When the mouse button is released, the new connection is created.

Remove and Redraw operations

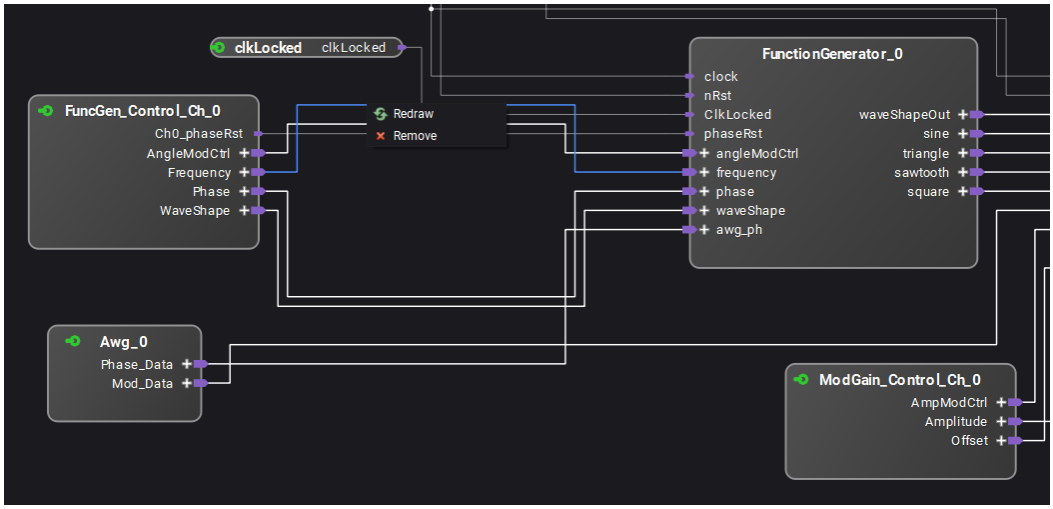
Right-click the line connecting the two ports to see two options: **Remove** and **Redraw**. Remove will delete the connecting line.



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

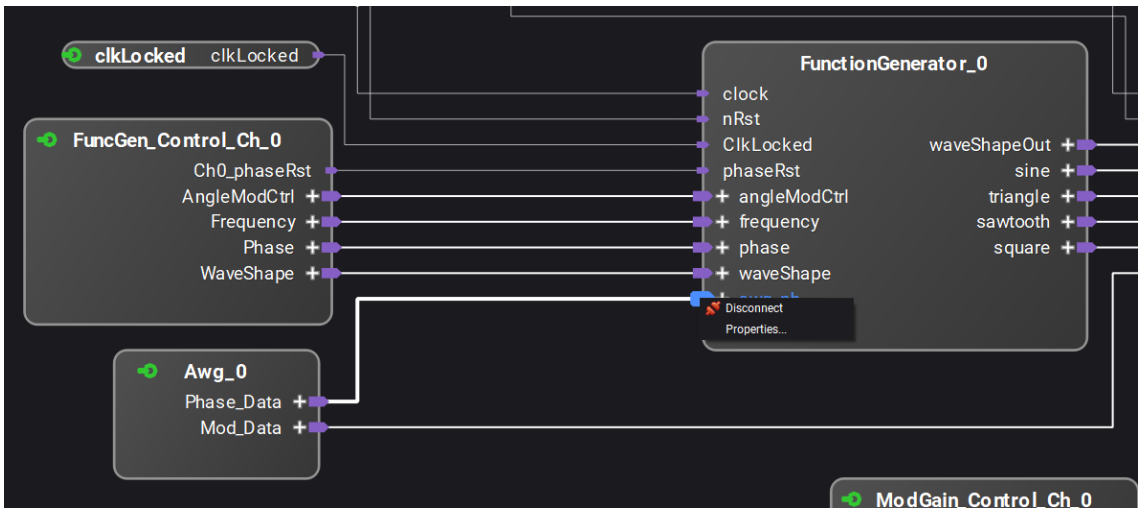


Delete the block that was just added and notice that the connecting line stays unchanged. Right-click the line and select **Redraw**. The line will be straight again.



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:

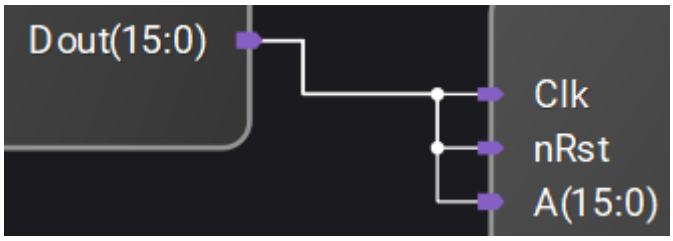
- **Integer**: A integer number, negative numbers set a two's complement format. The range for valid inputs is from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807, or from −(2^63) to 2^63 − 1
- **Hexadecimal**: A hexadecimal number using the characters 0 - F can be entered, followed by an **h**;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

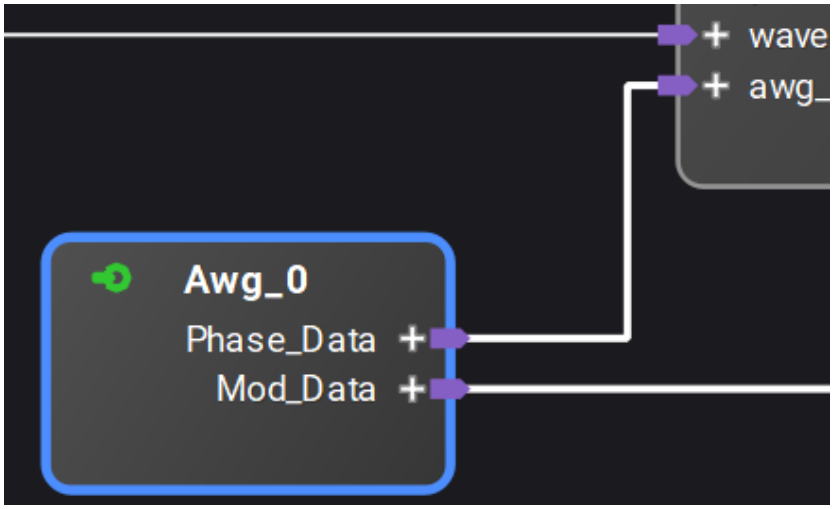
If a narrow output port is connected to a wider input port, the output port connects to the LSBs of the input port. The remaining bits of the input port are set to zero.

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.

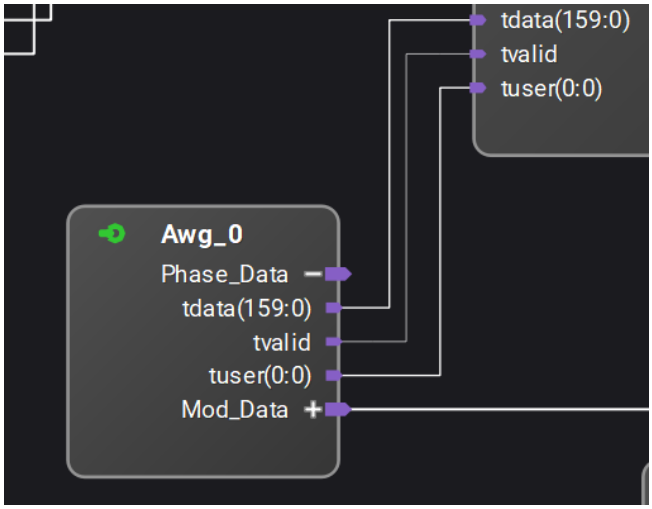
In the second example shown above, both clk and rst will be connected to Dout(0).

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 no 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	awsize	width of data bus in bytes as a power of 2, default assumes a bus width of 32-bits	2
AXI4MM	arsize	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.

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 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*, MY*REG, MyReg* are all considered the same.

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,
```

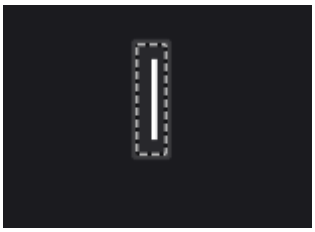
```
package, packed, parameter, pmos, port, posedge, postponed, primitive,
priority, procedure, process, program, property, protected, pull0,
pull1, pulldown, pullup, pulsestyle_ondetect, pulsestyle_onevent,
pure, rand, randc, randcase, randsequence, range, rcmos, real,
realtime, record, ref, reg, register, reject, release, rem, repeat,
report, return, rnmos, 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, tri1, 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
```

Adding and Editing Comments

To add a comment:

1. Position the cursor within the project where the comment is to be inserted.

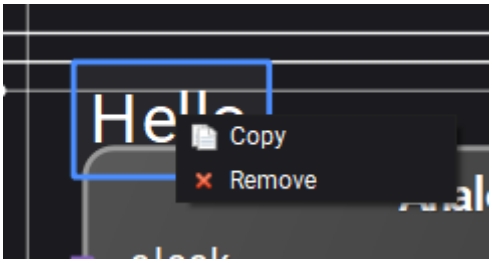
2. Right-click on a blank part of the canvas and select **Insert Comment...** .



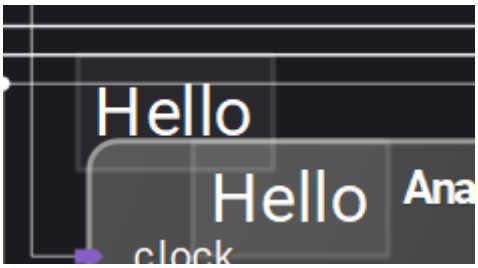
3. Add text to the comment text box.

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.

Naming Collisions

PathWave FPGA is using the concept of VLNV for identifying IP and reporting naming collisions. VLNV stands for Vendor-Library-Name-Version and is a concept introduced by IP-XACT.

- **Two IPs have the same name, but different VLNV.** In this case, user will have to resolve it using one of the workarounds.
- **Two IPs have have the same VLNV, apart from the version field.** In this case, PathWave FPGA will give the user the option to upgrade/downgrade. Note that 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 VLNV, but different contents.** In this case, PathWave FPGA will give the user the option to update to the desired definition. Note that 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 VLNV and contents, but are stored in different location.** In this case, PathWave FPGA will use the last loaded location as the correct location of the IP.
- **Two IPs have the same name, but they do not have a VLNV.** In this case, user will have to resolve it using one of the workarounds.
- **Two IPs have the same name, but are coming from different import method.** In this case, user will have to resolve it using one of the workarounds.

- **An IP is using a name of a reserved word.** In this case, a possible workaround is to create a wrapper for that IP which will have a non-colliding name

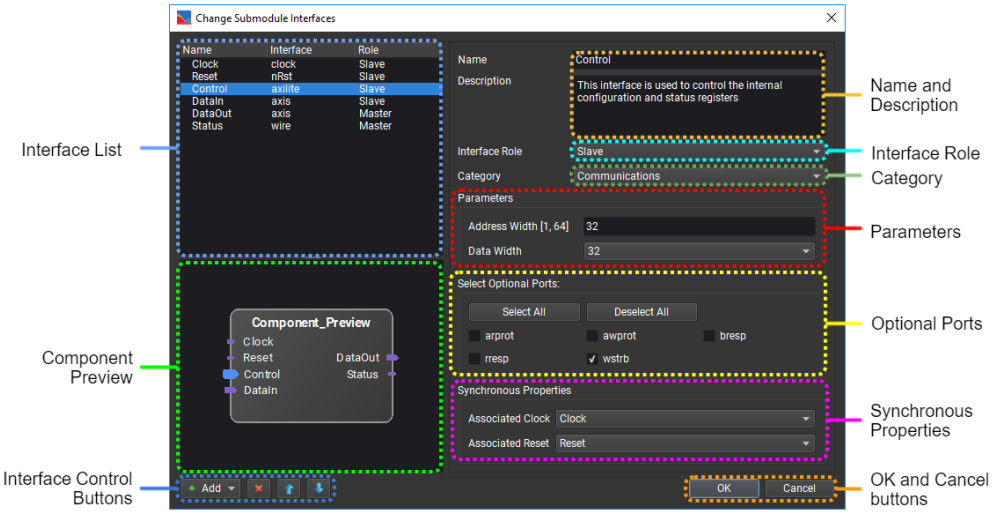
Workarounds

When a name collision is detected, the user will have to take action and resolve it.

- **Rename the IP to a non-conflicting name.** This is simplest and fastest solution. However, if the user is not the owner of the IP, it might not be feasible. In this case, the user has to follow the second workaround
- **Load only the IPs that are necessary for the project.** This is by definition possible only if the conflicting IPs are not needed at the same time in the design. 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, if this is not possible, move the conflicting IP definition file (IP-XACT file) outside of the IP repository location or any sub-directory.
- **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.

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 **Submodule I/O** 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 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 **Submodule I/O** 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 Submodule I/O 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 Submodule I/O 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 Submodule I/O 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 Submodule I/O 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 Submodule I/O 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 Submodule I/O blocks. But you could not replace an aximm named 'host' with a PC_MEM named 'host'.

Adding an Interface

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

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 is 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*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.

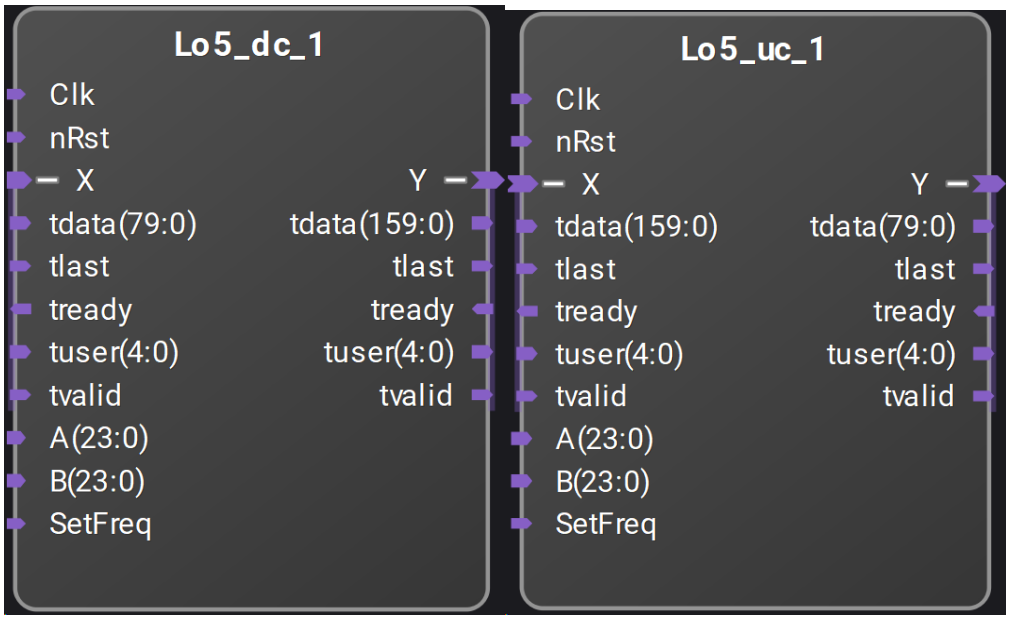
These IP blocks support the optional AXI-streaming signals TUSER and TLAST in addition to the main data bus TDATA. The connection or use of TUSER or TLAST is not required. These signals may be ignored if they are not being used. The TLAST signal indicates the last sample in a data block. It is passed through the IP block unchanged along with the data. TUSER bits can be used to associate some data with some particular 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.

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.

Detail IP Block Descriptions

Local Oscillator

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.

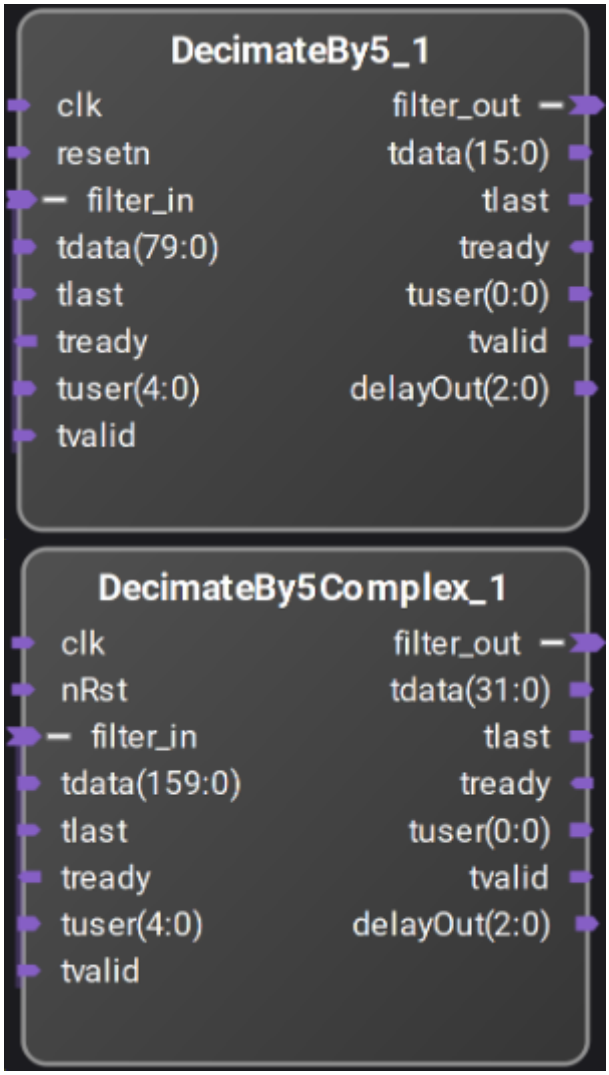
The two input vectors A and B determine the frequency of the local oscillator. If the sample rate is f_s, then the LO frequency is $f_s * (A+B/5^{10})/(5*2^{22})$. Note that f_s is the sample rate of the data, not the clock rate of the FPGA which is 1/5 of the sample rate. The LO is designed so that 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.

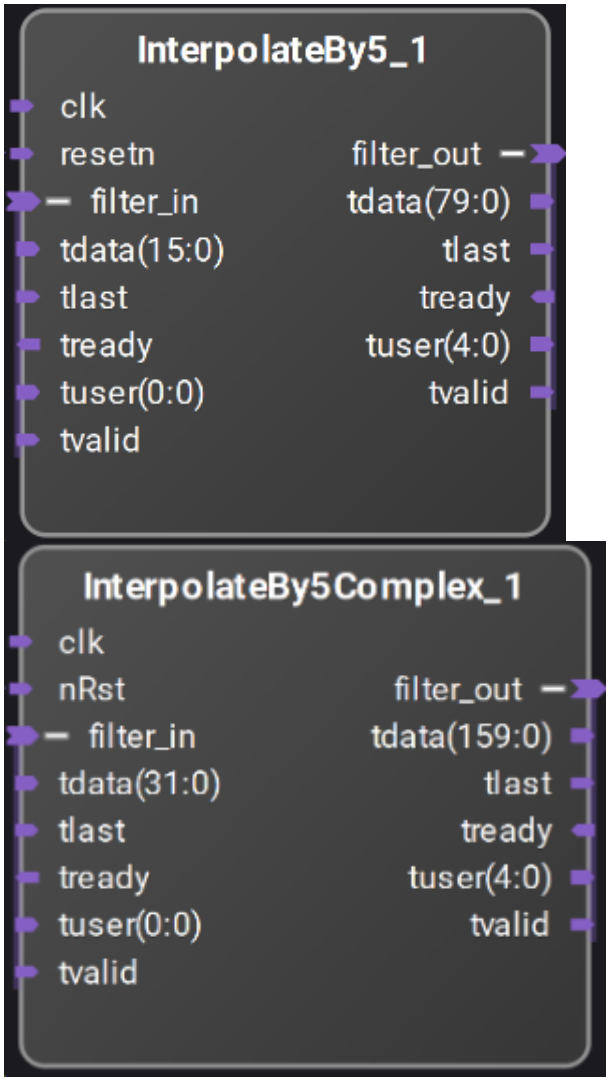
Lo5_dc will multiply the real input stream X with the complex local oscillator and generate the complex output stream Y. This block multiplies the real input by exp(j2π f t).

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 exp(-j2π f 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 √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 ±full scale. Note: this will cause distortion so it is recommended that the magnitude of the complex input be kept less than full scale.

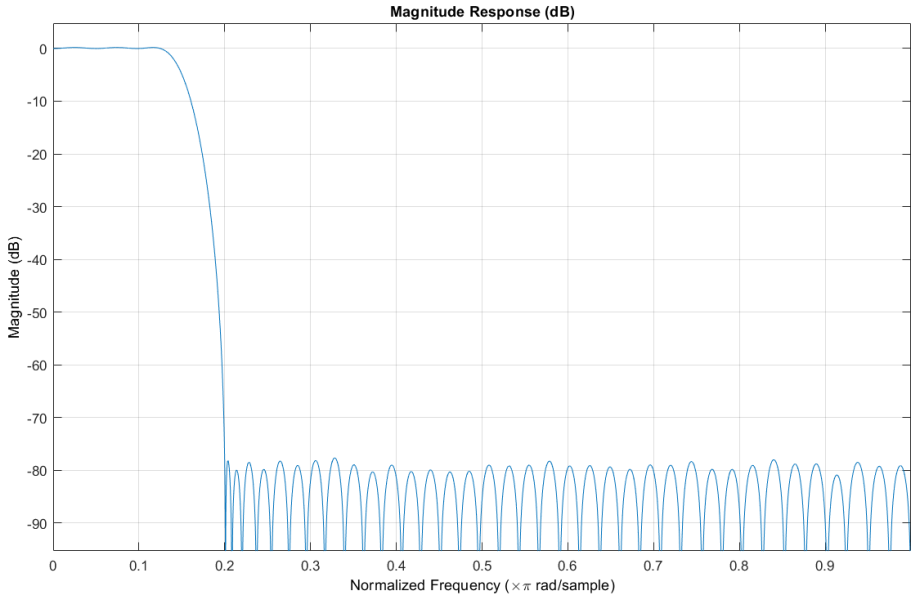
DecimateBy5/InterpolateBy5

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).


DecimateBy5_1
clk
resetn
filter_in
tdata(79:0)
tlast
tready
tuser(4:0)
tvalid
filter_out
tdata(15:0)
tlast
tready
tuser(0:0)
tvalid
delayOut(2:0)
DecimateBy5Complex_1
clk
nRst
filter_in
tdata(159:0)
tlast
tready
tuser(4:0)
tvalid
filter_out
tdata(31:0)
tlast
tready
tuser(0:0)
tvalid
delayOut(2:0)



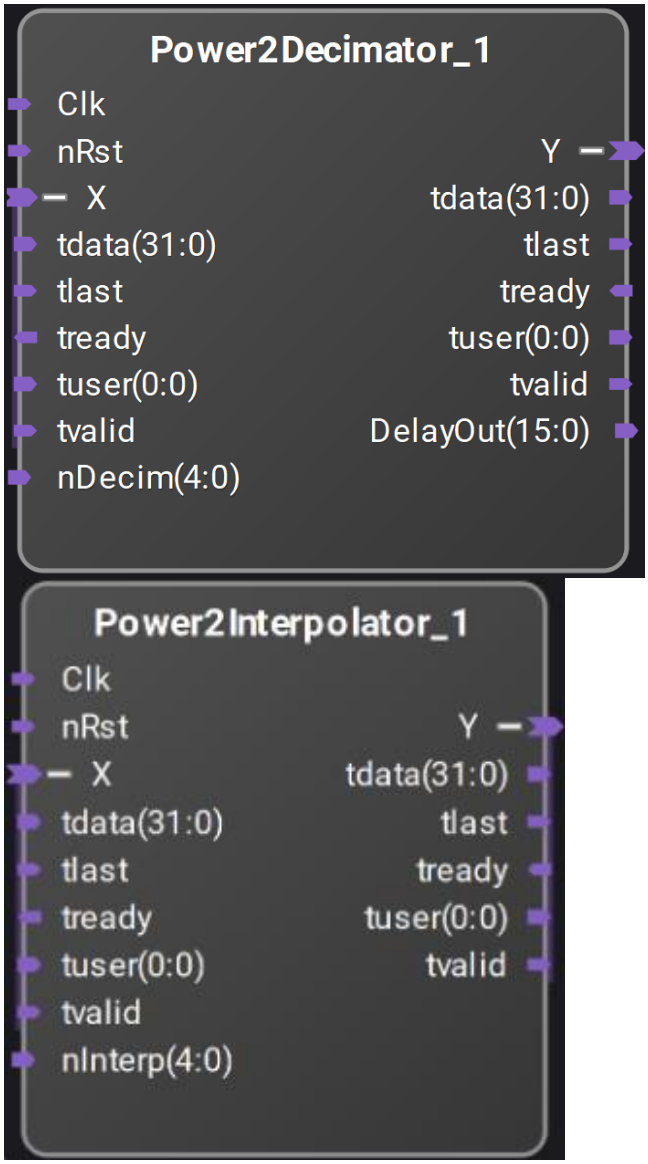
The DecimateBy5 block first low pass filters the input to protect against aliasing and then decimates by 5 (discarding 4 of every 5 output samples). The InterpolateBy5 block first interpolates by 5 by inserting 4 zero samples between each input sample and then low pass filtering to protect against aliasing. Both IP blocks use the same filter which has the frequency response:



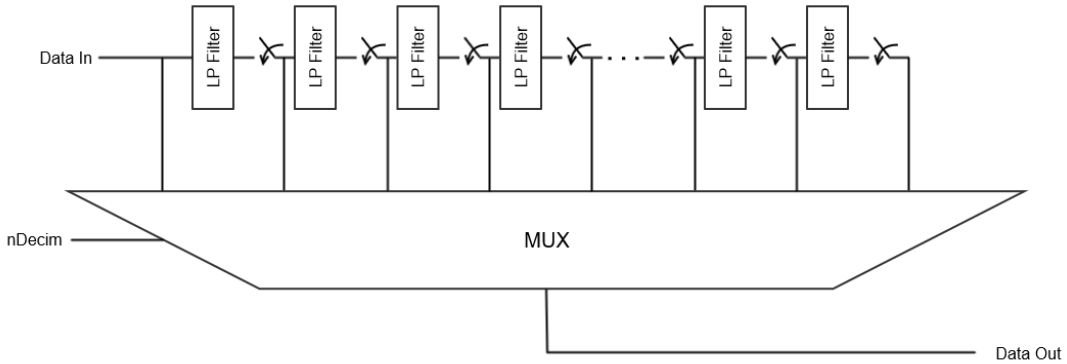
Note that the x-axis is in terms of the normalized frequency where 1 means $f_s/2$. The passband extends up to 0.125 with the stopband starting at 0.2. For example, the M3102 digitizer has a sample rate of 500 Ms/s. Thus $f_s/2$ is 250 MHz and the passband is +/- 31.25 MHz with the stopband above 50 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) complex data, 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).

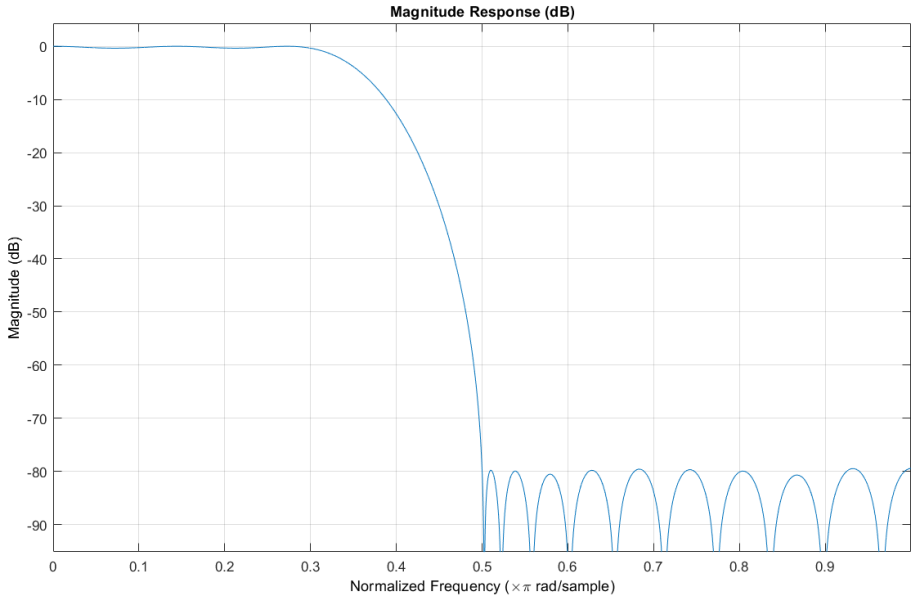


Conceptually, the Power2Decimator can be thought of as a set of 16 cascaded decimate by 2 stages (the internal design uses a more efficient architecture). Each stage first low pass filters its input and then decimates by two. A MUX controlled by nDecim selects the output of one of these filters.



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 use the same filter for each stage. This filter has the frequency response shown:



Note that the x-axis is in terms of the normalized frequency where 1 means $f_s/2$ where f_s is the higher sample rate for that stage. For decimators, this is the input sample rate while for interpolators, this is the output sample rate. The passband is +/- 0.15 f_s which is 60% of the output Nyquist rate, while the stopband starts at $f_s/4$. 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. 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.

The bit width of each data sample as well as the width of the TUSER signal can be modified, if needed, 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. The Tuser size parameter denotes how many TUSER bits are associated with each (complex) sample.

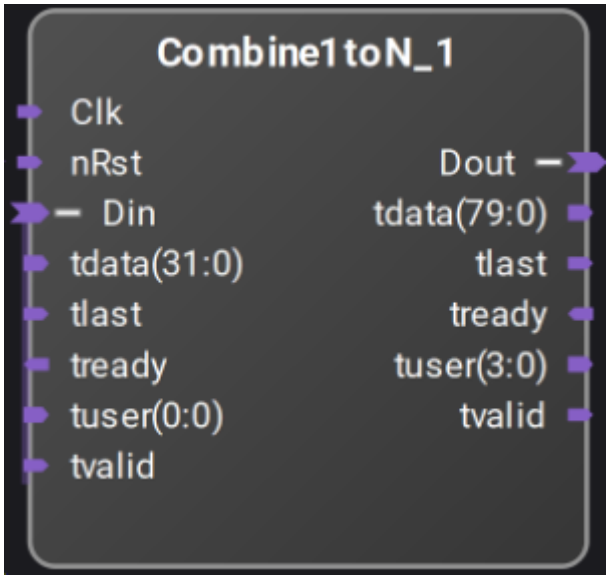
The TUSER and TLAST bits are passed through the decimation stages along with the data. Due to the filter response, there is no one output sample that corresponds to each input sample. A input consisting of an impulse will result in a broad output consisting of the impulse response of the filter. Thus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

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.

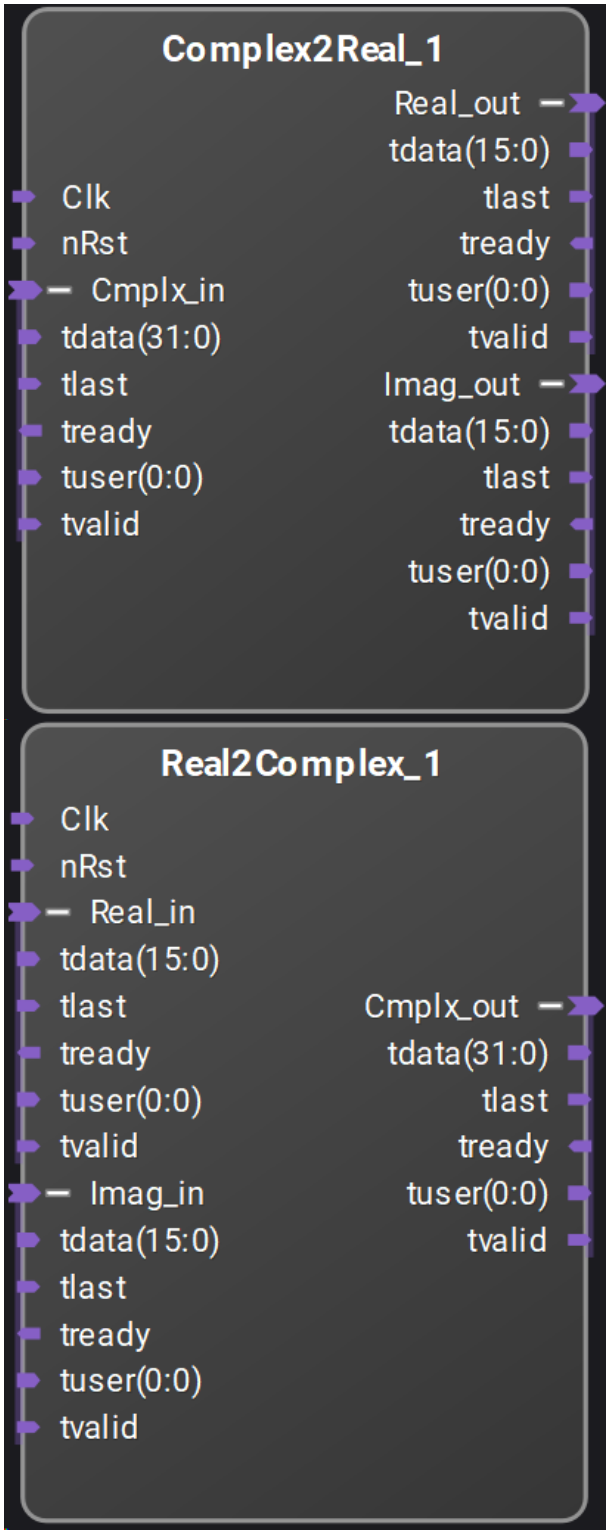
Logically in this case, R0 and I0 are parts of the same (complex) sample. Hence they share the same Din_tuser bit(s). However, some samples, such as R2/I2 are output in different bus cycles. The tuser bits for the R2/I2 input are output for both output bus cycles where R2 or I2 are output. So in this 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.

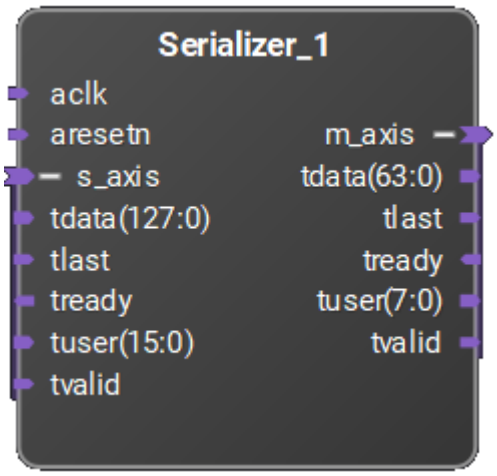


In order to know how to correctly interleave the complex data, these blocks need to know the size of the real data sample and any supersample value. The above pictures show a "Tdata size" of 16 and a "Supersample" of 1 (no supersampling). This means that the Real and Imaginary tdata busses are 16 bits wide, and the Cmplx tdata bus is twice this or 32 bits wide.

Serializer

The Serializer can be used to downsize an AXI4-Stream. The modulus parameter is the AXI4-Stream downsizing factor. That is, the slave interface TDATA and TUSER signals are modulus times larger than the master interface TDATA and TUSER signals. For each input transfer there

are modulus output transfers during which the slave TREADY signal is de-asserted for at least (modulus - 1) clock cycles. AXI4-Stream Master interface TDATA and TUSER elements are packed together on the AXI4-Stream Slave interface such that earlier in time elements occupy the lesser significant position and those elements are then serialized onto the AXI4-Stream Master interface. Any TLAST assertion transferred on the Slave interface occurs only on the last corresponding transfer on the Master interface.



The design can be parameterized for custom modulus, number of Master Interface TDATA bytes, and TUSER bits per byte. The example block above is parameterized for Master Interface TDATA width of 8 bytes, 1 TUSER bit per byte, and a modulus (downsizing factor) of 2. Thus TDATA and TUSER are 2x larger for the AXI4-Stream Slave interface than for the AXI4-Stream Master Interface.

Design Examples

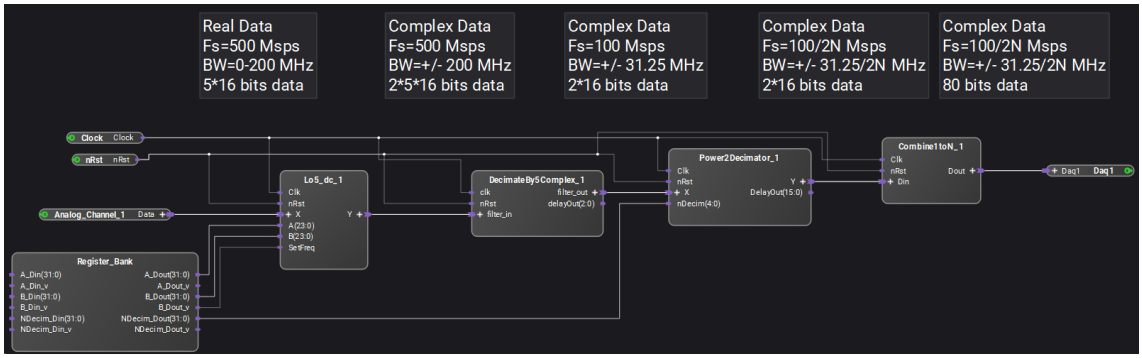
To see how these IP blocks can be used to build up and down converters, consider two example designs, one for a digital down converter, and one for a digital up converter. These examples are built in a M3302, 500 Msps Combination AWG and Digitizer.

Digital Down Converter (DDC)

For a digitizer to analyzer signals with narrower bandwidth than the full digitizer bandwith, 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 Lo5_dc (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 Lo5_dc block is designed 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.

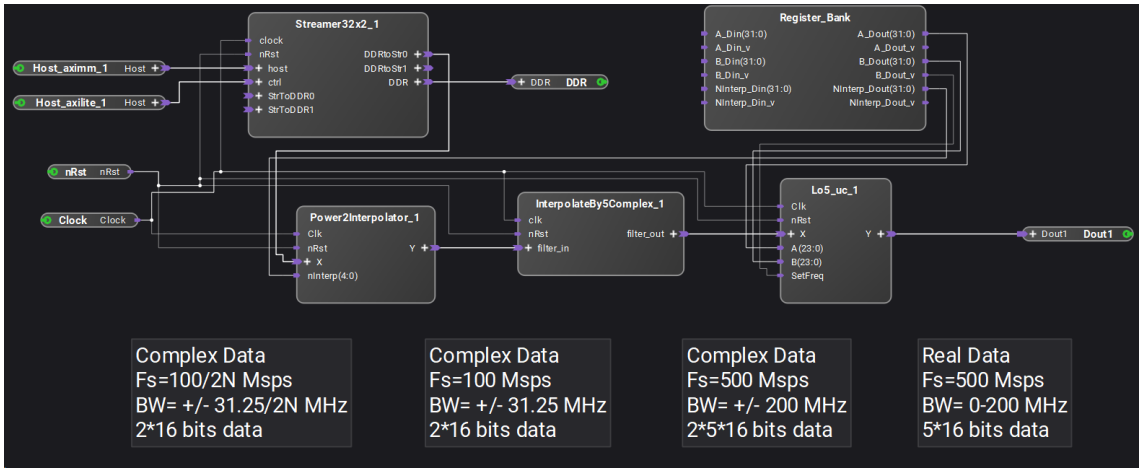
This results in a data record in memory consisting of complex pairs, each consisting of the real part of a sample and the imaginary part of the sample.

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 then 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 use 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 Lo5_uc (Local Oscillator Up Converter) 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 Lo5_uc block is designed 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 increasees 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.

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, if the width range is defined as a 'to' range, the right boundary cannot be larger than 64. There is no such limitation for 'downto' ranges.
- 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`."

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

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:

- Input/output port sizes may only contain constant values. They may not use parameters or expressions, such as "`input [WIDTH-1:0] x`".
- When input/output port declarations come after the port list (not ANSI-style/Verilog-2001), all port declarations must appear before any other declarations, such as `parameter`, `reg`, or `signal`.
- Definition of port attributes is not supported, such as "`(* attribute definition *) input portName,`".
- When the module declaration contains a parameter list, there must be a space between the module name and the '`#`' for the parameter list.
- Parameters used in a module declaration may not be defined using parenthesis, unless such a parameter is the last item in the parameter list. (for example, `parameter myParam = (6),`)
- Port definitions in a module declaration may not be conditionally included using `` `ifdef `` / `` `endif `` statements
- A module name must include one or more port definitions.

- To import Verilog source files into PathWave FPGA for use within a design, a module declaration format should conform with of one of the following examples:

```
module foo  #( parameter myParam1 = 14, myParam2 = 32) ( input wire
clk, output reg [31:0] d_out); endmodule
```

or:

```
module foo  (clk, d_out); input wire clk; output reg [31:0] d_out;
endmodule
```

- When Kactus2 is used for creating IP-XACT for a Verilog file, avoid comments of the form "`// input name;`" or "`// output name;`" in the Verilog source file as these will cause the Verilog parser to not work properly.

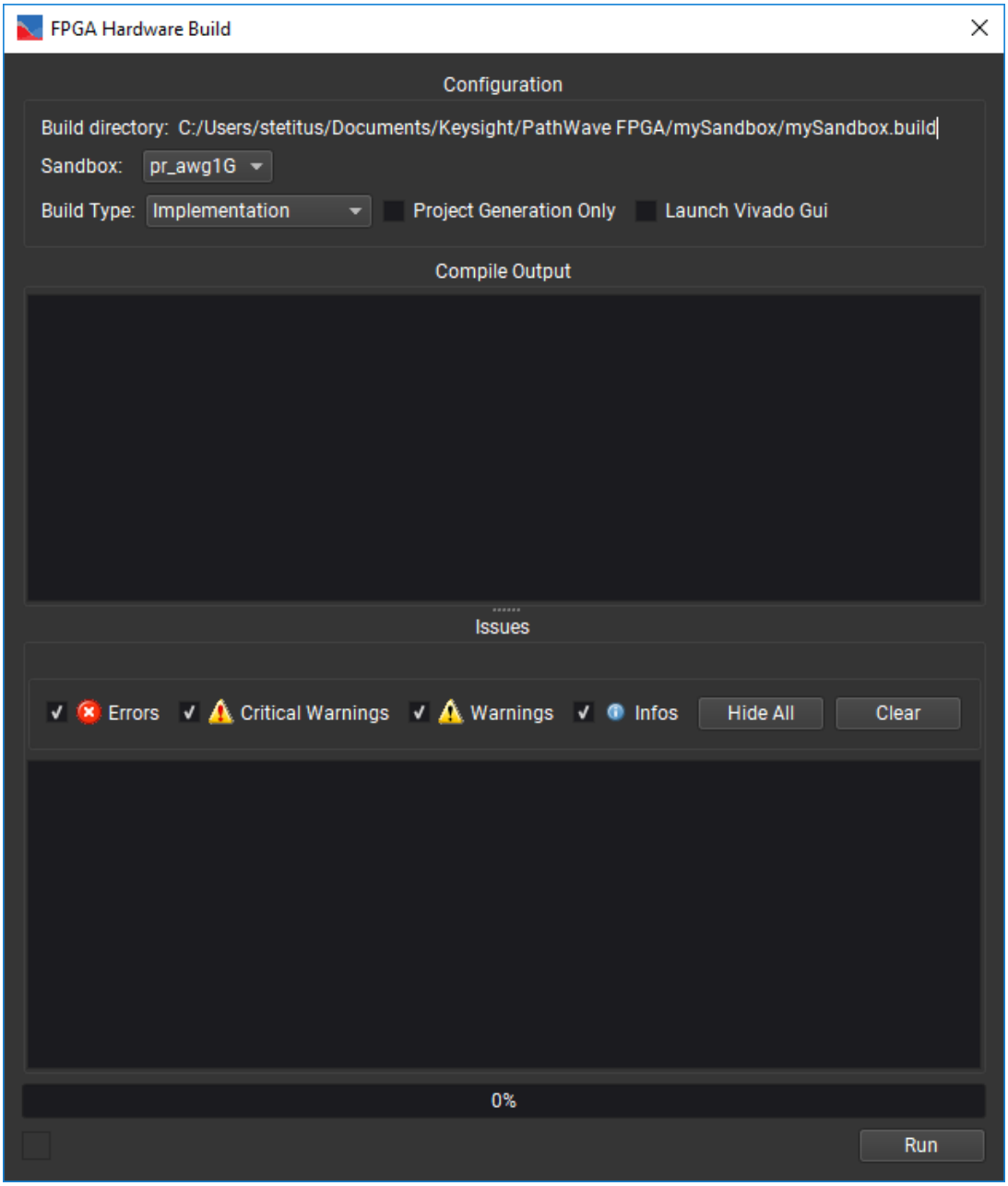
Generating the Bit File

Synthesizing and Implementing your Design inside of PathWave FPGA

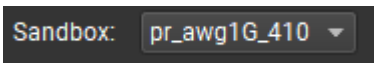
After creating your new hardware project and adding your FPGA logic, you are ready to generate the bit file that implements your design.

To build the bitfile based on your design, complete the following steps:

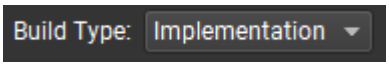
1. Select **Module**> **Generate Bit File...** or click the toolbar icon with tooltip "Generate Bit File...". The FPGA Hardware Build dialog will appear.



2. Choose the sandbox that you want to target for this build.



3. Choose the **Implementation** build type. This will build the complete project, including the bit file.



4. Click **Run** to start the build.

Different FPGA Build options

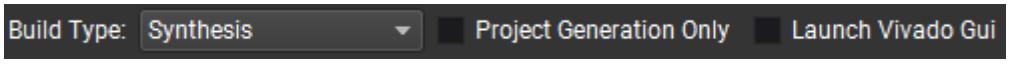
The FPGA Hardware Build has two different build options that affect what options are displayed by the build dialog. The version of the BSP affects what options are available. The same basic build types are available between each, but the newer BSPs add additional usability features.

Basic Build Types (common between all BSPs)

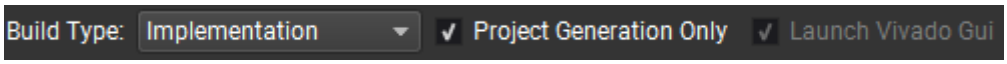
- Synthesis: Builds what is present in the sandbox only.
- Implementation: Builds what is present in the sandbox and places it into the static region of the selected BSP and runs to bit generation.
- Implementation from DCP: Takes a provided DCP and places it into the static region of the selected BSP and runs to bit generation.

Usability features (newer BSPs)

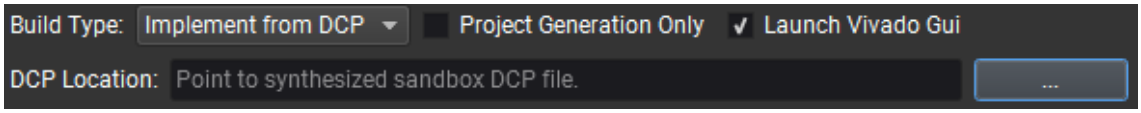
- Two new options are available, launch the Vivado GUI to monitor the build, and only run project generation on a design.



- When project generation is selected, the Vivado GUI will always be launched.



- The GUI can be selected to launch regardless of project generation

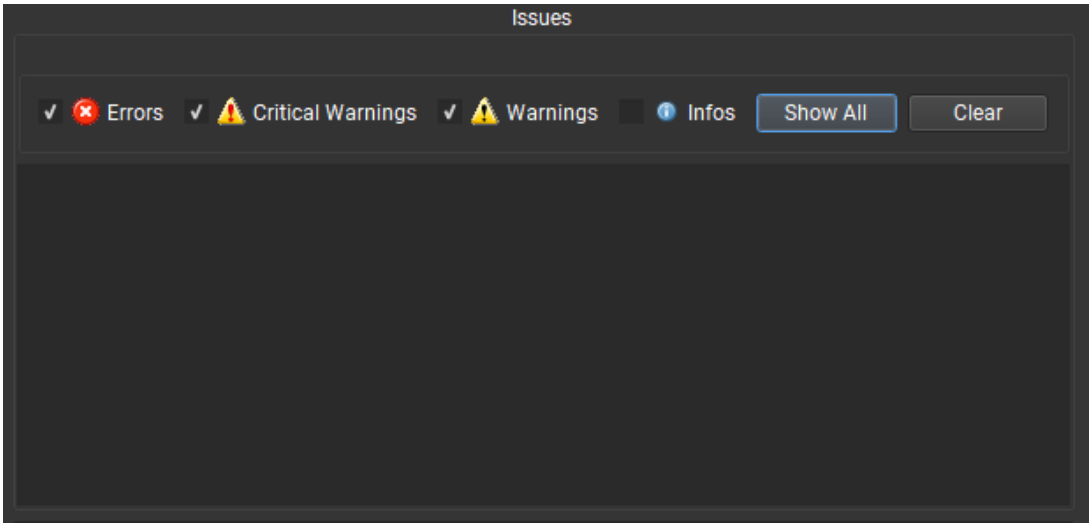


Monitoring the Build

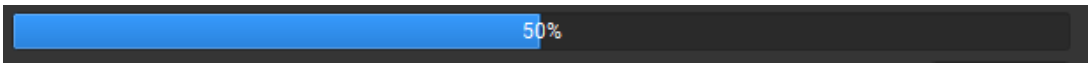
The FPGA Hardware Build dialog contains several panes to monitor the progress of the build:

- The Compile Output pane displays all build output.

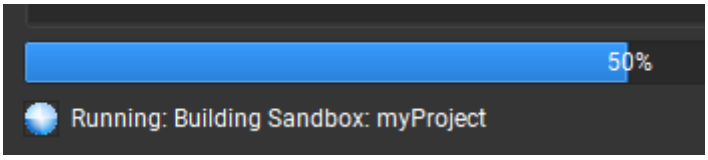
- The Issues pane shows filtered build output. You can set the filters by checking the boxes (Errors, Critical Warnings. etc.) at the top of the Issues pane. The filters can be set at any time while the build is running or after it is complete.



- The progress bar shows the approximate progress of the build.



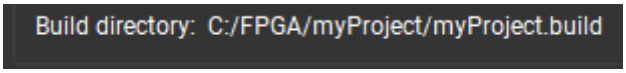
- The status bar at the bottom left shows what step of the build is being performed. When the build is finished, the build status will be displayed.



- 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.



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 FGPA 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.

Verifying the Bit File

After you generate your FPGA bit file, you are ready to deploy and verify it on the FPGA. The Board Support Package for your FPGA supplies the *run-time support package* (RSP) C API that provides programmatic control of the FPGA. Using the RSP you can create a C application to verify your bit file. Note, you will need Visual Studio C++ and CMake, please see the System Requirements for more details.

The RSP documentation and example program are provided in a separate Help area available from the **Help > Programmer's Guide** menu.

After you have verified the bit file, you are ready to deploy it in a measurement application. Please consult your instrument driver manual to learn how to integrate the bit file into your custom measurement application.

Advanced Features

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.

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-XACT is an IEEE 1685-2014 standard which defines a set of xml schemas which allow the description of IP. For more information on IP-XACT, please consult the manual IEEE 1685-2014 standard. As explained earlier, PathWave FPGA is using this file format to improve the usability of IP within the software. PathWave FPGA is supporting a subset of the elements defined in the IP-XACT standard along with custom defined elements. A detailed description of which elements are supported and how they should be used is provided in section IP-XACT file composition.

Since the process of creating an IP-XACT file can be tedious and error-prone, PathWave FPGA is coming along with a software tool, IP Packager, that allow IP developers to quickly and effectively create IP-XACT files for their IP. A detailed description of the usage of this tool is explained in section IP Packager.

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.

IP-XACT file composition

Definition of the IP-XACT file

For an IP-XACT file to correctly describe an IP, the guidelines described below should be followed:

- the IP-XACT file should follow the IEEE 1685-2014 standard
- the root element should be an `ipxact:component`
- the vendor name (element `ipxact:vendor`, first child of `ipxact:component`) should be equal to the internet domain of the vendor of the IP (for example, for Keysight Technologies this will be `keysight.com`)
- the name of the library (element `ipxact:library`, first child of `ipxact:component`) will be the name of the library the IP belongs to. This name is also used inside PathWave FPGA for categorizing the IPs

- the name (element `ipxact:name`, first child of `ipxact:component`) should be the same as the name of the IP ("module name" in Verilog, SystemVerilog and SystemC, or "entity name" in VHDL)
- if the IP uses Keysight Standard Interfaces, these should be described using `ipxact:busInterface` elements
- the mappings between logical ports of the `busInterface`s to the physical ports of the IP should be one-to-one. This means that one physical port maps completely (same width, direction) and only to one logical port
- the files that are necessary for an IP to be included in a build process (synthesis, implementation, bit generation) should be defined inside an `ipxact:fileset` component, named "synthesis".

A detailed description of all the elements that are required by PathWave FPGA in order to identify correctly an IP is given in the following table. For more information on the various elements that are supported by IP-XACT, please consult the manual IEEE 1685-2014 standard.

Element	Parent Element	Content
ipxact:component	<root>	This is the root element of the XML file
ipxact:vendor	ipxact:component	Vendor's name. Should be equal to the internet domain of the vendor of the IP (e.g. keysight.com)
ipxact:library	ipxact:component	The name of the library the IP belongs to
ipxact:name	ipxact:component	The name of the IP. Should be the same as the name of the IP in the source file (i.e. *module name* in Verilog, SystemVerilog and SystemC, or *entity name* in VHDL)
ipxact:version	ipxact:component	The version number of the IP.
ipxact:busInterfaces	ipxact:component	Contains a list of ipxact:busInterface elements
ipxact:busInterface	ipxact:busInterfaces	Contains information about a used Keysight Standard Interface
ipxact:name	ipxact:busInterface	The name of the Interface that is used in this IP
ipxact:busType	ipxact:busInterface	The type of the Interface that is used in this IP. This essentially is the VLNV of the Keysight Standard Interface to be used. This should match one of the bus definitions (IP-XACT files with <ipxact:busDefinition> as the root element) defined by PathWave FPGA. See Keysight Standard Interfaces for more information
ipxact:abstractionTypes	ipxact:busInterface	Contains a list of ipxact:abstractionType elements. PathWave FPGA will only support one, the first
ipxact:abstractionType	ipxact:abstractionTypes	Contains information about a used Keysight Standard Interface and the mapping to the physical ports
ipxact:abstractionRef	ipxact:abstractionType	The type of the Interface definition that is used in this IP. This essentially is t he VLNV

Element	Parent Element	Content
		of the definition of the Keysight Standard Interface to be used. This should match one of the abstraction definitions (IP-XACT files with <ipxact:abstractionDefinition> as the root element) defined by PathWave FPGA. See Keysight Standard Interfaces for more information
ipxact:portMaps	ipxact:abstractionType	Contains a list of ipxact:portMap elements
ipxact:portMap	ipxact:portMaps	Contains information about a specific port mapping
ipxact:logicalPort	ipxact:portMap	Contains information about the logical port (port defined in the abstractionDefinition of the enclosing abstractiontype) that participates in the port mapping
ipxact:name	ipxact:logicalPort	The name of the logical port (As this is defined in the abstractionDefinition for the selected Interface Type)
ipxact:physicalPort	ipxact:portMap	Contains information about the physical port (port of the IP) that participates in the port mapping
ipxact:name	ipxact:physicalPort	The name of the physical port (As this is defined in the ipxact:ports section in the same file)
ipxact:model	ipxact:component	Contains information about the modeling of the IP
ipxact:ports	ipxact:model	Contains a list of ipxact:port elements, which represent the physical ports of the IP
ipxact:port	ipxact:ports	Contains information about a specific physical port
ipxact:name	ipxact:port	The name of the physical port. This should match the name defined in the source HDL file
ipxact:wire	ipxact:port	Contains information about the physical representation of a physical port
ipxact:direction	ipxact:wire	Specifies the direction of this port: in for input ports, out for output ports
ipxact:vectors	ipxact:wire	Contains a list of ipxact:vector elements. PathWave FPGA will only support one, the first
ipxact:vector	ipxact:vectors	Specifies the dimensions for a non-scalar port
ipxact:left	ipxact:vector	Specifies the left range for the bit slice used to map a port vector to the bus interface
ipxact:right	ipxact:vector	Specifies the right range for the bit slice used to map a port vector to the bus interface

Element	Parent Element	Content
ipxact:fileSets	ipxact:component	Contains a list of ipxact:fileSet elements
ipxact:fileSet	ipxact:fileSets	Contains information about a specific set of files. Can contain one or multiple ipxact:file elements
ipxact:name	ipxact:fileSet	The name for this set of files.
ipxact:file	ipxact:fileSet	Contains information about a specific file
ipxact:name	ipxact:file	The path to the file. This should be relative to the path of the current IP-XACT document
ipxact:fileType	ipxact:file	Describes the type of file. PathWave FPGA understands one of the following names: • vhdlSource: It is a VHDL source file • verilogSource: It is a Verilog source file • systemVerilogSource: It is a SystemVerilog source file • user: It is a user defined source, described by the attribute "user"
user	attribute of ipxact:fileType	Can be: • xci: Xilinx Core Instance • dcp : It is a Vivado design checkpoint file
ipxact:description	ipxact:component	A short description of the IP

Keysight Standard Interfaces

The bus interfaces that are currently supported by PathWave FPGA to be used inside an IP component definition are described as Keysight Standard Interfaces. Each of these interfaces has IP-XACT definitions, which are defined by, and installed with, PathWave FPGA.

More specifically, for each interface, two IP-XACT files are defined:

- the **Bus Definition**: IP-XACT file with ipxact:busDefinition as root element
- the **Abstraction definition**: IP-XACT file with abstractionDefinition as root element

The **Bus Definition** is used to define the high-level details of an interface, such as if is addressable or not, if it supports direct connection between a master and a slave, etc.

Code Block 2 Example Bus Definition for AXI4-Stream interface

```
<ipxact:busDefinition xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xmlns:ipxact="http://www.accellera.org/XMLSchema/IPXACT/1685-
2014" xsi:schemaLocation="http://www.accellera.org/XMLSchema/IPXACT/1685-
2014/http://www.accellera.org/XMLSchema/IPXACT/1685-2014/index.xsd">
 <ipxact:vendor>keysight.com</ipxact:vendor>
 <ipxact:library>interfaces</ipxact:library>
 <ipxact:name>axis</ipxact:name>
 <ipxact:version>1.0</ipxact:version>
 <ipxact:directConnection>true</ipxact:directConnection>
 <ipxact:isAddressable>false</ipxact:isAddressable>
</ipxact:busDefinition>
```

The **Abstraction Definition** is used to define the low-level details of an interface, such as the port and the parameter list.

Code Block 3 Example Abstraction Definition for AXI4-Stream interface

```
<ipxact:abstractionDefinition
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:ipxact="http://www.accellera.org/XMLSchema/IPXACT/1685-2014"
xsi:schemaLocation="http://www.accellera.org/XMLSchema/IPXACT/1685-
2014/http://www.accellera.org/XMLSchema/IPXACT/1685-2014/index.xsd">
 <ipxact:vendor>keysight.com</ipxact:vendor>
 <ipxact:library>interfaces</ipxact:library>
 <ipxact:name>axis.absDef</ipxact:name>
 <ipxact:version>1.0</ipxact:version>
 <ipxact:busType vendor="keysight.com" library="interfaces" name="axis"
version="1.0"/>
 <ipxact:ports>
    <ipxact:port>
       <ipxact:logicalName>tdata</ipxact:logicalName>
       <ipxact:wire>
          <ipxact:qualifier>
             <ipxact:isData>true</ipxact:isData>
          </ipxact:qualifier>
          <ipxact:onMaster>
             <ipxact:presence>optional</ipxact:presence>
             <ipxact:width>64</ipxact:width>
             <ipxact:direction>out</ipxact:direction>
          </ipxact:onMaster>
          <ipxact:onSlave>
             <ipxact:presence>optional</ipxact:presence>
             <ipxact:width>64</ipxact:width>
             <ipxact:direction>in</ipxact:direction>
          </ipxact:onSlave>
          <ipxact:defaultValue>0</ipxact:defaultValue>
       </ipxact:wire>
    </ipxact:port>
    <ipxact:port>
       <ipxact:logicalName>tvalid</ipxact:logicalName>
       <ipxact:wire>
          <ipxact:onMaster>
             <ipxact:presence>required</ipxact:presence>
             <ipxact:width>1</ipxact:width>
             <ipxact:direction>out</ipxact:direction>
          </ipxact:onMaster>
          <ipxact:onSlave>
             <ipxact:presence>required</ipxact:presence>
             <ipxact:width>1</ipxact:width>
             <ipxact:direction>in</ipxact:direction>
          </ipxact:onSlave>
       </ipxact:wire>
    </ipxact:port>
       :
       :
       :
 </ipxact:ports>
</ipxact:abstractionDefinition>
```

Managing Multiple Clocks and Resets

PathWave FPGA needs to know which clock synchronous interfaces use. If there is only one clock in an IP block's definition, then there is no ambiguity. However, if there is more than one clock interface, then the tools need to know which clock corresponds to which interfaces. To do this, one adds the **ASSOCIATED_BUSIF** parameter to the bus interface definition of each clock interface. The value of the **ASSOCIATED_BUSIF** parameter is a colon separated list of the names of the interfaces that use that clock. This should include all the synchronous interfaces

(things such as AXI and PC_MEM interfaces) that use that clock. If every synchronous interface uses the same clock, then the **ASSOCIATED_BUSIF** can be set to the value ***** as a wildcard to denote all interfaces.

Additionally, reset signals are usually synchronous with a clock in order to generate clean reset events. If there are more than one clock and more than one reset signal, then PathWave FPGA also needs to know which reset signal is associated with a particular clock. To do this, one adds the **ASSOCIATED_RESET** parameter to the bus interface definition of the pertinent clock interface. The value of the **ASSOCIATED_RESET** parameter is the name of a single reset interface that should be used with that clock. Note that while **ASSOCIATED_BUSIF** can accept multiple colon separated names or the * wildcard, **ASSOCIATED_RESET** can only be a single name.

Parameterizing IP Designs

For added generality, IP-XACT standard allows the usage of *parameters* to control various aspects of the IP block's definition, so that the same block may be used with different configurations. These parameters can be simple constants such as 16, or they can be mathematical expressions involving multiple constants and/or other parameters. The format of expressions in IP-XACT are detailed in *Annex C* of the IP-XACT 1685-2014 standards document. The format is based on *System Verilog*'s expression syntax.

IP-XACT provides different ways to define parameters, however, in the context of PathWave FPGA, two methods are currently supported:

- Component Parameters
- Module Parameters

The following table summarizes the elements/attributes that PathWave takes into account when parsing an IP-XACT file, with respect to the parameters:

Element/Attribute	Parent Element	Content
ipxact:parameter (or ipxact:moduleParameter)	ipxact:parameters (or ipxact:moduleParameters)	The root element to define a parameter. It requires the definition of attributes and children element for the proper description of a parameter
resolve	attribute of ipxact:parameter (or ipxact:moduleParameter)	Can take one of the values: "user", "immediate" or "generated". To specify that a parameter should be configured by the user of the IP, the value "user" should be used. This will also display the parameter in the properties dialog of an IP inside PathWave FPGA This attribute defaults to "immediate" if not defined
type	attribute of ipxact:parameter (or ipxact:moduleParameter)	Defines the datatype of the value. Possible values are: "int", "bit", "byte". For a complete list, please refer to IP-XACT 1685-2014
parameterId	attribute of ipxact:parameter (or ipxact:moduleParameter)	Defines a unique (in the context of the IP-XACT file) ID for this parameter. This ID should then be used in any expression required within the file
ipxact::name	ipxact:parameter	The name of the parameter

Element/Attribute	Parent Element	Content
	(or ipxact:moduleParameter)	
ipxact:value	ipxact:parameter (or ipxact:moduleParameter)	The default value (or expression) of the parameter

Component Parameters

Parameters defined as children of the elements path *component->parameters*. These can be used throughout the IP-XACT document to configure any aspect of the file (can be used in any field that accepts expressions as values, e.g. other parameter values, port ranges, port presence etc.)

```
<ipxact:component>
    :
    :
 <ipxact:parameters>
      <ipxact:parameter resolve="user" type="int"
parameterId="gen_input_length" >
         <ipxact:name>gen_input_length</ipxact:name>

 <ipxact:value>3*uuid_5e192450_89f2_48a9_8906_ee47dbbe0b15</ipxact:value>
      </ipxact:parameter>
      <ipxact:parameter type="int"
parameterId="uuid_f4a7c3f8_a1b3_496a_9730_17d721278396" >
         <ipxact:name>output_length</ipxact:name>
         <ipxact:value>2*gen_input_length</ipxact:value>
      </ipxact:parameter>
      <ipxact:parameter resolve="user" type="int"
parameterId="uuid_5e192450_89f2_48a9_8906_ee47dbbe0b15" >
         <ipxact:name>supersample</ipxact:name>
         <ipxact:value>1</ipxact:value>
      </ipxact:parameter>
 </ipxact:parameters>
    :
    :
</ipxact:component>
```

Notes:

- The attribute *resolve="user"* indicates that these parameters are ones that the user can change when instantiating the IP block. If the parameter should always be calculated from other values or remain fixed, the attribute *resolve="immediate"* should be used. In that case the user will not be given the option of modifying the value of the parameter.
- The *parameterId* is the one used inside an expression (**not** the ipxact:name), in which a parameter participates (see ipxact:value of parameter gen_input_length). However, if the *ipxact:name* of the parameter is unique throughout the document, it can also be used as *parameterId*. This way it is easier to construct expressions using parameters (see ipxact:value of parameter *output_length*)
- The value of *output_length* parameter **shall not** be modifiable **directly** by user input (as it does not contain the attribute *resolve* set to *"user"*), rather, **indirectly**, through the *input_length* parameter, as its expression implies (i.e. *2*gen_input_length*)
- The value of gen_input_length parameter is defined as user modifiable. That means that the expression **shall not** play any role, other than defining the default value. Therefore, if a user selects a value of "10" for this parameter, and a value of "5" for the parameter *supersample*, the final value of gen_input_length will be "10" and **not** "15" (*3*supersample*)

Module Parameters

Parameters defined as children of the elements path *component->model->instantiations->componentInstantiation->moduleParameters*. These are more specific to a Module Definition. Represent the generics of a VHDL entity, or the parameters of a Verilog module.

Code Block 4 Example Module Parameters Definition

```
<ipxact:component>
    :
    :
    <ipxact:model>
        <ipxact:instantiations>
            <ipxact:componentInstantiation>
                <ipxact:name>flat_vhdl_component</ipxact:name>
            <ipxact:language>vhdl</ipxact:language>
            <ipxact:moduleName>parameterizedIp</ipxact:moduleName>
            <ipxact:moduleParameters>
               <ipxact:moduleParameter type="int"
parameterId="input_length" resolve="user">
                    <ipxact:name>input_length</ipxact:name>
                    <ipxact:value>3*supersample</ipxact:value>
                </ipxact:moduleParameter>
              <ipxact:moduleParameter type="int"
parameterId="output_length">
                    <ipxact:name>output_length</ipxact:name>
                    <ipxact:value>2*input_length</ipxact:value>
                </ipxact:moduleParameter>
              <ipxact:moduleParameter type="int"
parameterId="supersample" resolve="user">
                    <ipxact:name>supersample</ipxact:name>

 <ipxact:value>uuid_5e192450_89f2_48a9_8906_ee47dbbe0b15</ipxact:value>
                </ipxact:moduleParameter>
            </ipxact:moduleParameters>
       </ipxact:componentInstantiation>
    </ipxact:instantiations>
    :
    :
 </ipxact:model>
    :
    :
</ipxact:component>
```

Notes:

- The guides for creating component parameters also apply to the module parameters.
- The value of the *supersample* parameter depends on a parameter defined elsewhere in the document (*uuid_5e192450_89f2_48a9_8906_ee47dbbe0b15* is the *parameterId* defined for the parameter supersample, defined in the previous example and can exist in the same document)

Example: Parameterized Port Sizing

IP-XACT parameters can be used to define the bounds (sizes) of the IP module's ports. These expressions may be solely the parameterId of an ipxact:moduleParameter or may be more complicated expressions as shown in this example:

```
<ipxact:port>
    <ipxact:name>Din_vector</ipxact:name>
```

```
    <ipxact:wire>
        <ipxact:direction>in</ipxact:direction>
        <ipxact:vectors>
            <ipxact:vector>
                <ipxact:left>input_length*supersample-1</ipxact:left>
                <ipxact:right>0</ipxact:right>
            </ipxact:vector>
        </ipxact:vectors>
        <ipxact:wireTypeDefs>
            <ipxact:wireTypeDef>
                <ipxact:typeName>std_logic_vector</ipxact:typeName>
                <ipxact:typeDefinition></ipxact:typeDefinition>
            </ipxact:wireTypeDef>
        </ipxact:wireTypeDefs>
    </ipxact:wire>
</ipxact:port>
```

Note:

- **Only ipxact:moduleParameter parameters can be used in expressions defining port ranges**. This is because the actual expression will also be used during code generation and only the ipxact:moduleParameters are defined at that time
- Tools such as *Kactus2* can facilitate defining and evaluating expressions.

IP Restrictions

For IP to be used in PathWave FPGA, there will need to be a set of IP restrictions that specify which BSPs and FPGA device families the IP can be used with. This information will be used to determine which IP will show up in the IP catalog in the GUI for use in a design. Only the IP that will work with a given BSP and FPGA will show up for a design so that the user cannot place incompatible IP in a design.

An IP developer may specify in the IP-XACT which BSPs (eg. M3102A, M3202A), which FPGA vendors (eg. Xilinx), and which FPGA families (eg. Virtex, Kintex) are supported. If the IP can work for all families for a given FPGA vendor or all BSPs, then the family parameter or the bsp parameter does not need to be set.

IP Restrictions Format

The IP restrictions will be added to the IP-XACT file inside the 'ipxact:vendorExtensions' element of an 'ipxact:component'. The elements to be used are defined by Keysight and are as follows:

Element	Parent Element	Content
keysight:ipMetadata	ipxact:vendorExtensions (direct child of ipxact:component)	This is the root element of the Keysight Vendor Extensions for IP metadata
keysight:supportedHardware	keysight:ipMetadata	Contains information about the hardware to which this IP is supported
keysight:supportedBoards	keysight:supportedHardware	Contains a list of Vendor-Boards pairs of supported boards. **If this element is not specified, all boards are supported**
keysight:vendorBoards	keysight:supportedBoards	A Vendor-Boards pair

Element	Parent Element	Content
keysight:vendor	keysight:vendorBoards	The name of the vendor. Should be equal to the internet domain of the vendor of the boards (e.g. keysight.com)
keysight:boards	keysight:vendorBoards	Contains a list of board names that are supported
keysight:board	keysight:boards	The name of the board where this IP can be used
keysight:supportedParts	keysight:supportedHardware	Contains a list of Vendor-Parts pairs of supported FPGA parts. **If this element is not specified, all FPGA parts are supported**
keysight:vendorParts	keysight:supportedParts	A Vendor-Parts pair
keysight:vendor	keysight:vendorParts	Vendor's name. Should be equal to the internet domain of the vendor of the parts (e.g. keysight.com)
keysight:families	keysight:vendorParts	Contains a list of family names that are supported
keysight:family	keysight:families	The name of the family as this is defined by the part number (e.g. 'xc7k' should be used if the supported family is 'Kintex-7')

To use any of the Keysight defined elements inside an IP-XACT file, you need to specify the 'keysight' namespace: "xmlns:keysight=''http://www.keysight.com'''' in the xml root element (i.e. ipxact:component)

IP Categorization

In addition to defining the library in which the IP belongs, it is possible to define a subcategory for an IP. To achieve that, PathWave FPGA has defined some extension elements for IP-XACT.

The IP restrictions will be added to the IP-XACT file inside the 'ipxact:vendorExtensions' element of an 'ipxact:component'. The elements to be used are defined by Keysight and are as follows:

Element	Parent Element	Content
keysight:ipMetadata	ipxact:vendorExtensions (direct child of ipxact:component)	This is the root element of the Keysight Vendor Extensions for IP metadata
keysight:categories	keysight:ipMetadata	A list of categories. Currently, only one category can be specified
keysight:category	keysight: categories	The name of the category that this IP belongs into

To use any of the Keysight defined elements inside an IP-XACT file, you need to specify the 'keysight' namespace: "xmlns:keysight="http://www.keysight.com"" in the xml root element (i.e. ipxact:component)

IP Naming Collisions

PathWave FPGA does not accept IP with the same name to be loaded at the same time in a project. PathWave FPGA uses the concept of VLNV for identifying IP and reporting naming collisions. VLNV stands for Vendor-Library-Name-Version and is a concept introduced by IP-XACT. The VLNV of an IP is defined in the first four fields of an IP-XACT component (see IP-XACT definition)

For more information on naming collisions and how to resolve them, please read here.

For the case of an IP developer, this might happen as multiple versions of the same IP might be created in the development phase. Even though the case of multiple IPs with the same VLNV but different contents is detected by PathWave FPGA, it is recommended to update the version field of the IP-XACT file for every change applied to the file. This will provide better issue reporting and easier resolution.

An Example IP-XACT File

Code Block 5 Sample IP-XACT file

```
<ipxact:component xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:ipxact="http://www.accellera.org/XMLSchema/IPXACT/1685-2014"
xmlns:keysight="http://www.keysight.com"
xsi:schemaLocation="http://www.accellera.org/XMLSchema/IPXACT/1685-2014
http://www.accellera.org/XMLSchema/IPXACT/1685-2014/index.xsd">
  <ipxact:vendor>keysight.com</ipxact:vendor>
  <ipxact:library>myCustomLibrary</ipxact:library>
  <ipxact:name>SampleIp</ipxact:name>
  <ipxact:version>1.0</ipxact:version>
  <ipxact:busInterfaces>
    <ipxact:busInterface>
      <ipxact:name>clkSignal</ipxact:name>
      <ipxact:busType vendor="keysight.com" library="interfaces"
name="clock" version="1.0"/>
      <ipxact:abstractionTypes>
        <ipxact:abstractionType>
          <ipxact:abstractionRef vendor="keysight.com"
library="interfaces" name="clock.absDef" version="1.0"/>
          <ipxact:portMaps>
            <ipxact:portMap>
              <ipxact:logicalPort>
                <ipxact:name>clk</ipxact:name>
              </ipxact:logicalPort>
              <ipxact:physicalPort>
                <ipxact:name>clk</ipxact:name>
              </ipxact:physicalPort>
            </ipxact:portMap>
          </ipxact:portMaps>
        </ipxact:abstractionType>
      </ipxact:abstractionTypes>
      <ipxact:slave/>
```

```
      <ipxact:parameters>
        <ipxact:parameter
parameterId="uuid_4e5d34f4_ff5d_4244_92b4_c0d0ec78d043">
          <ipxact:name>ASSOCIATED_BUSIF</ipxact:name>
          <ipxact:value>myAxiStreamMaster:myAxiStreamSlave</ipxact:value>
        </ipxact:parameter>
        <ipxact:parameter
parameterId="uuid_c127b078_eb51_42f4_aaf8_58e93ad84b21">
          <ipxact:name>ASSOCIATED_RESET</ipxact:name>
          <ipxact:value>Reset</ipxact:value>
        </ipxact:parameter>
      </ipxact:parameters>
    </ipxact:busInterface>
    <ipxact:busInterface>
      <ipxact:name>Reset</ipxact:name>
      <ipxact:busType vendor="keysight.com" library="interfaces"
name="nRst" version="1.0"/>
      <ipxact:abstractionTypes>
        <ipxact:abstractionType>
          <ipxact:abstractionRef vendor="keysight.com"
library="interfaces" name="nRst.absDef" version="1.0"/>
          <ipxact:portMaps>
            <ipxact:portMap>
              <ipxact:logicalPort>
                <ipxact:name>nRst</ipxact:name>
              </ipxact:logicalPort>
              <ipxact:physicalPort>
                <ipxact:name>rstn</ipxact:name>
              </ipxact:physicalPort>
            </ipxact:portMap>
          </ipxact:portMaps>
        </ipxact:abstractionType>
      </ipxact:abstractionTypes>
      <ipxact:slave/>
    </ipxact:busInterface>
    <ipxact:busInterface>
      <ipxact:name>myAxiStreamSlave</ipxact:name>
      <ipxact:busType vendor="keysight.com" library="interfaces"
name="axis" version="1.0"/>
      <ipxact:abstractionTypes>
        <ipxact:abstractionType>
          <ipxact:abstractionRef vendor="keysight.com"
library="interfaces" name="axis.absDef" version="1.0"/>
          <ipxact:portMaps>
            <ipxact:portMap>
              <ipxact:logicalPort>
                <ipxact:name>tvalid</ipxact:name>
              </ipxact:logicalPort>
              <ipxact:physicalPort>
                <ipxact:name>my_stream_valid_in</ipxact:name>
              </ipxact:physicalPort>
            </ipxact:portMap>
            <ipxact:portMap>
              <ipxact:logicalPort>
                <ipxact:name>tuser</ipxact:name>
              </ipxact:logicalPort>
              <ipxact:physicalPort>
                <ipxact:name>my_stream_user_in</ipxact:name>
              </ipxact:physicalPort>
            </ipxact:portMap>
            <ipxact:portMap>
              <ipxact:logicalPort>
                <ipxact:name>tdata</ipxact:name>
              </ipxact:logicalPort>
              <ipxact:physicalPort>
                <ipxact:name>my_stream_data_in</ipxact:name>
```

```
              </ipxact:physicalPort>
            </ipxact:portMap>
          </ipxact:portMaps>
        </ipxact:abstractionType>
      </ipxact:abstractionTypes>
      <ipxact:slave/>
    </ipxact:busInterface>
    <ipxact:busInterface>
      <ipxact:name>myAxiStreamMaster</ipxact:name>
      <ipxact:busType vendor="keysight.com" library="interfaces"
name="axis" version="1.0"/>
      <ipxact:abstractionTypes>
        <ipxact:abstractionType>
          <ipxact:abstractionRef vendor="keysight.com"
library="interfaces" name="axis.absDef" version="1.0"/>
          <ipxact:portMaps>
            <ipxact:portMap>
              <ipxact:logicalPort>
                <ipxact:name>tvalid</ipxact:name>
              </ipxact:logicalPort>
              <ipxact:physicalPort>
                <ipxact:name>my_stream_valid_out</ipxact:name>
              </ipxact:physicalPort>
            </ipxact:portMap>
            <ipxact:portMap>
              <ipxact:logicalPort>
                <ipxact:name>tdata</ipxact:name>
              </ipxact:logicalPort>
              <ipxact:physicalPort>
                <ipxact:name>my_stream_data_out</ipxact:name>
              </ipxact:physicalPort>
            </ipxact:portMap>
          </ipxact:portMaps>
        </ipxact:abstractionType>
      </ipxact:abstractionTypes>
      <ipxact:master/>
    </ipxact:busInterface>
  </ipxact:busInterfaces>
  <ipxact:model>
    <ipxact:ports>
      <ipxact:port>
        <ipxact:name>clk</ipxact:name>
        <ipxact:wire>
          <ipxact:direction>in</ipxact:direction>
          <ipxact:wireTypeDefs>
            <ipxact:wireTypeDef>
              <ipxact:typeName>std_logic</ipxact:typeName>
              <ipxact:typeDefinition></ipxact:typeDefinition>
            </ipxact:wireTypeDef>
          </ipxact:wireTypeDefs>
        </ipxact:wire>
      </ipxact:port>
      <ipxact:port>
        <ipxact:name>rstn</ipxact:name>
        <ipxact:wire>
          <ipxact:direction>in</ipxact:direction>
          <ipxact:wireTypeDefs>
            <ipxact:wireTypeDef>
              <ipxact:typeName>std_logic</ipxact:typeName>
              <ipxact:typeDefinition></ipxact:typeDefinition>
            </ipxact:wireTypeDef>
          </ipxact:wireTypeDefs>
        </ipxact:wire>
      </ipxact:port>
      <ipxact:port>
        <ipxact:name>my_stream_valid_in</ipxact:name>
```

```
      <ipxact:wire>
        <ipxact:direction>in</ipxact:direction>
        <ipxact:wireTypeDefs>
          <ipxact:wireTypeDef>
            <ipxact:typeName>std_logic</ipxact:typeName>
            <ipxact:typeDefinition></ipxact:typeDefinition>
          </ipxact:wireTypeDef>
        </ipxact:wireTypeDefs>
      </ipxact:wire>
    </ipxact:port>
    <ipxact:port>
      <ipxact:name>my_stream_data_in</ipxact:name>
      <ipxact:wire>
        <ipxact:direction>in</ipxact:direction>
        <ipxact:vectors>
          <ipxact:vector>
            <ipxact:left>79</ipxact:left>
            <ipxact:right>0</ipxact:right>
          </ipxact:vector>
        </ipxact:vectors>
        <ipxact:wireTypeDefs>
          <ipxact:wireTypeDef>
            <ipxact:typeName>std_logic_vector</ipxact:typeName>
            <ipxact:typeDefinition></ipxact:typeDefinition>
          </ipxact:wireTypeDef>
        </ipxact:wireTypeDefs>
      </ipxact:wire>
    </ipxact:port>
    <ipxact:port>
      <ipxact:name>my_stream_user_in</ipxact:name>
      <ipxact:wire>
        <ipxact:direction>in</ipxact:direction>
        <ipxact:vectors>
          <ipxact:vector>
            <ipxact:left>0</ipxact:left>
            <ipxact:right>0</ipxact:right>
          </ipxact:vector>
        </ipxact:vectors>
        <ipxact:wireTypeDefs>
          <ipxact:wireTypeDef>
            <ipxact:typeName>std_logic_vector</ipxact:typeName>
            <ipxact:typeDefinition></ipxact:typeDefinition>
          </ipxact:wireTypeDef>
        </ipxact:wireTypeDefs>
      </ipxact:wire>
    </ipxact:port>
    <ipxact:port>
      <ipxact:name>my_stream_valid_out</ipxact:name>
      <ipxact:wire>
        <ipxact:direction>out</ipxact:direction>
        <ipxact:wireTypeDefs>
          <ipxact:wireTypeDef>
            <ipxact:typeName>std_logic</ipxact:typeName>
            <ipxact:typeDefinition></ipxact:typeDefinition>
          </ipxact:wireTypeDef>
        </ipxact:wireTypeDefs>
      </ipxact:wire>
    </ipxact:port>
    <ipxact:port>
      <ipxact:name>my_stream_data_out</ipxact:name>
      <ipxact:wire>
        <ipxact:direction>out</ipxact:direction>
        <ipxact:vectors>
          <ipxact:vector>
            <ipxact:left>79</ipxact:left>
            <ipxact:right>0</ipxact:right>
```

```
            </ipxact:vector>
          </ipxact:vectors>
          <ipxact:wireTypeDefs>
            <ipxact:wireTypeDef>
              <ipxact:typeName>std_logic_vector</ipxact:typeName>
              <ipxact:typeDefinition></ipxact:typeDefinition>
            </ipxact:wireTypeDef>
          </ipxact:wireTypeDefs>
        </ipxact:wire>
      </ipxact:port>
    </ipxact:ports>
  </ipxact:model>
  <ipxact:fileSets>
    <ipxact:fileSet>
      <ipxact:name>synthesis</ipxact:name>
      <ipxact:file>
        <ipxact:name>sampleIp.vhd</ipxact:name>
        <ipxact:fileType>vhdlSource</ipxact:fileType>
      </ipxact:file>
    </ipxact:fileSet>
  </ipxact:fileSets>
  <ipxact:description>This is a Sample IP. It contains two Stream
Interfaces and two system ports</ipxact:description>
  <ipxact:vendorExtensions>
    <keysight:ipMetadata>
      <keysight:supportedHardware>
        <keysight:supportedBoards>
          <keysight:vendorBoards>
            <keysight:vendor>keysight.com</keysight:vendor>
            <keysight:boards>
              <keysight:board>M3202A</keysight:board>
            </keysight:boards>
          </keysight:vendorBoards>
        </keysight:supportedBoards>
        <keysight:supportedParts>
          <keysight:vendorParts>
            <keysight:vendor>xilinx.com</keysight:vendor>
            <keysight:families>
              <keysight:family>xc7k</keysight:family>
            </keysight:families>
          </keysight:vendorParts>
        </keysight:supportedParts>
      </keysight:supportedHardware>
      <keysight:categories>
        <keysight:category>General</keysight:category>
      </keysight:categories>
    </keysight:ipMetadata>
  </ipxact:vendorExtensions>
</ipxact:component>
```

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

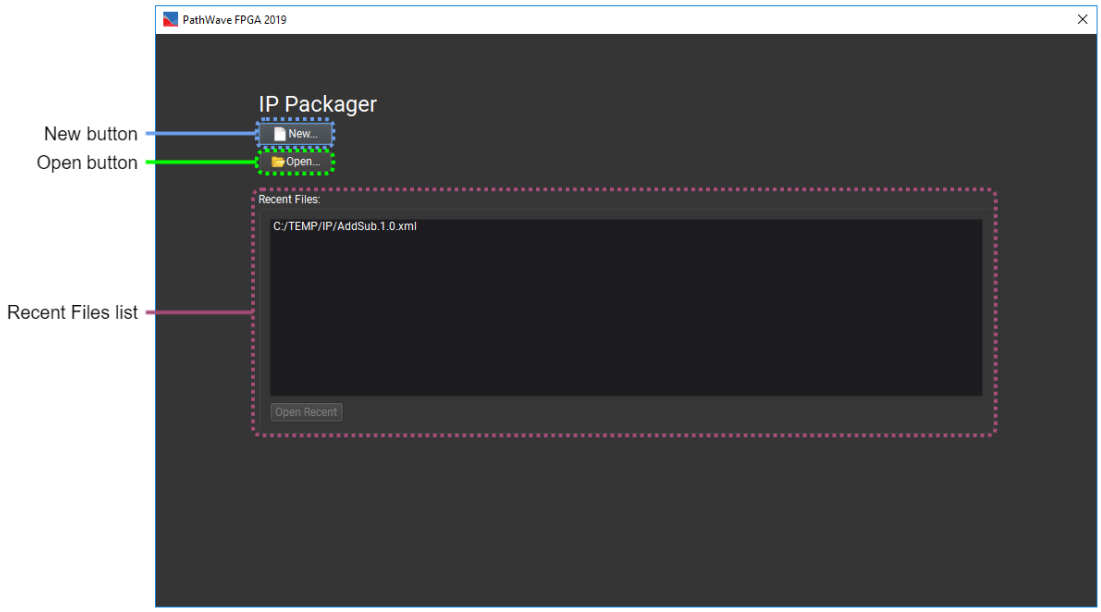
To open up *IP Packager* GUI, start *PathWave FPGA*, go to the *Tools* menu and click *IP Packager*. This will bring up the *IP Packager* GUI.

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.

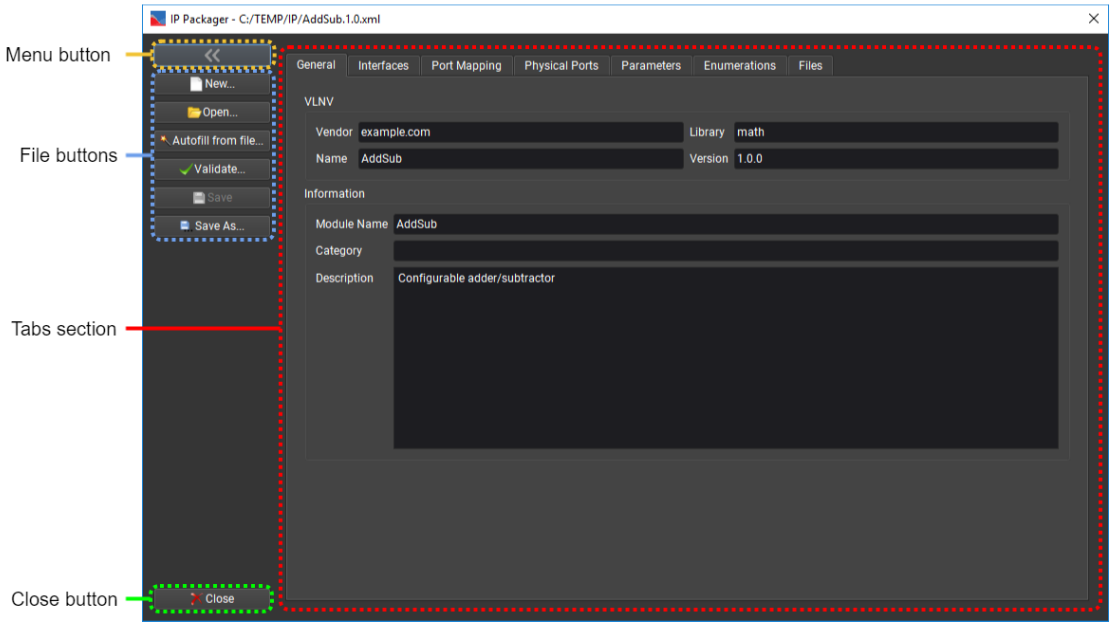
Open Button

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.

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. The default for the checkbox controlling interface inference is set in the PathWave FPGA Configuration dialog. See the **Infer Interfaces** button in the **Port Mapping Tab** for a discussion of the port naming inference rules. 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.

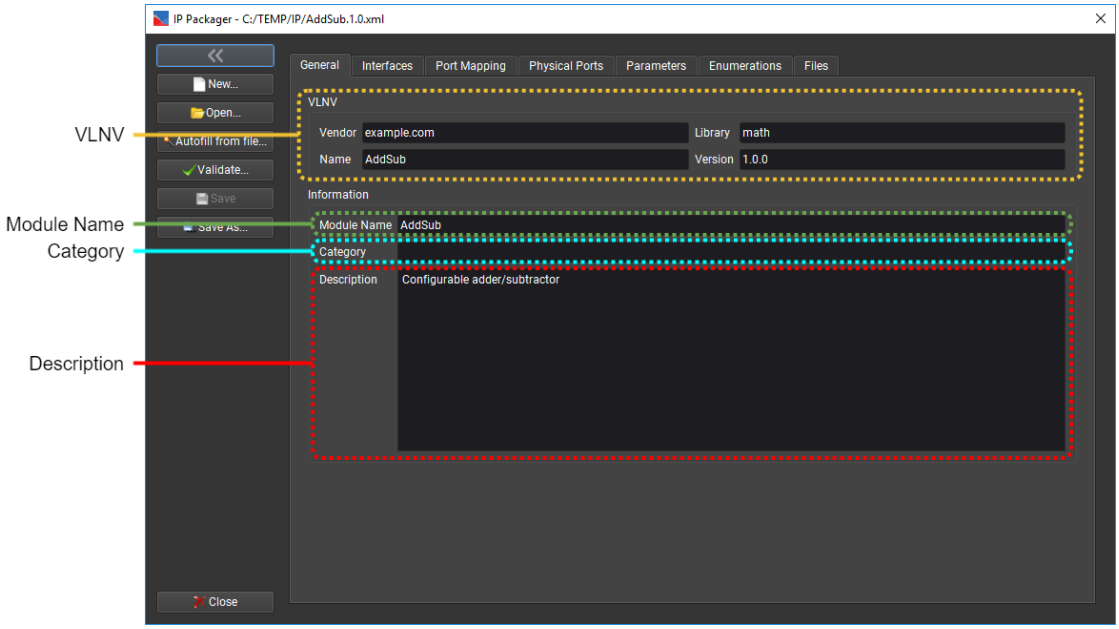
Close button

This will close the *IP Packager* window. The user will be prompted if unsaved changes should be saved. If a project is loaded in *PathWave FPGA* while starting *IP Packager* GUI, the user will be asked if any valid IP-XACT files that were generated during the *IP Packager* session should be imported into that 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 VLNV of an IP is defined in the first four fields of an IP-XACT component (see IP-XACT definition).

PathWave FPGA uses the VLNV value to resolve name conflicts. The library field is used to categorize IP in the IP Library.

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 Library. 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 VLNV *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:

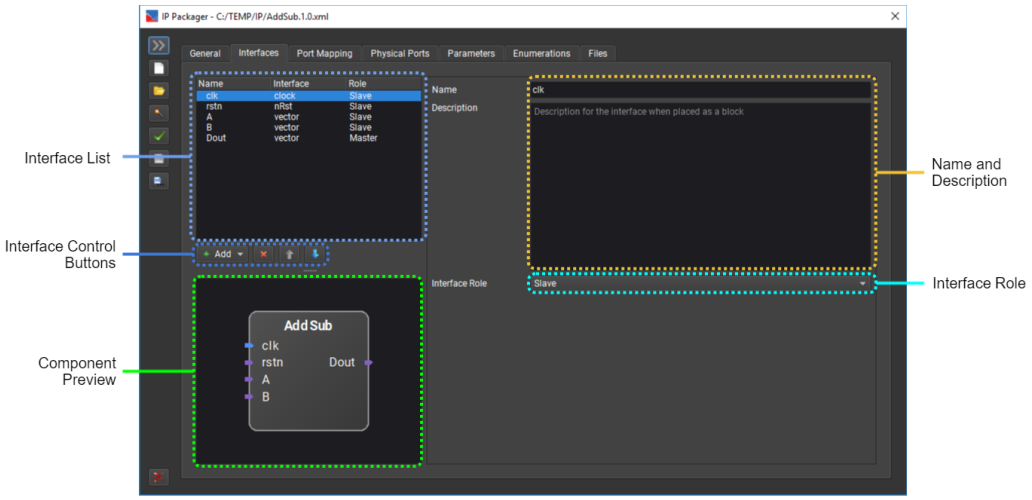
- ALGORITHMS
  - MATH
    - ADDERS
      - StreamAdder

Description

This is an optional field. 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.

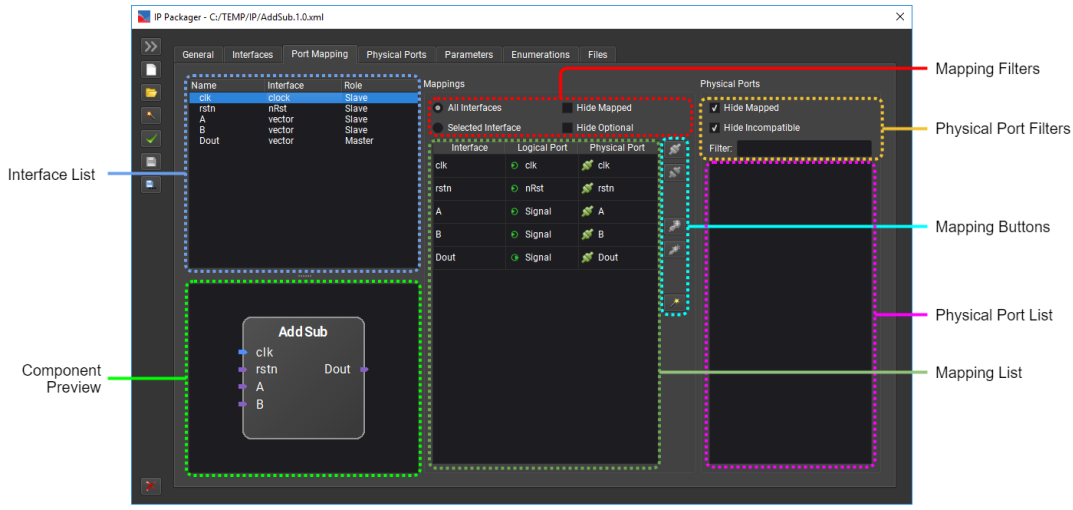
Interfaces Tab

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. The 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. The inferred interface name will usually be the common prefix of the included physical ports. The inference rules follow the conventions in the Xilinx document ug1118, for packaging custom ip in Vivado. Clock, reset, AXI4, and PathWave FPGA PC_MEM interfaces may be inferred. 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**.

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

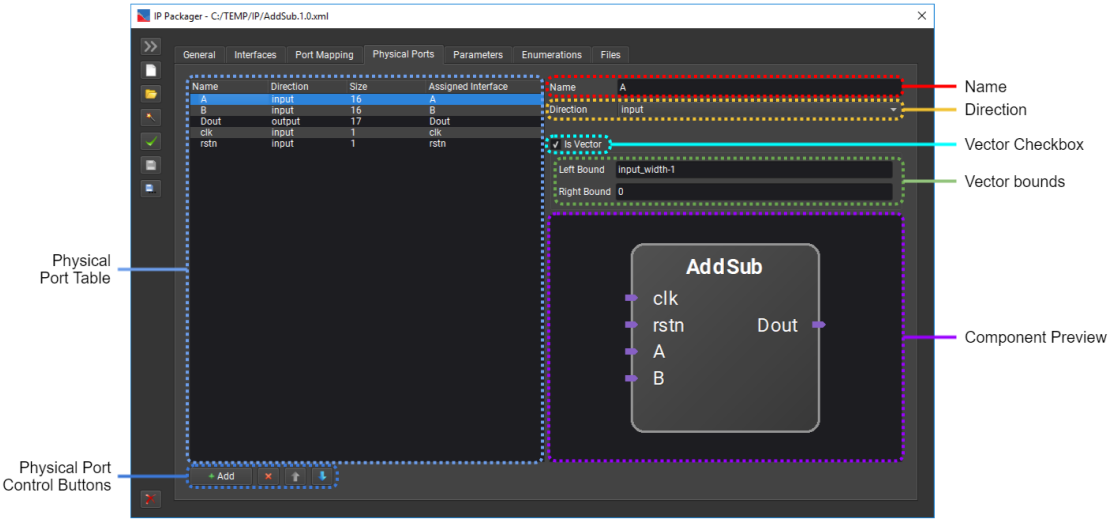
 icon is used for unmapped compatible physical ports

 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.



Vector Bounds

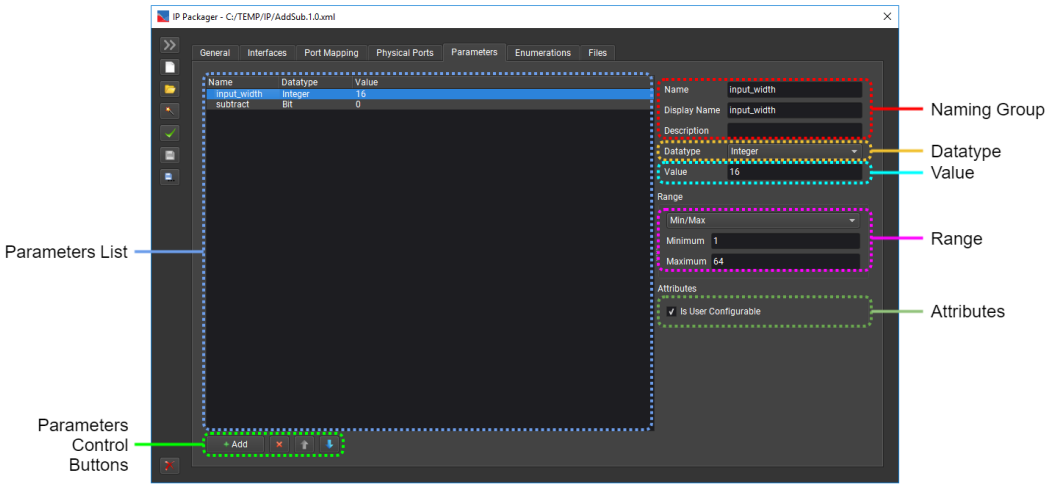
Configure the left and right bounds of a vector to set the width. Either the left or right bound must be 0.

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 the user to add/modify/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.

Naming Group

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 to the user in PathWave FPGA.
- **Description**: a description for this parameter. The description will be available to the user in PathWave FPGA.

Datatype

A list of supported datatypes for the parameter:.

- *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

Value

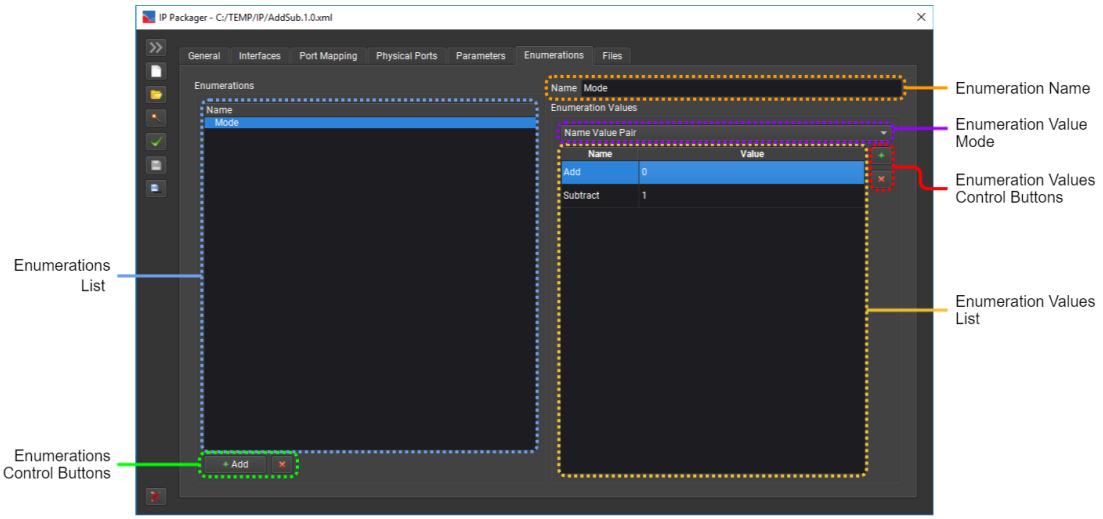
The value or expression to be used by default for this parameter. The possible value is restricted by the selected datatype and the specified Range.

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 -∞.
    - *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 +∞.
- **Enumeration**: If this is selected, a combobox 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, it allows the user to give a different value than the default to this parameter. User Configurable parameters will be displayed to the *PathWave FPGA* users in the component dialog of this IP.

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**.

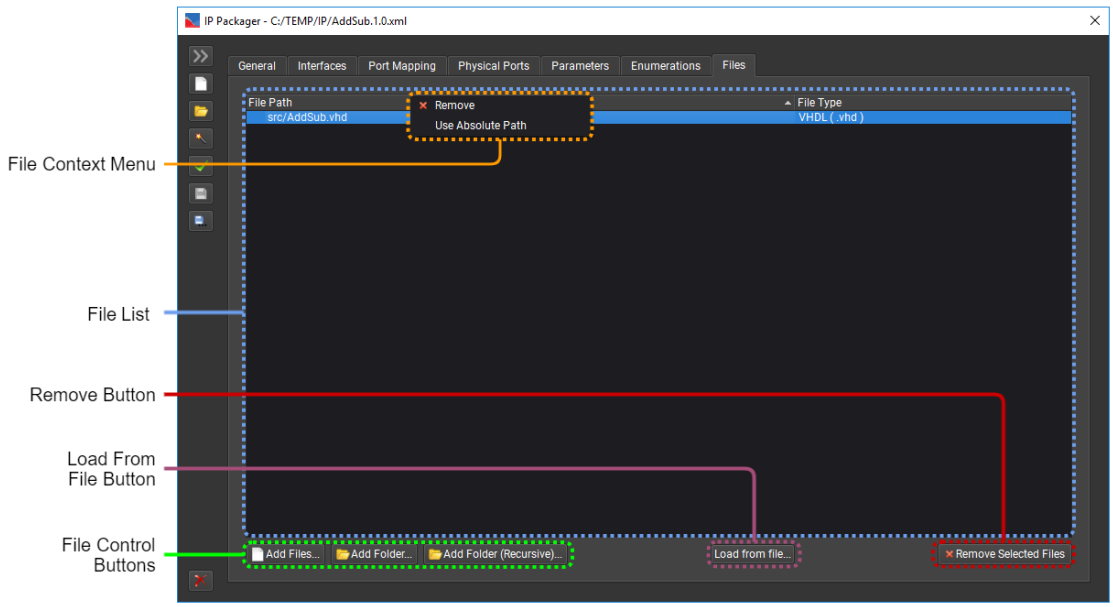
Enumeration Values Control Buttons

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.



File List

Displays the list of files that will be used during the synthesis and implementation of the IP. There must be at least one 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.

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. The default for the checkbox controlling interface inference is set in the PathWave FPGA Configuration dialog. See the **Infer Interfaces** button in the **Port Mapping Tab** for a discussion of the port naming inference rules.

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.

Keysight Standard Interfaces

Introduction

To facilitate connectivity between IP blocks and Sandbox 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 sandbox 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.
    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. PC-MEM: a very light weight Keysight proprietary interface.
    a. Can be bi-directional.
    b. Includes addressing.
    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.

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:

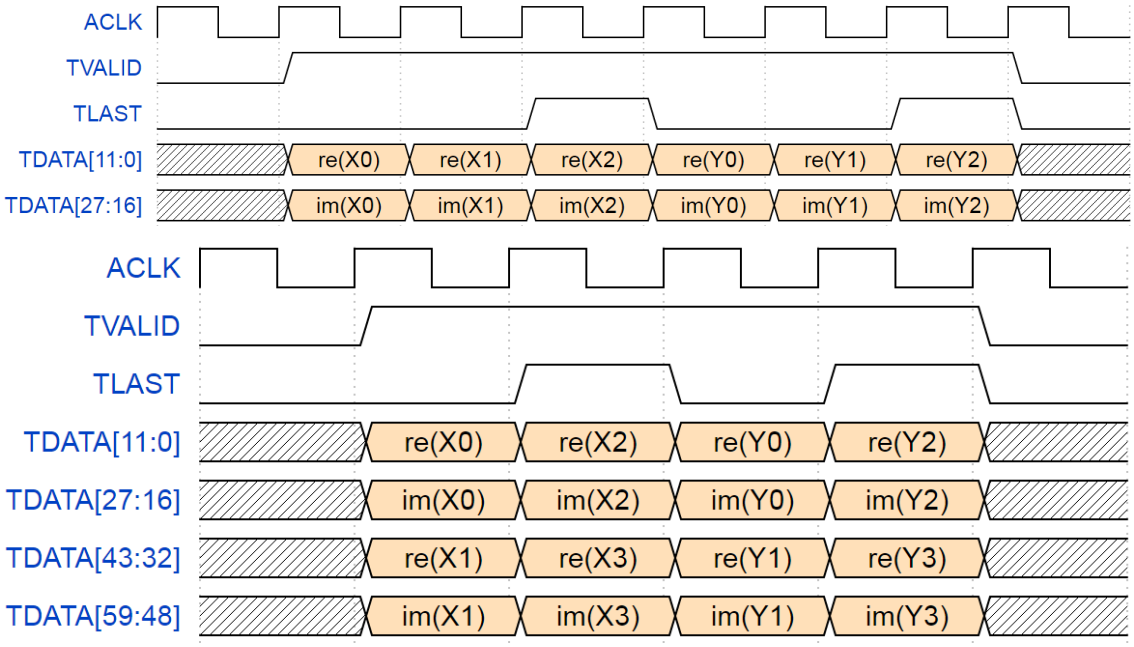
1. Control Bus Slaves. Typically these would be register control/status blocks where the driver could read and write status and control data.
2. Control Bus Master. This is for the case where the user IP wants to communicate with external devices via the PCIe (or other host control) bus, e.g. write to 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

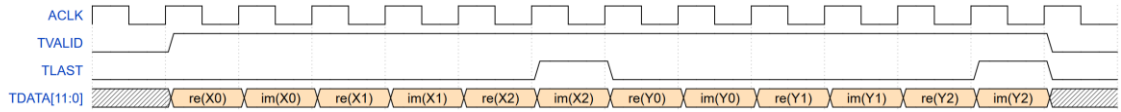
Most of the data that PathWave FPGA will be processing is likely to be fixed point (scaled ints) of varying bit widths. To facilitate interconnection of IP, limit the amount of data width conversion, and allow the use of standard interfaces, PathWave FPGA standardizes on data widths that are an integral 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.

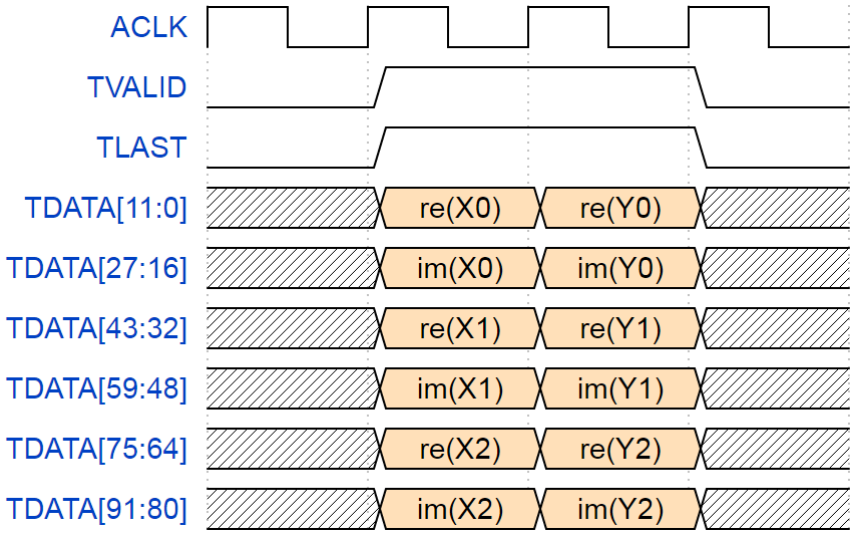
It is not uncommon to process complex data (that is, data consisting of a real and an imaginary component). If complex data is being sent over a single stream, the real and imaginary parts will be sent in parallel over a wider stream with the real part will go in the least significant word. For Serial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

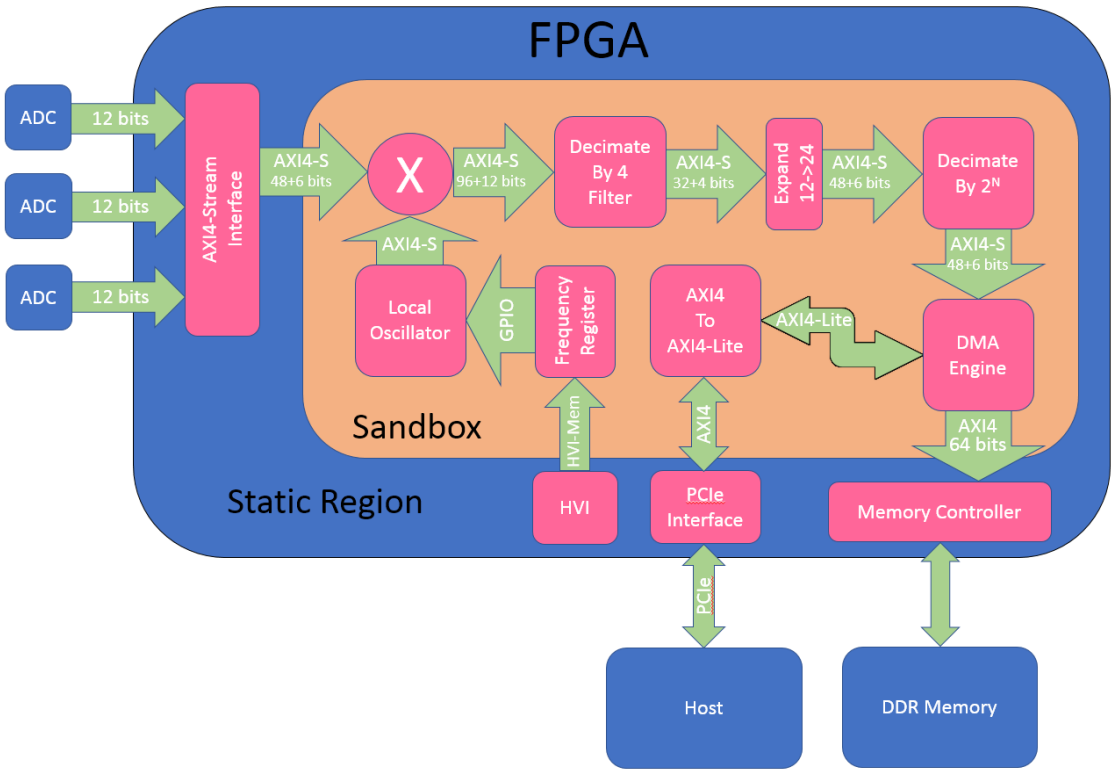
Polarity

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.
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, AXI4-Streaming	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	AXI4-Streaming	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.
PC-MEM	PC-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

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. A BSP is made up of two parts, the *FPGA support package* (FSP) and the *run-time support package* (RSP) .
FPGA support package (FSP)	The portion of the BSP that allows you to build a bit file for the target FPGA.
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 another compatible port.
Module	Either a top level module or submodule that is currently the top level module for simulation purposes
Port	An input or output signal to a block.
Program archive	An archive file (.k7z) containing one or more bit files and associated metadata.
Run-time support package (RSP)	The portion of the BSP that allows you to control your target FPGA. It provides a C API that you can use to download and verify your FPGA bit image.
Sandbox	The user-configurable region in the FPGA.
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.