Visual Applets 3

Creating Custom Operators and Custom Libraries

Concept Description and User Guide

Imprint

Basler AG
Konrad-Zuse-Ring 28
68163 Mannheim, Germany
Tel.: +49 (0) 621 789507 0
Fax: +49 (0) 621 789507 10

Document Version: 2.1
Document Language: en (US)

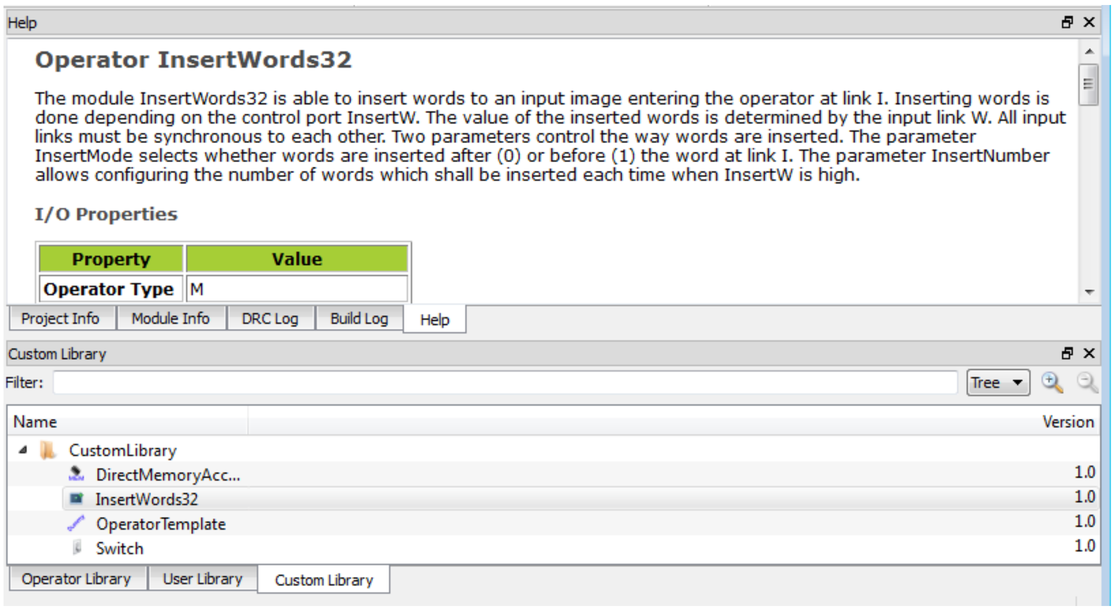
Last Change: June 2021

Contents

1 Introduction

With the VisualApplets 3 extension *Expert*, you have the possibility to convert image processing modules you have designed in VHDL into VisualApplets operators.

You incorporate your modules as IP cores into VisualApplets. Each IP core builds one operator. After implementation, these operators work like built-in VisualApplets operators. Operators implemented in such a way are called ***Custom Operators***.



Operator InsertWords32

The module InsertWords32 is able to insert words to an input image entering the operator at link I. Inserting words is done depending on the control port InsertW. The value of the inserted words is determined by the input link W. All input links must be synchronous to each other. Two parameters control the way words are inserted. The parameter InsertMode selects whether words are inserted after (0) or before (1) the word at link I. The parameter InsertNumber allows configuring the number of words which shall be inserted each time when InsertW is high.

I/O Properties

Property	Value
Operator Type	M

Name	Version
CustomLibrary	
DirectMemoryAcc...	1.0
InsertWords32	1.0
OperatorTemplate	1.0
Switch	1.0

To make your custom operators available on the VisualApplets GUI, you also need to define one or more custom libraries that contain the custom operator(s.) Each custom operator needs to be part of one specific custom library.

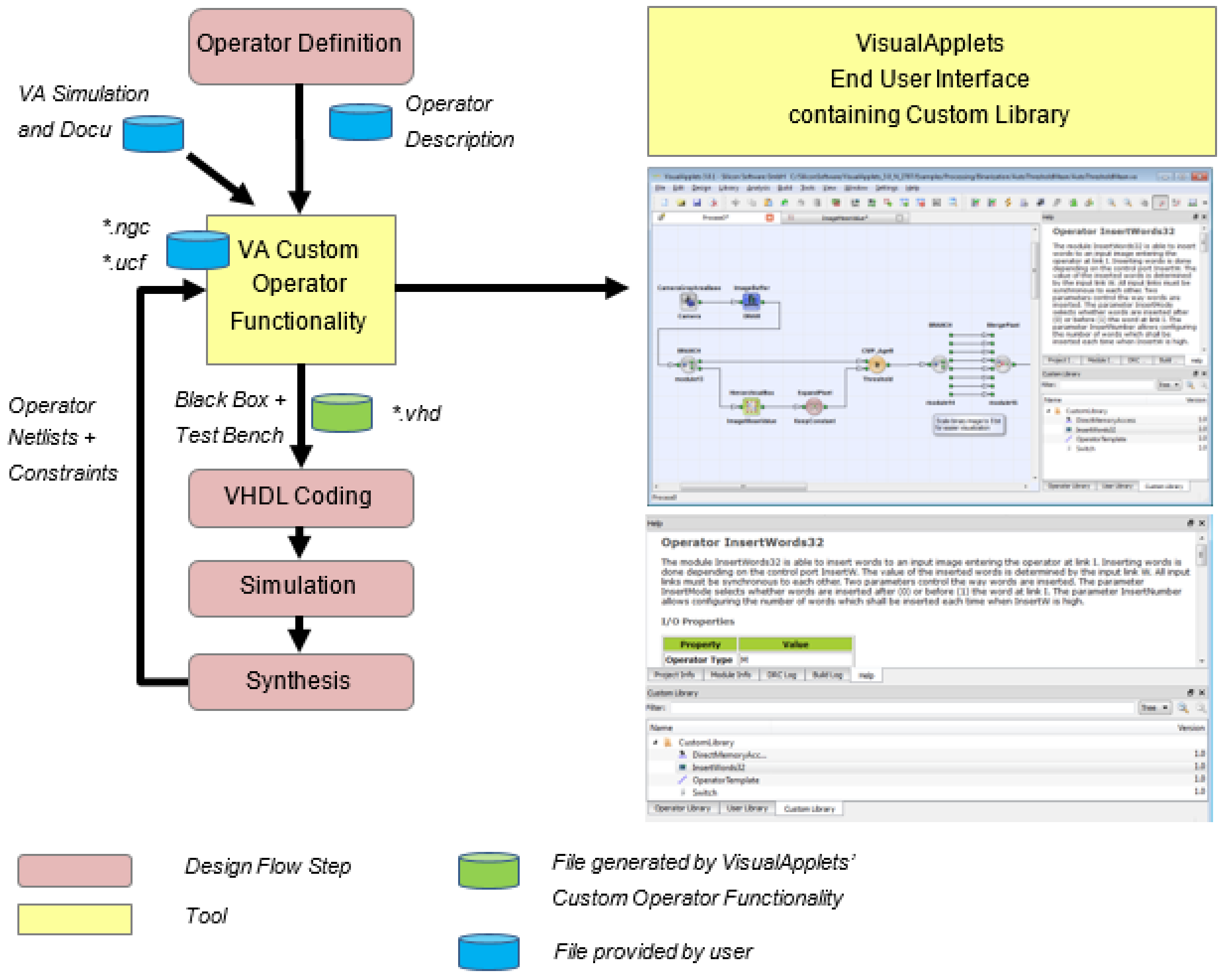
1.1 Workflow

You add a new ***Custom Operator*** to VisualApplets in just a few steps. You can complete the whole work flow by your own:

1. Specify the custom operator's main properties and its interface directly on the VA GUI (operator name, operator version, number and properties of required image in, image out, memory ports, etc.).
2. Based on your input of step 1, let VisualApplets generate the VHDL code for the operator interface (black box) and a VHDL test bench for testing your implementation.

3. Wrap your HDL code so that its interface matches the generated black box. For testing your implementation the automatically generated test bench may help.
4. Create a net list of your implementation. Also create a constraints file if required.
5. Optionally, create the operator documentation (for the operator help window) and a simulation model (that later allows to simulate a VA design containing the custom operator).
6. Edit the custom operator in VA again: Add the generated netlist and optionally also the help files and simulation model.

After these steps, your image processing module is available as custom operator directly in VisualApplets and can be used the same way as any other operator. The custom libraries are saved as *.vl files (similar to the user libraries). They can be deployed and distributed in this format.



Operator InsertWords32

The module InsertWords32 is able to insert words to an input image entering the operator at link I. Inserting words is done depending on the control port InsertW. The value of the inserted words is determined by the input link W. All input links must be synchronous to each other. Two parameters control the way words are inserted. The parameter InsertMode selects whether words are inserted after (0) or before (1) the word at link I. The parameter InsertNumber allows configuring the number of words which shall be inserted each time when InsertW is high.

I/O Properties

Property	Value
Operator Type	M

Figure 1: Custom Operators design flow.

1.2 VisualApplets Custom Operator Functionality

VisualApplets (i.e., the VisualApplets Custom Operator Functionality) is used two times during this work flow:

1. For the generation of an operator prototype in VisualApplets allowing to export HDL code for defining the concerning IP core interface (black box and test bench).
2. For completing the operator by adding the necessary files for synthesis and (optionally) simulation and help content.

Generation of Operator Prototype: The VA Custom Operator Functionality lets you create an operator prototype which can immediately be used for instantiating the operator in Visual Applets. For this operator prototype a black box interface and an RTL level simulation entity for emulating the communication ports of the generated operator interface can be exported. Then you can start coding (i.e., implementing your HDL code complying with the interface of the black box) and simulating your Custom Operator design. The resulting FPGA design you then synthesize to an EDIF or NGC netlist. Optionally, you add a constraints file, create a dynamic link library for VisualApplets high-level simulation, and write HTML documentation for the VisualApplets GUI.

Completing the operator definition: The VA Custom Operator Functionality lets you specify the netlist, simulation library and documentation files . Supplemented with these files the operator is ready for use immediately.

1.3 Operator Types

VisualApplets knows different types of operators and ports, depending on the underlying flow control mechanism. Operators may be of type O or type M.

Custom Operators are always of type M.

	Custom Operator Type: M Custom Operators are always of type M.

1.4 Synchronous and Asynchronous Operator Ports

Operator ports can be synchronous or asynchronous. Being synchronous in VisualApplets basically means that data of several ports is transferred synchronously, whereas ports which are asynchronous to each other support non-aligned communication patterns.

Ports are only synchronous if they have a common M-source, or if they are sourced from a SYNC module; any constellation of O-operators may be between that source and the ports.

Depending on the relation of the operator input ports to each other, we differentiate between the following options:

1. **Synchronous inputs**: All input ports are synchronous to each other. There is one output port.
2. **Asynchronous inputs**: Some of the input ports are asynchronous to each other and all outputs are synchronous to each other.

Operators with asynchronous outputs are not allowed. Operators with synchronous inputs may only have a single output. If more than one output is required, the inputs must be declared as being asynchronous.

Defining Multiple Outputs

If you want to create an operator with multiple outputs, you need to declare its inputs to be asynchronous. Multiple outputs are always synchronous.

Examples for both classes (built-in Visual-Applets operators):

RemoveImage	M-Operator with synchronous inputs and one output
SYNC	M-Operator with asynchronous inputs and multiple, synchronous outputs

2 Interface Architecture

VisualApplets Custom Operator interfaces are designed for smoothly integrating your new operators so they behave inside VisualApplets like built-in operators. You can define any number of input and output ports for your custom operators.

Image In / Image Out: The *Image In* and *Image Out* ports may support multiple image formats. They are driven by simple-to-use FIFO interfaces. The FIFOs reside in the VA part of the custom operator, so that you only need to implement a flow control, but not the FIFO.

Memory ports: You also can define any number of memory ports. They also use FIFOs residing in the VA part of the custom operator.

GPIO ports: In addition to the image ports, you can define general-purpose I/O ports, e.g., for communicating asynchronous signals to the operator.

Registers: To allow the final user of your operator to configure the operator and to get access to status information, you can define any number of write and read registers.

Clock: The ports for receiving clock pulses are set up automatically for every custom operator.

Reset/Enable port: The ports for receiving reset or enable commands are set up automatically for every custom operator.

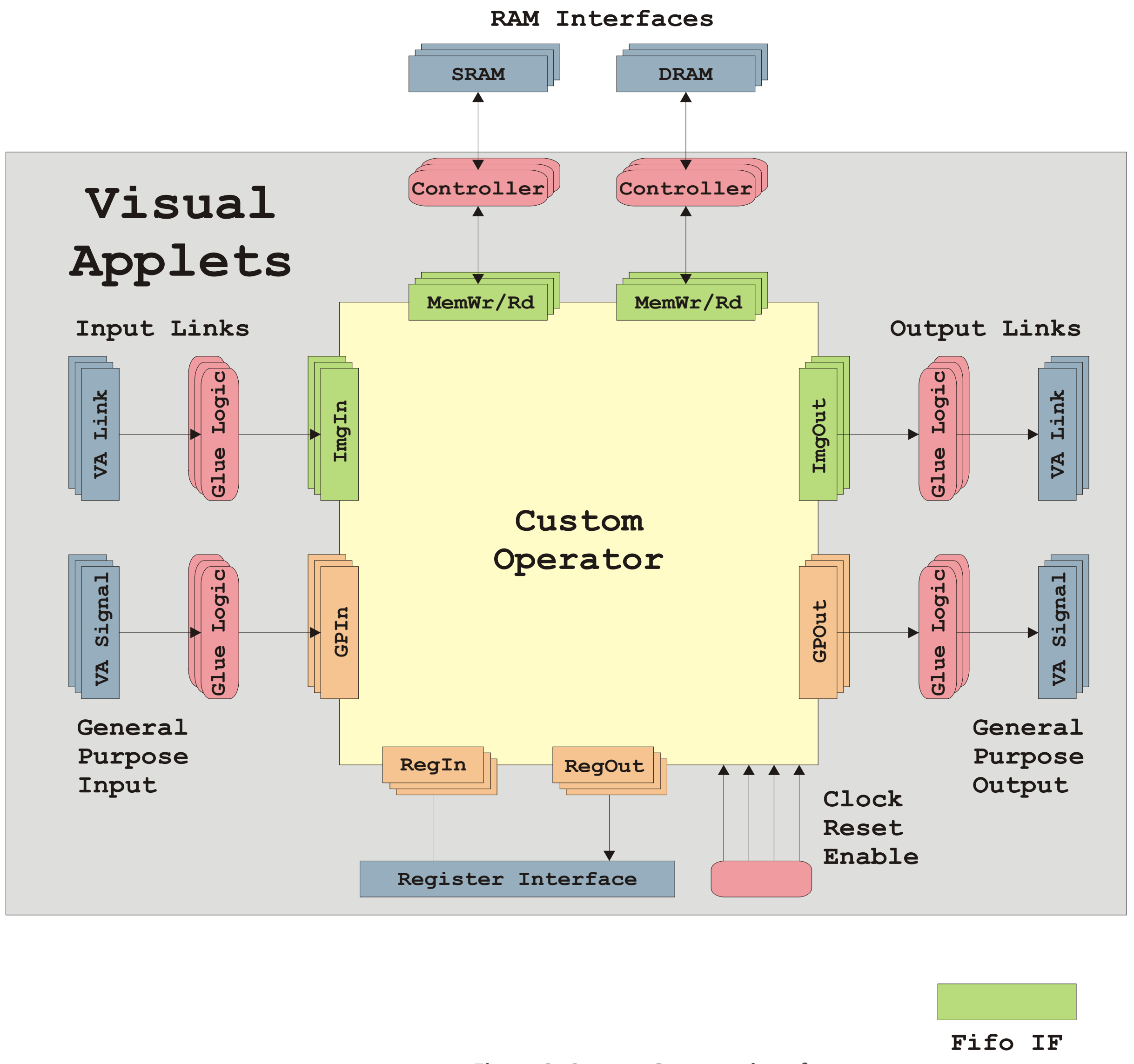
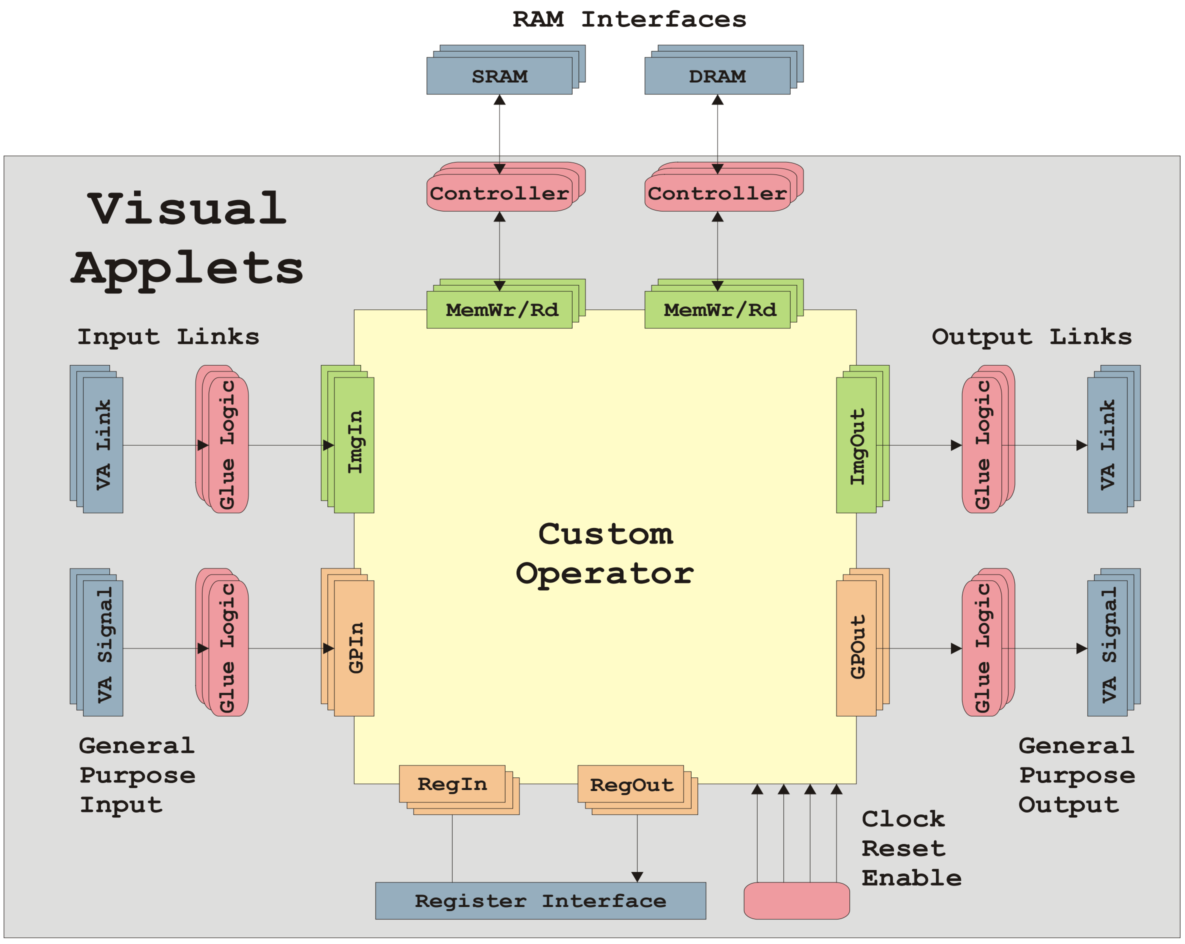


Figure 2: Custom Operator interfaces.

The following sections describe the different types of interfaces shown in figure 2 in detail.

2.1 Clock Interface

VisualApplets connects two clock inputs – the design clock and a second clock synchronous to the design clock but with double frequency. All interfaces except the memory interface must be synchronous to the design clock. The memory interface may be configured using the design clock or the double frequency clock for the read and/or write interface.



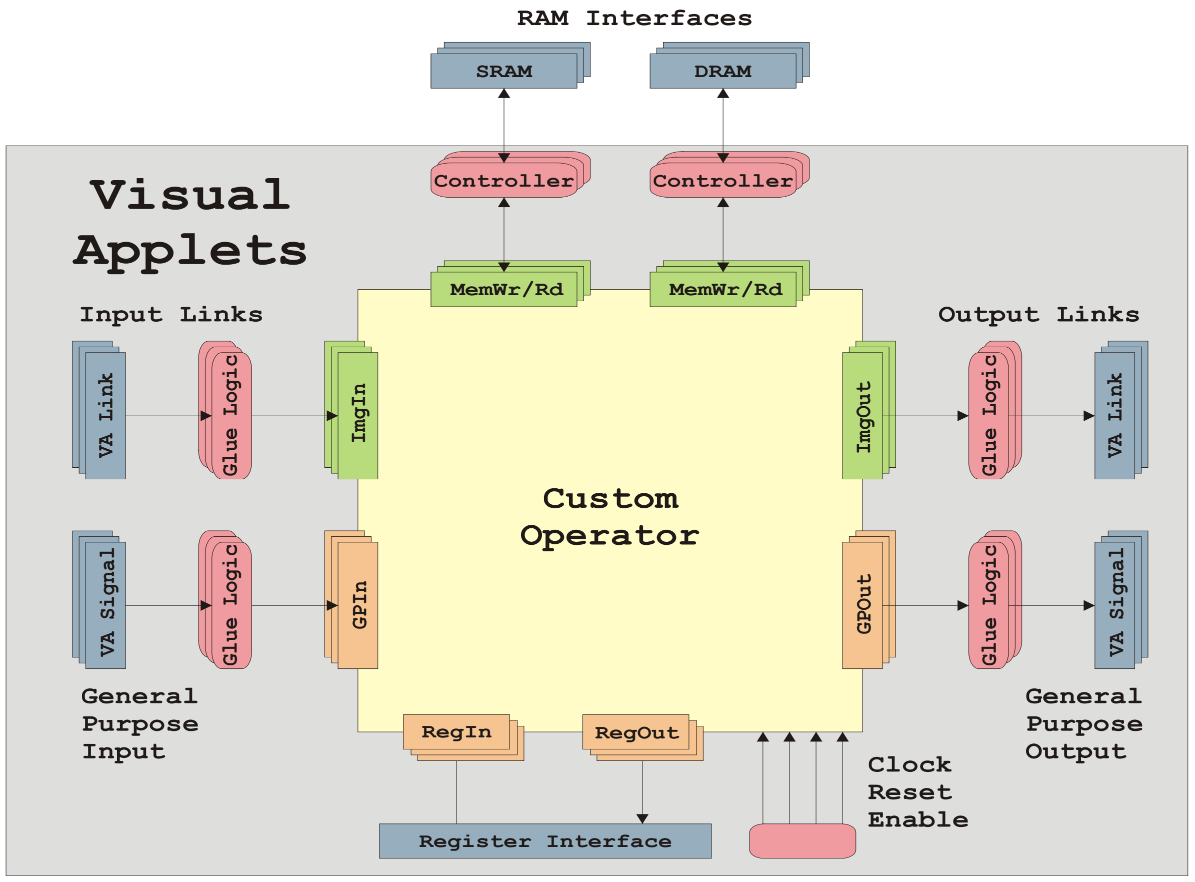
2.2 Reset and Enable

The Reset and Enable inputs are driven by the according “process enable” and “process reset” signals of the VA-process where the operator is instantiated. Make sure you implement the following behavior as reaction to these signals into your operator:

- Assertion of Reset puts the operator in its init state.
- Assertion of Enable starts processing.
- Deactivating Enable stops processing.
- (When Enable=0, the output FIFOs of the operator are not read. Depending on the state of the image processing pipeline some data may still be written to the input ports but the flow control safely prevents that any FIFO content gets corrupted.)
- Reset is only asserted when Enable=0

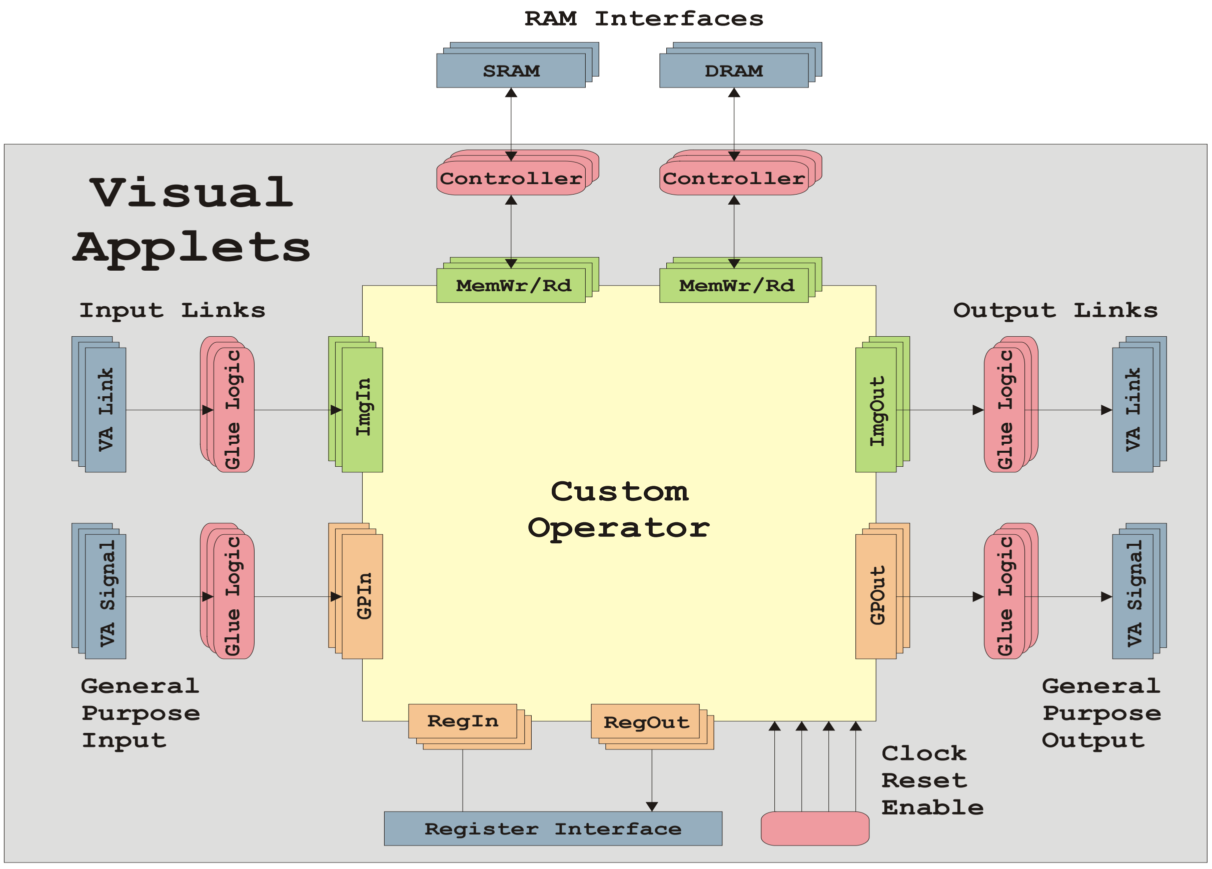
The following behavior to these signals is implemented in the VA part of the custom operator:

- Reset will empty all port interface FIFOs.
- Reset and Enable have no effect on the parameters of the operator.
- Reset and Enable have no effect on the GPIO interface of the operator.

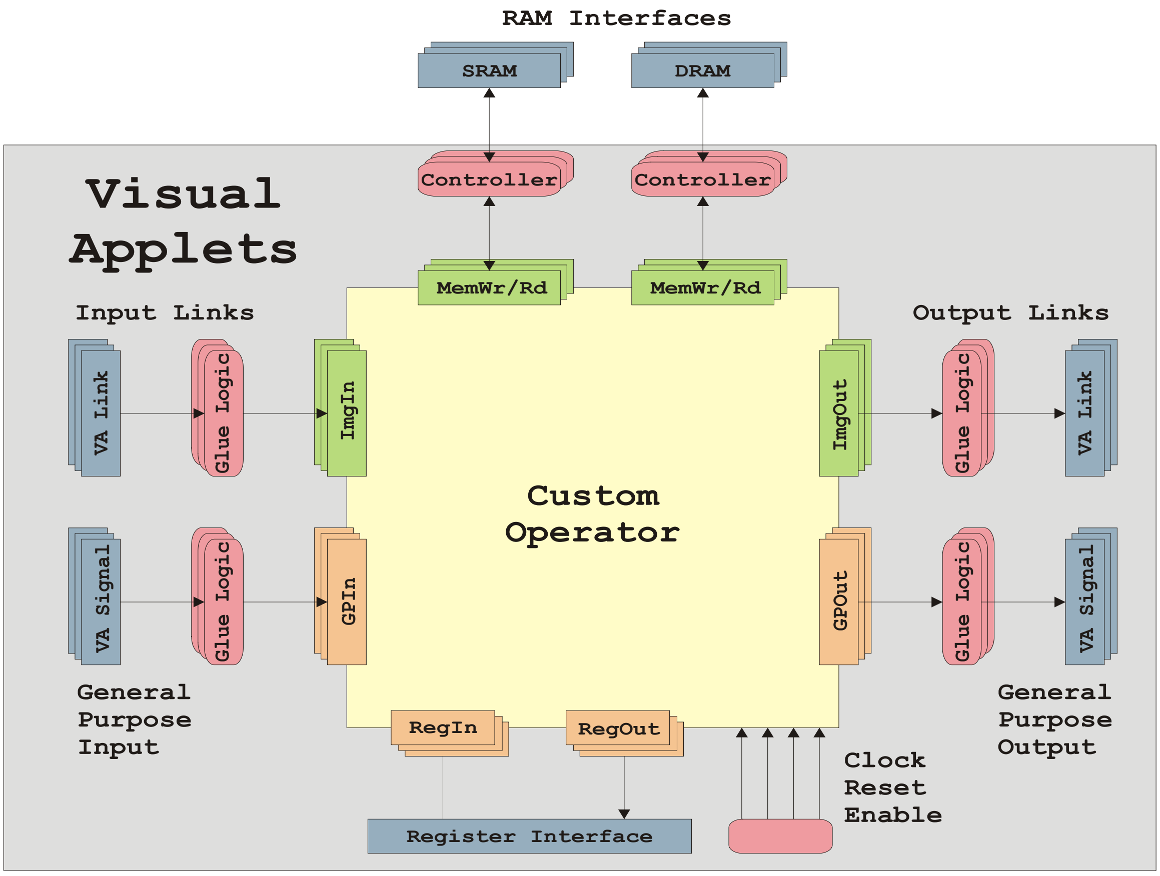

RAM Interfaces
SRAM
DRAM
Controller
Controller
MemWr/Rd
MemWr/Rd
Visual Applets
Input Links
VA Link
Glue Logic
ImgIn
VA Signal
Glue Logic
GPIn
General Purpose Input
Custom Operator
Output Links
ImgOut
Glue Logic
VA Link
GPOut
Glue Logic
VA Signal
General Purpose Output
RegIn
RegOut
Register Interface
Clock
Reset
Enable

2.3 Register Interface

For communicating operator parameters and status, the Custom Operator may be supplied with an arbitrary number of VisualApplets parameters. Each of the parameters translates to a separate register port of the Custom Operator. VisualApplets cares for dispatching the accesses to and from the operator registers.



2.4 Interfaces for Image Data



2.4.1 Image Protocols

You can define the image protocols that will be supported by the image in and image out ports of your custom operator. The future user of your operator will then be able to select from the list of image protocols you provide.

VisualApplets offers the following image formats to be supported by your operator's ImgIn and ImgOut ports:

- **gray*XxP***: gray image with ***X*** bits per pixel and parallelism ***P***
- **rgb*YxP*:** color image with ***Y***/3 bits per color component (red, green, blue) and parallelism ***P***
- **hsi*YxP*:** color image with ***Y***/3 bits per color component (HSI color model) and parallelism ***P***
- **hsl*YxP*:** color image with ***Y***/3 bits per color component (HSL color model) and parallelism ***P***
- **hsv*YxP*:** color image with ***Y***/3 bits per color component (HSV color model) and parallelism ***P***

- **yuv*YxP*:** color image with ***Y***/3 bits per color component (YUV color model) and parallelism ***P***
- **ycrcb*YxP*:** color image with ***Y***/3 bits per color component (YCrCb color model) and parallelism ***P***
- **lab*YxP*:** color image with ***Y***/3 bits per color component (LAB color model) and parallelism ***P***
- **xyz*YxP*:** color image with ***Y***/3 bits per color component (XYZ color model) and parallelism ***P***

Additionally, the image dimension and the information whether pixel components are signed or unsigned can be coded by optional suffixes.

The pixel data width ***X*** is limited to 64 bit**.** The width ***Y*** must be a multiple of 3 and is limited to 63 bit. The parallelism ***P*** defines the number of pixels which are contained in a single data word at the interface port. It must be chosen from following set of allowed values: ***P*** = {1, 2, 4, 8, 16, 32, 64}. Packing of image data into words of a given interface width ***N*** must follow certain rules:

- The data of all ***P*** pixels must fit in a single word of length ***N***. The data is stored LSB aligned which means that for a pixel width ***Z*** (***Z***=***X*** for grey, ***Z***=***Y*** for color) data is distributed as follows: Pixel[0]->Bits[0..***Z***-1] .. Pixel[***P***-1]->Bits[(***P***-1)****Z***..***P*******Z***-1].
- For RGB images the three color components are packed LSB aligned into a sub word[0..Y-1] in the following order: red uses the bits [0..***Y***/3-1], green the bits [Y/3..2*Y/3-1] and blue the bits [2*Y/3..3*Y/3-1].
- For YUV color images the same rules than for RGB applies where Y takes the role ofred, U that of green and V the role of blue.
- For HSI color images the same rules than for RGB applies where H takes the role ofred, S that of green and I the role of blue.
- For LAB color images the same rules than for RGB applies where L takes the role ofred, A that of green and B the role of blue.
- For XYZ color images the same rules than for RGB applies where X takes the role ofred, Y that of green and Z the role of blue.

In VisualApplets, any link carries the properties maximum image width and maximum image height. VisualApplets lets you define optional constraints for the maximum width and height for any of the supported image protocols of the custom operator separately.

For an image interface port, you define a list of allowed image protocols. This list makes up a subset of the possible VisualApplets image formats (see above). A format can be described by the following properties:

- Data type uint or int
- Pixel data bit width N = [1..64]
- Gray or color format (single or three data components with aggregated width N)
- Flavor of color format (RGB, HSI, HSL, HSV, YUV, YCrCb, LAB, XYZ)
- 2D, 1D, or 0D
- Parallelism P = {1,2,4,8,16,32,64}

Implicitly it is assumed that the kernel size is 1x1. The listed formats are numbered starting from zero and therewith define an ID.

When working with the final operator in VisualApplets, the user of your operator can select any of the formats you list here for the image communication port in question. According to the selection made by the VA user, the corresponding ID will be output to the related Custom Operator port. This enables the Custom Operator to adapt its behavior to the selected format.

2.4.2 Image Input Ports

Image input ports allow to communicate image data from the VisualApplets process to the custom operator. These ports are named ImgIn. If you designed the custom operator to support configuration of its input channel(s) (see section 2.4.1), several different protocols can be driven through a single port selected by the corresponding format parameter within VisualApplets. The interface basically consists of a FIFO and a parameter register providing an ID for the actually used data format. The Custom Operator must care for reading the FIFO and interpreting the image data according to the protocol of the selected image format. The operator must guarantee a correct flow control according to the status pins providing information about the filling state of the FIFO, i.e., no data may be read when the FIFO is empty.

For an image interface port, a list of allowed image formats needs to be defined. This list makes up a subset of possible VisualApplets image formats (see section 2.4.1) where a format can be described by the following properties:

- Data type uint or int
- Pixel data bit width N = [1..64]
- Gray or color format (single or three data components with aggregated width N)
- Flavor of color format (RGB, HSI, HSL, HSV, YUV, YCrCb, LAB, XYZ)

- 2D, 1D, or 0D
- Parallelism P = {1,2,4,8,16,32,64}

Implicitly it is assumed that the kernel size is 1x1. The listed formats are numbered starting from zero and therewith define an ID.

When working with the final operator in VisualApplets, the user can select any of the formats you list here for the concerning image communication port. According to the selection made by the VA user, the corresponding ID will be output to the related Custom Operator port. This enables the Custom Operator to adapt its behavior to the selected format.

2.4.3 Image Output Ports

Image output ports allow communicating image data from the Custom Operator to the VisualApplets process. These ports are named ImgOut. If you designed the custom operator to support appropriate configuration of its output channel(s) (see section 2.4.1), several different protocols can be driven through a single port selected by the corresponding format parameter within VisualApplets. The interface basically consists of a FIFO and a parameter register providing an ID for the actually used data format. The Custom Operator must care for feeding the FIFO with image data according to the protocol of the selected image format. The operator must guarantee a correct flow control according to the status pins providing information about the filling state of the FIFO, i.e., no data may be written when the FIFO is full.

For an image interface port, a list of allowed image formats needs to be defined. This list makes up a subset of possible VisualApplets image formats (see section 2.4.1) where a format can be described by the following properties:

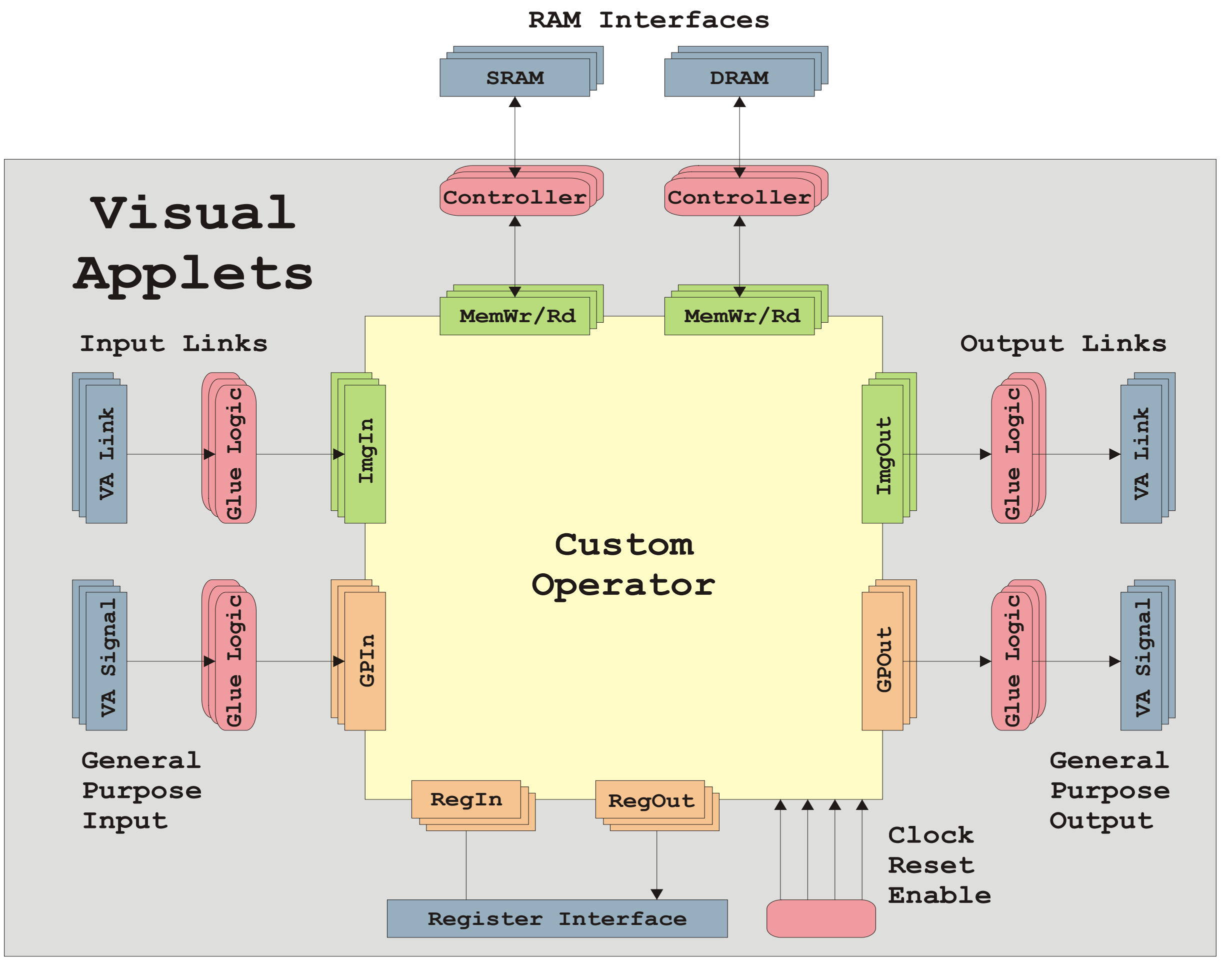
- Data type uint or int
- Pixel data bit width N = [1..64]
- Gray or color format (single or three data components with aggregated width N)
- Flavor of color format (RGB, HSI, HSL, HSV, YUV, YCrCb, LAB, XYZ)
- 2D, 1D, or 0D
- Parallelism P = {1,2,4,8,16,32,64}

Implicitly it is assumed that the kernel size is 1x1. The listed formats are numbered starting from zero and therewith define an ID.

When working with the final operator in VisualApplets, the user can select any of the formats you list here for the concerning image communication port. According to the selection made by the VA user, the corresponding ID will be output to the related custom operator port. This enables the custom operator to adapt its behavior to the selected format.

2.5 General purpose I/O

The General Purpose I/O interface allows connecting dedicated signal pins of the custom operator. Every GPIO port maps to a pin of the custom operator which is either an input or an output. Bidirectional pins are not supported. In VisualApplets, the corresponding operator ports are of type SIGNAL.

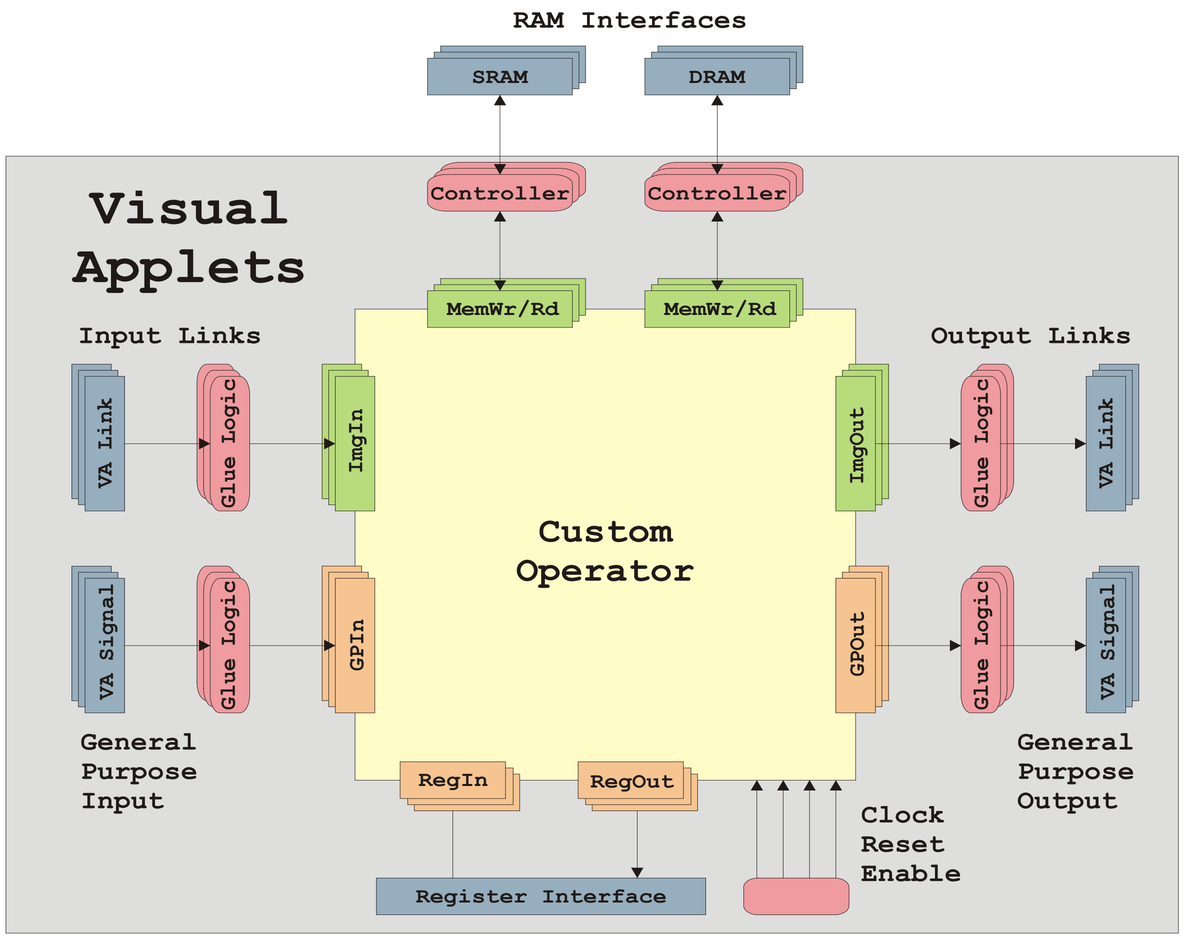


Bidirectional Pins not Supported

The GPIO pins must be either an input or an output. Bidirectional pins are not supported.

2.6 Memory Interface

A custom operator may be set up for accessing one or more banks of memory. The concerning memory ports have a FIFO like interface for write and read commands. The FIFOs reside in the VA part of the custom operator, so that you only need to implement a flow control, but not the FIFO. The timing of forwarding the FIFO content to the memory controller attached to the custom operator is fully controlled by VisualApplets.



3 Defining an Individual Custom Operator via GUI

First of all, you need to enter some details describing your new custom operator.

VisualApplets uses these details for generating a VHDL black box for your custom operator and an according test bench for simulation.

You enter the configuration for your individual custom operator via the VisualApplets GUI. VisualApplets makes the specified operator available for use in a design immediately, even if the operator specification is incomplete concerning netlist, simulation model and documentation.

Custom Library File

A custom library with all contained operators is stored as one single <LibaryName>.vl file. <LibaryName> is the name of the custom library.
This file can be distributed and directly applied in VisualApplets. It simply needs to be copied into the Custom Library directory which is specified in the VisualApplets settings.

Operator Configuration in XML Format

VisualApplets stores the custom operator specification in XML format. You can export the XML content from the custom library to a file, e.g., for handling it in a version control system. On the other hand you can import the XML for adding a custom operator (see section 10.3). You do not need to know how this XML file looks like. However, if you want to have a look, refer to the Appendix, section 12.1.

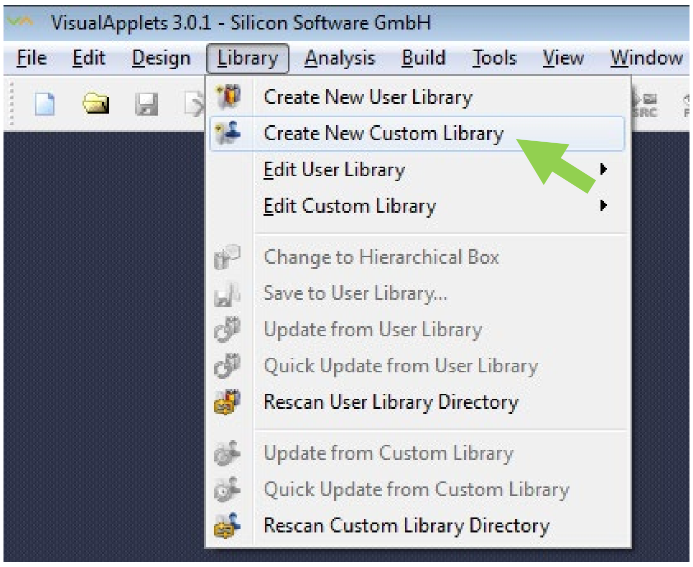
3.1 Creating a New Custom Library

Before you can start to define a new custom operator, you need to create a custom library where the new operator belongs to.

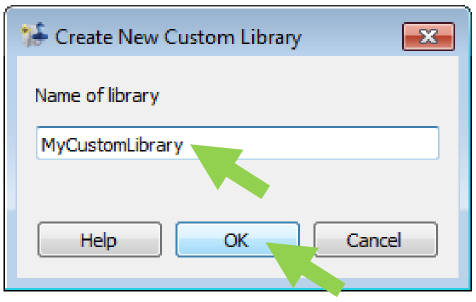
If you already have a custom library available where the new custom operator will belong to, skip this section and proceed with section 3.2.

To create a new custom library:

1. In menu *Library*, select menu item *Create New Custom Library*.

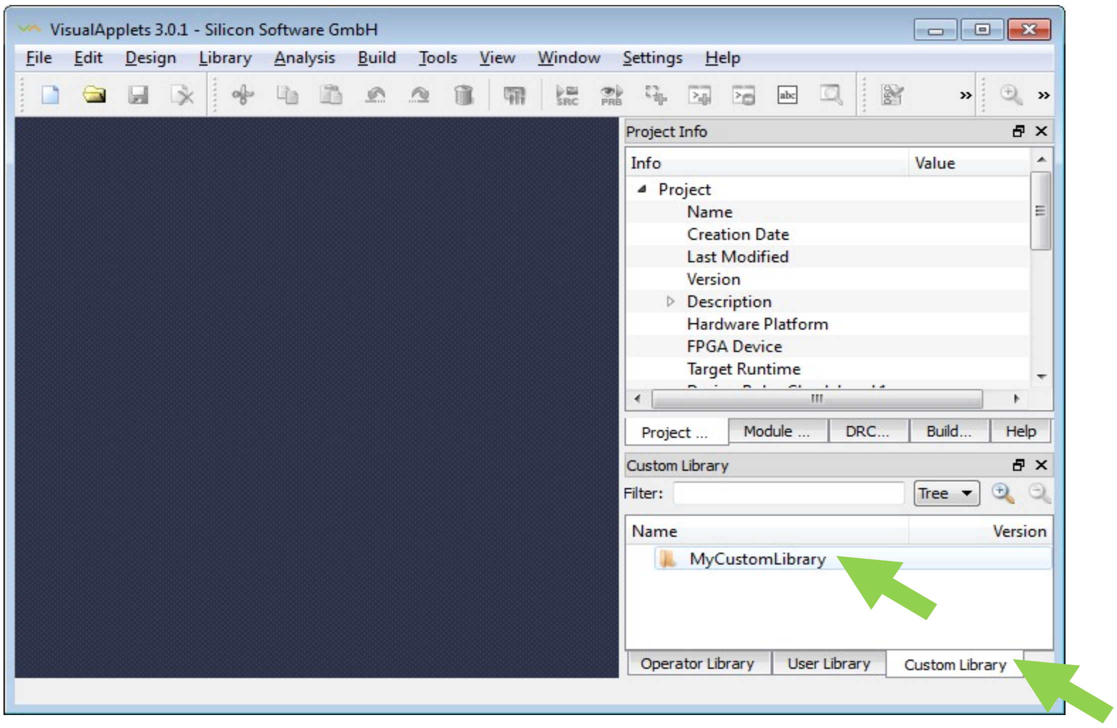

VisualApplets 3.0.1 - Silicon Software GmbH
File
Edit
Design
Library
Analysis
Build
Tools
View
Window
Create New User Library
Create New Custom Library
Edit User Library
Edit Custom Library
Change to Hierarchical Box
Save to User Library...
Update from User Library
Quick Update from User Library
Rescan User Library Directory
Update from Custom Library
Quick Update from Custom Library
Rescan Custom Library Directory

2. In the dialog that opens, enter a name for your new library and confirm with OK:



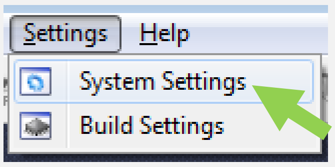
!	**Comply with Conventions for Valid C Identifiers** When defining the library name in the VA GUI, make sure you conform to the conventions for valid C identifiers.

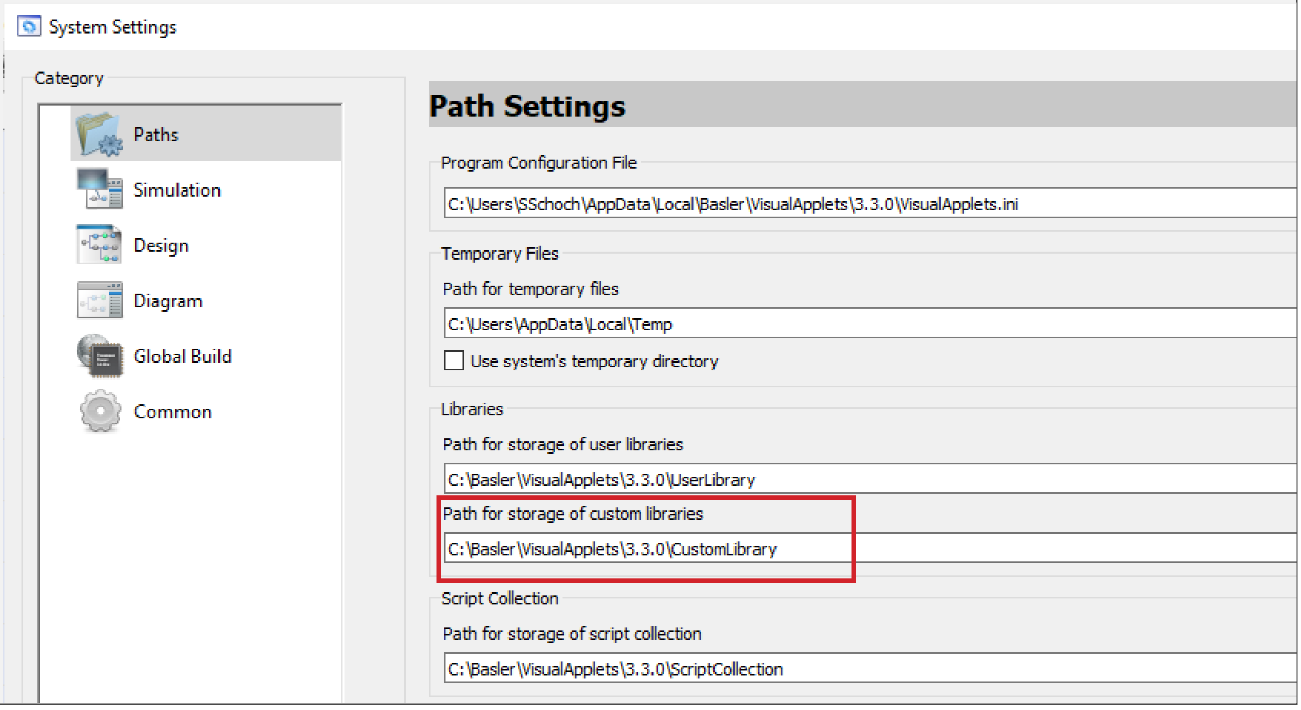
Now, the new custom library is created. You can see it in the operator panel under the *Custom Library* tab:


VisualApplets 3.0.1 - Silicon Software GmbH
File Edit Design Library Analysis Build Tools View Window Settings Help
Project Info
Info
Value
Project
Name
Creation Date
Last Modified
Version
Description
Hardware Platform
FPGA Device
Target Runtime
Project ... Module ... DRC... Build... Help
Custom Library
Filter:
Tree
Name
Version
MyCustomLibrary
Operator Library User Library Custom Library

Specifying a Custom Library Directory

For creating a new custom library, you may need to specify a directory where all custom-library-related files are stored. You do this under *Settings -> System Settings -> Paths*:





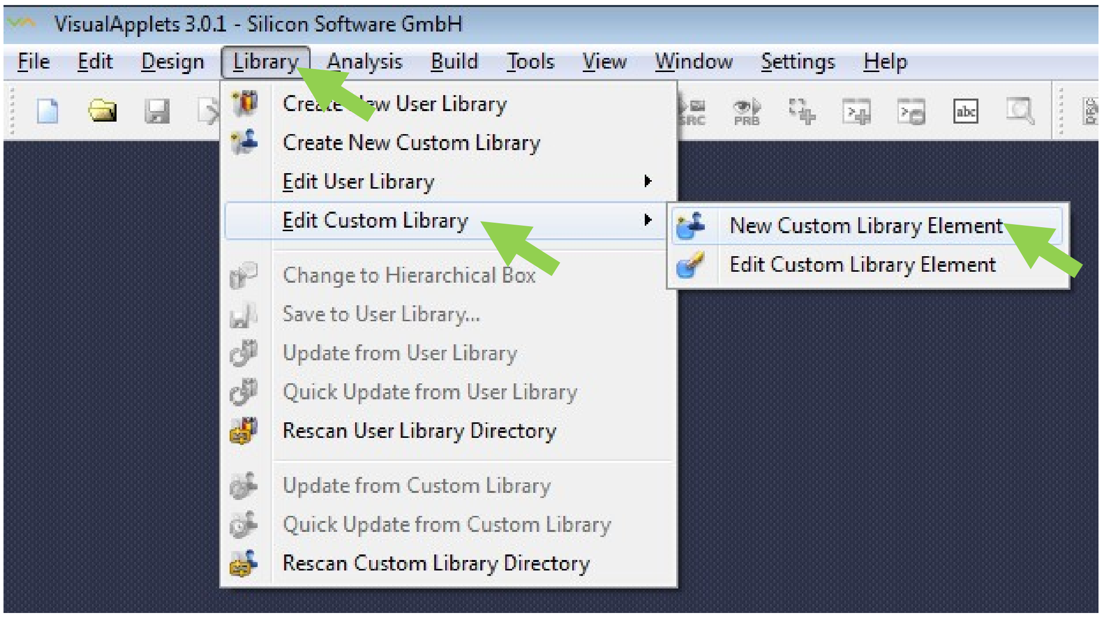
Program Configuration File
C:\Users\SSchoch\AppData\Local\Basler\VisualApplets\3.3.0\VisualApplets.ini
Temporary Files
Path for temporary files
C:\Users\AppData\Local\Temp
☐ Use system's temporary directory
Libraries
Path for storage of user libraries
C:\Basler\VisualApplets\3.3.0\UserLibrary
Path for storage of custom libraries
C:\Basler\VisualApplets\3.3.0\CustomLibrary
Script Collection
Path for storage of script collection
C:\Basler\VisualApplets\3.3.0\ScriptCollection

3.2 Creating a New Custom Operator

To define a new custom operator:

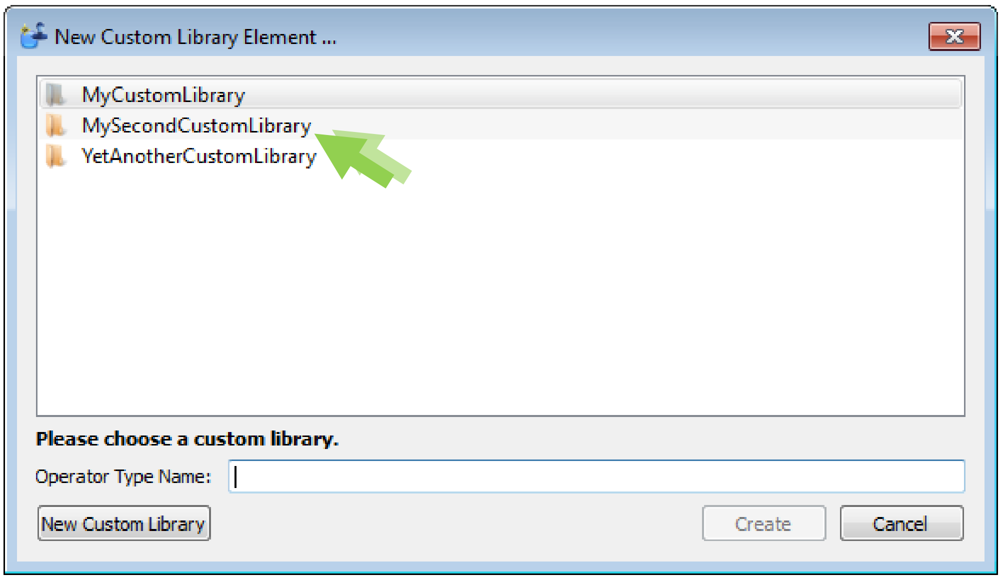
1. In menu *Library*, select menu item *Edit Custom Library*.

2. In the submenu that opens, select *New Custom Library Element*.

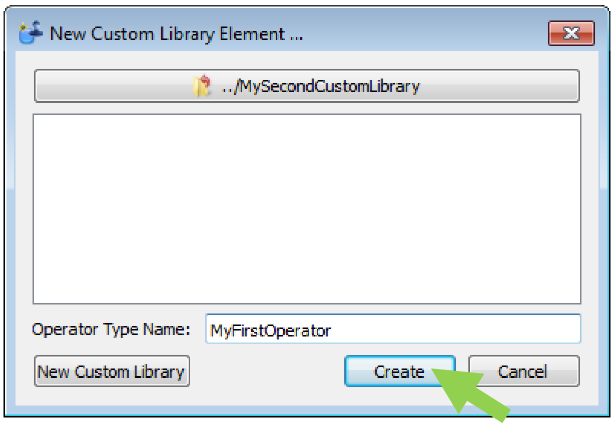

VisualApplets 3.0.1 - Silicon Software GmbH
File
Edit
Design
Library
Analysis
Build
Tools
View
Window
Settings
Help
Create New User Library
Create New Custom Library
Edit User Library
Edit Custom Library
Change to Hierarchical Box
Save to User Library...
Update from User Library
Quick Update from User Library
Rescan User Library Directory
Update from Custom Library
Quick Update from Custom Library
Rescan Custom Library Directory
New Custom Library Element
Edit Custom Library Element

In the window that opens:

3. Select a custom library via double-click on the library name.



4. Enter a name for your custom operator:



Operator Type Name: MyFirstOperator

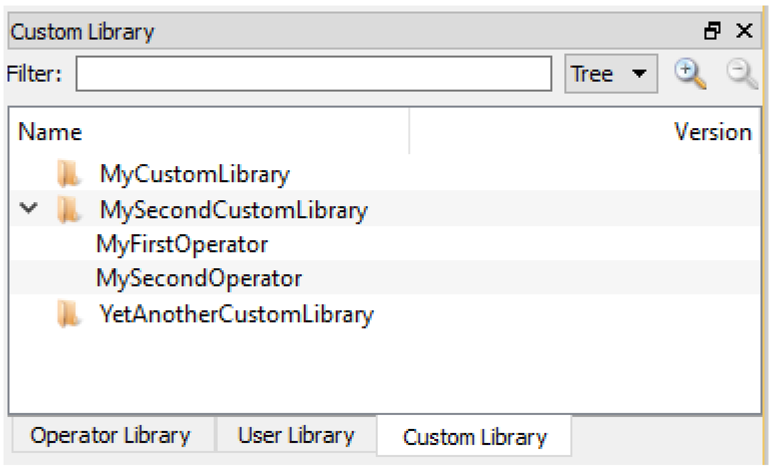
!	**Comply with VHDL Naming Conventions** When defining the operator name in the VA GUI, make sure you conform to the VHDL naming conventions. VHDL valid names are defined as follows: “A valid name for a port, signal, variable, entity name, architecture body, or similar object consists of a letter followed by any number of letters or numbers, without space. A valid name is also called a named identifier. VHDL is not case sensitive. However, an underscore may be used within a name, but may not begin or end the name. Two consecutive underscores are not permitted.“

5. Click the **Create** button.

Dialog *Edit Custom Operator* opens. Here, you can define your custom operator.

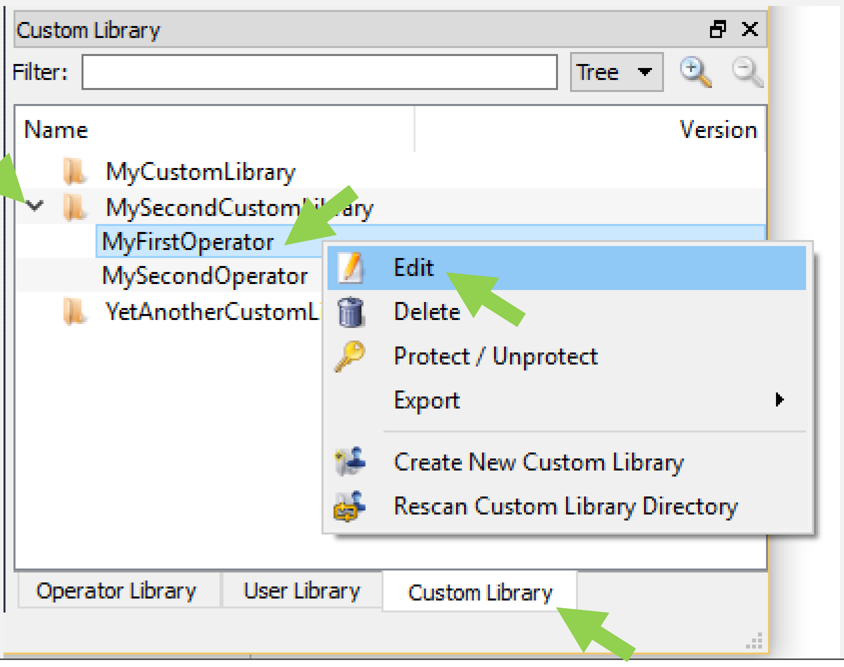
6. Click the **Save** button.

Now, your new custom operator is visible under the custom library it belongs to:



Interrupting your Work

Once you have created a new custom operator and saved it to VisualApplets, you can interrupt your work and proceed any time. To proceed, you go to the *Custom Library* tab, open the library, right-click on the operator name, and from the sub menu, select **Edit**.



Use Operator Template Instead

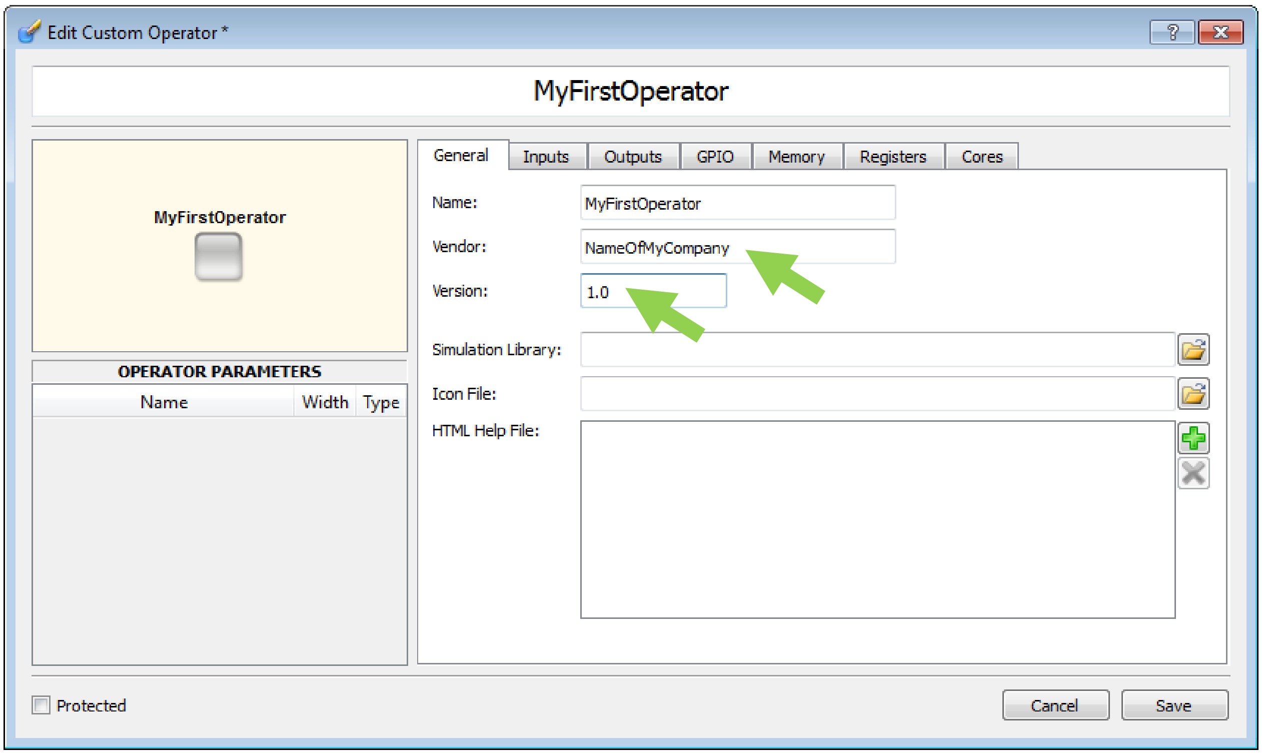
Alternatively, you can use the custom operator template provided in your VisualApplets installation to define new custom operators. How to use the template, see section 11.2.

3.3 Defining Basic Information about Custom Operator

In a first step, you define your custom operator's interface.

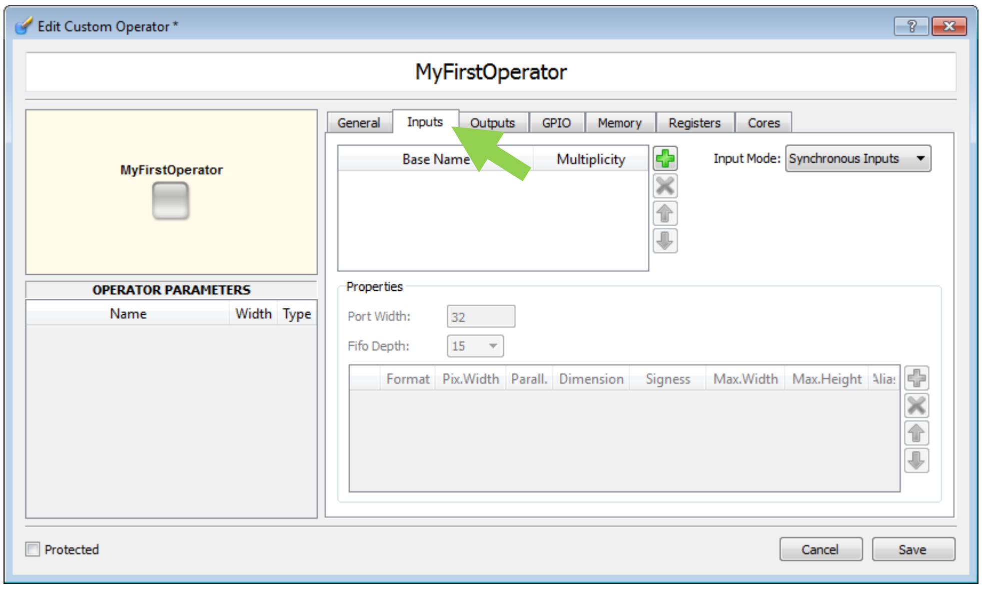
1. Provide your vendor name. You can enter any string. This information is intended for operator identification by the user.

2. Provide a version number for your operator, e.g., version 1.0. You can enter any number but you should comply with the version scheme "<major>.<minor>". This information is intended for operator version identification by the user.



Name: MyFirstOperator
Vendor: NameOfMyCompany
Version: 1.0
Simulation Library:
Icon File:
HTML Help File:

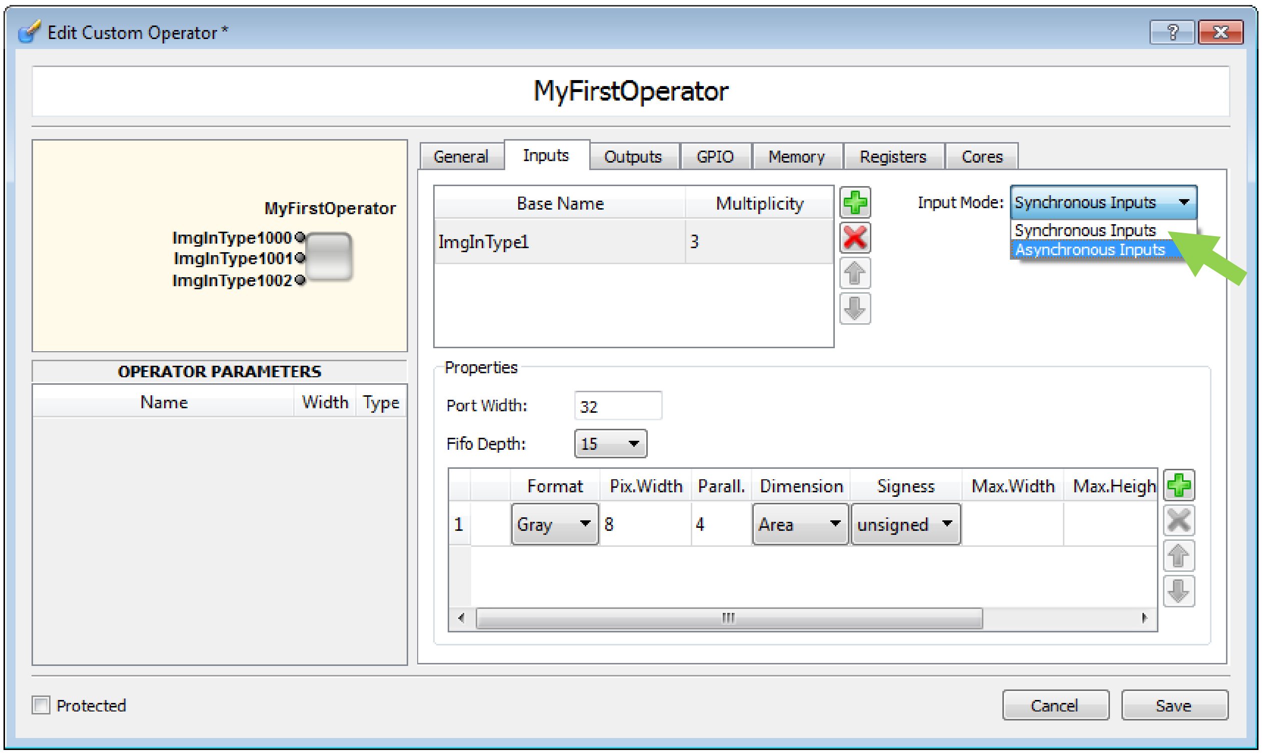
3. Proceed to the tab *Inputs*.



3.4 Defining the Image Input Ports

Under tab *Inputs*, you describe the properties of the image input ports.

1. First of all, you define the input mode of your custom operator's ImgIn ports:



Base Name	Multiplicity
ImgInType1	3

Port Width: 32

Fifo Depth: 15

	Format	Pix.Width	Parall.	Dimension	Signess	Max.Width	Max.Heigh
1	Gray	8	4	Area	unsigned		

Synchronous and Asynchronous Operator Ports

Operator ports can be synchronous or asynchronous. Being synchronous in VisualApplets basically means that data of several ports is transferred synchronously, whereas ports which are asynchronous to each other support non-aligned communication patterns.
Ports are only synchronous if they have a common M-source, or if they are sourced from a SYNC module; any constellation of O-operators may be between that source and the ports.
Depending on the relation of the operator input ports to each other, we differentiate between the following options:

1. **Synchronous inputs**: All input ports are synchronous to each other. There is one output port.
2. **Asynchronous inputs**: Some of the input ports are asynchronous to each other and all outputs are synchronous to each other.

Operators with asynchronous outputs are not allowed. Operators with synchronous inputs may only have a single output. If more than one output is required, the inputs must be declared as being asynchronous.

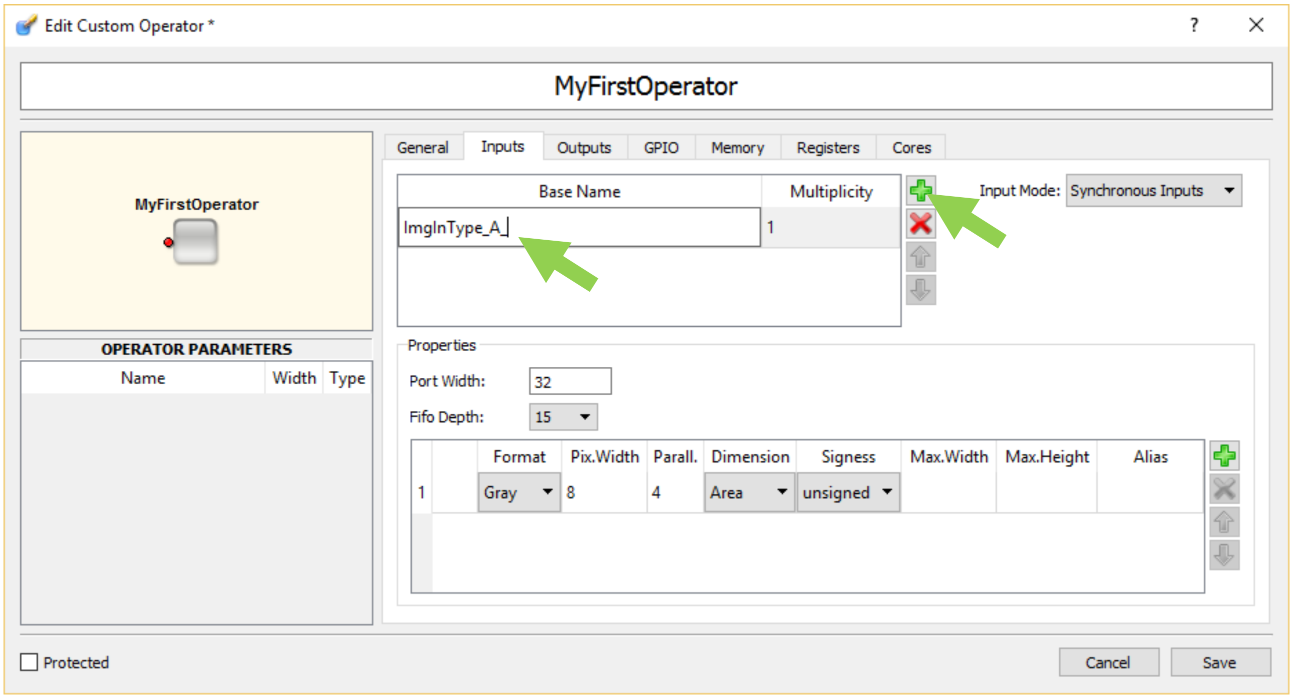
If you want to create an operator with multiple outputs, you need to declare its inputs to be asynchronous. Multiple outputs are always synchronous.

Examples for both classes (built-in Visual-Applets operators):

	RemoveImage	M-Operator with synchronous inputs and one output
	SYNC	M-Operator with asynchronous inputs and multiple, synchronous outputs

Second, you can define one or more image input ports (ImgIn). Each ImgIn port may be used as often as you specify.

2. Click on the plus button to create a first image in (ImgIn) port.
3. Give a name to the ImgIn port.



Base Name: ImgInType_A_ — Multiplicity: 1

Input Mode: Synchronous Inputs

Properties

Port Width: 32

Fifo Depth: 15

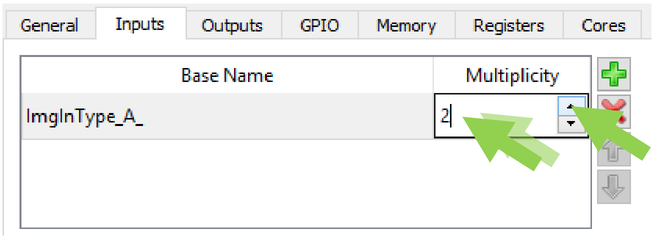
	Format	Pix.Width	Parall.	Dimension	Signess	Max.Width	Max.Height	Alias
1	Gray	8	4	Area	unsigned			

OPERATOR PARAMETERS

Name	Width	Type

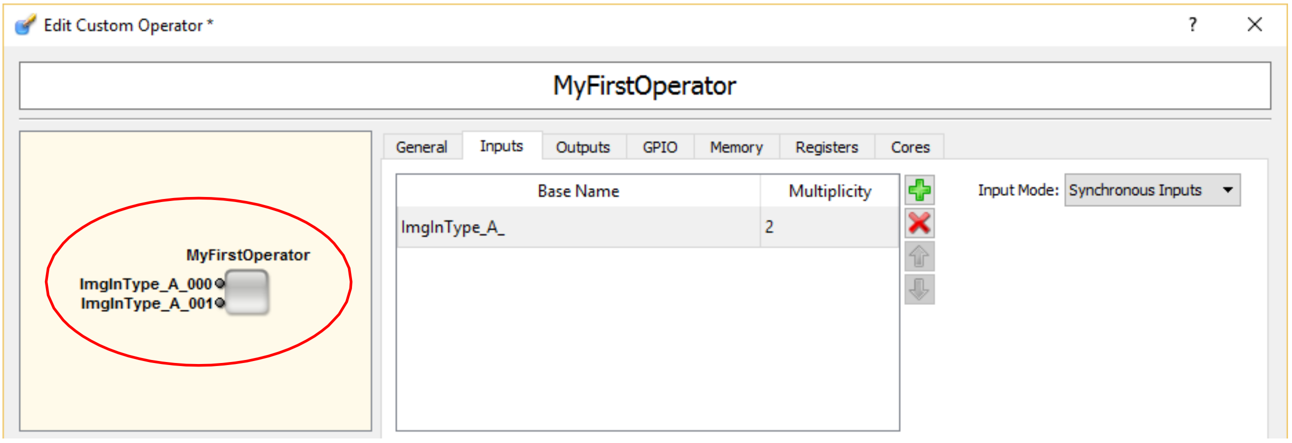
☐ Protected

4. Double-click in the field of the *Multiplicity* column to create an array of ports. Multiplicity >1 defines an array of ports with a name consisting of the base name and an index.



Base Name	Multiplicity
ImgInType_A_	2

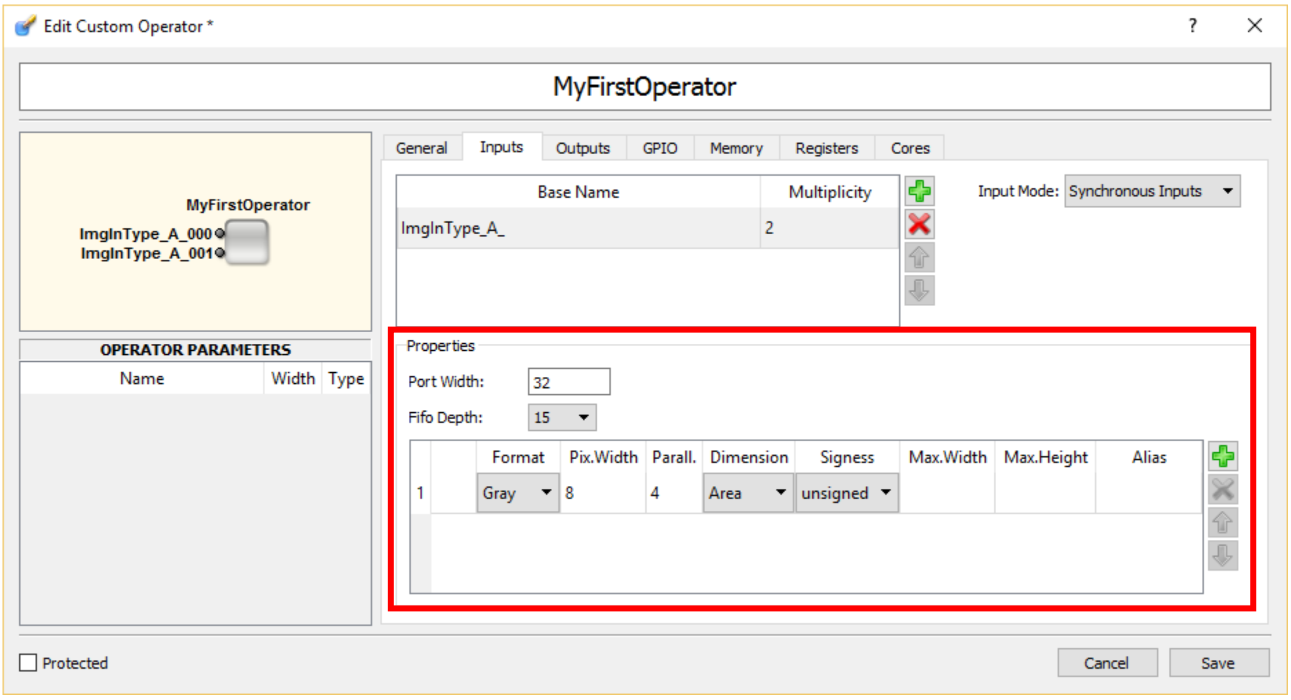
Immediately, the operator depiction in the program window displays the entered array of ImgIn ports:



Base Name	Multiplicity
ImgInType_A_	2

Input Mode: Synchronous Inputs

In the ***Properties*** panel, you specify the properties of the protocols that are supported by this ImgIn port.


Edit Custom Operator *

MyFirstOperator

Tabs: General, Inputs, Outputs, GPIO, Memory, Registers, Cores

Input Mode: Synchronous Inputs

Base Name	Multiplicity
ImgInType_A_	2

Properties

Port Width: 32

Fifo Depth: 15

	Format	Pix.Width	Parall.	Dimension	Signess	Max.Width	Max.Height	Alias
1	Gray	8	4	Area	unsigned			

OPERATOR PARAMETERS

Name	Width	Type

Protected

5. Under ***Port Width***, specify the width of the ImgIn port.

6. Under ***Fifo Depth***, specify the depth of the buffer FIFO for input data which at least needs to be provided by the VA core. The value must be a power of two minus 1 between 15 and 1023.

For an image interface port, you define a list of allowed protocols. A protocol can be described by the following properties:

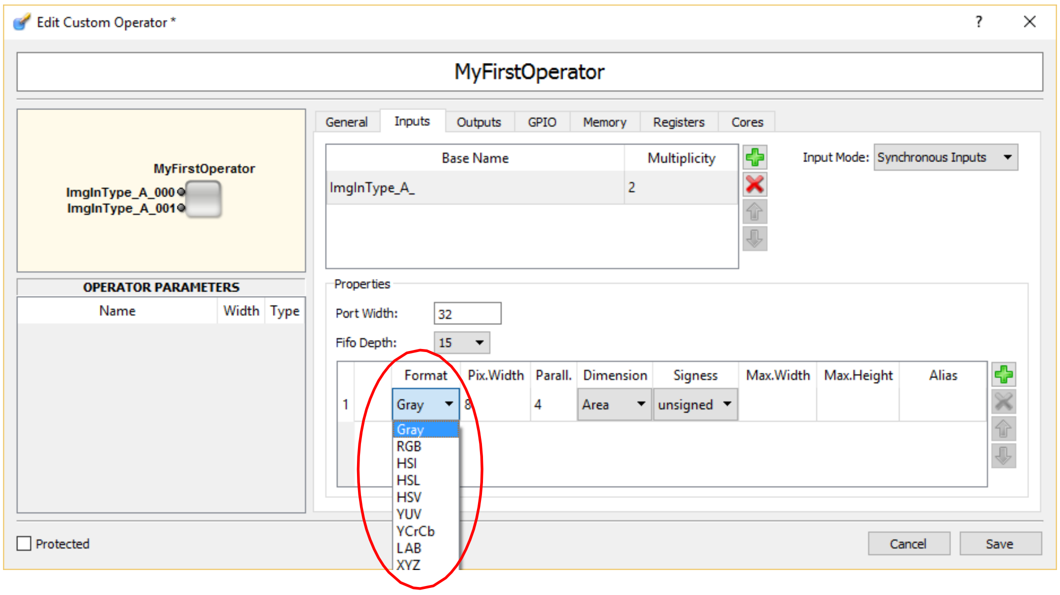
- Gray or color format (single or three data components with aggregated width N)
- Flavor of color format (RGB, HSI, HSL, HSV, YUV, YCrCb, LAB, XYZ)
- Pixel data bit width N = [1..64]
- Parallelism P = {1,2,4,8,16,32,64}
- 2D (Aray), 1D (Line), or 0D (Raw)
- Data type uint or int
- Max. image dimensions

Implicitly it is assumed that the kernel size is 1x1.

The listed protocols are numbered starting from zero and therewith define an ID (in the image below visible in the left hand column of the table in the *Properties* panel).

If you specify more than one protocol, you design the custom operator to support configuration of its input channel(s). In this case, several different protocols can be driven through a single port. The user of your custom operator can select the protocol he wants to use on a specific ImgIn port. According to the selection made by the VA user, the corresponding ID will be output to the related custom operator port. This enables the custom operator to adapt its behavior to the selected protocol.

7. Under ***Format***, specify the color format of the protocol.


Edit Custom Operator *

MyFirstOperator

Base Name	Multiplicity
ImgInType_A_	2

Input Mode: Synchronous Inputs

Properties

Port Width: 32

Fifo Depth: 15

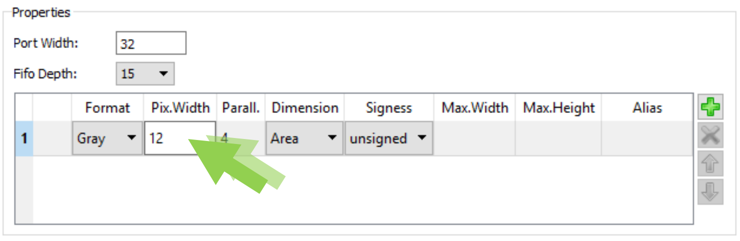
	Format	Pix.Width	Parall.	Dimension	Signess	Max.Width	Max.Height	Alias
1	Gray	8	4	Area	unsigned			

The following color formats are allowed:

- **gray*XxP***: gray image with ***X*** bits per pixel and parallelism ***P***
- **rgb*YxP*:** color image with ***Y***/3 bits per color component (red, green, blue) and parallelism ***P***
- **hsi*YxP*:** color image with ***Y***/3 bits per color component (HSI color model) and parallelism ***P***
- **hsl*YxP*:** color image with ***Y***/3 bits per color component (HSL color model) and parallelism ***P***
- **hsv*YxP*:** color image with ***Y***/3 bits per color component (HSV color model) and parallelism ***P***
- **yuv*YxP*:** color image with ***Y***/3 bits per color component (YUV color model) and parallelism ***P***
- **ycrcb*YxP*:** color image with ***Y***/3 bits per color component (YCrCb color model) and parallelism ***P***
- **lab*YxP*:** color image with ***Y***/3 bits per color component (LAB color model) and parallelism ***P***
- **xyz*YxP*:** color image with ***Y***/3 bits per color component (XYZ color model) and

parallelism ***P***

8. Double-click in the field of column ***Pix.Width*** and specify the pixel data width for the specific format:



Properties

Port Width: 32

Fifo Depth: 15

	Format	Pix.Width	Parall.	Dimension	Signess	Max.Width	Max.Height	Alias
1	Gray	12	4	Area	unsigned			

The value range of ***Pix.Width*** depends on your choice under ***Format***:

Gray: The pixel data width (in the following referred to as ***X***) is limited to 64 bit**.**

All color formats: The pixel data width (in the following referred to as ***Y***) must be a multiple of 3 and is limited to 63 bit.

9. Double-click in the field of column ***Parall.*** and specify the parallelism for the specific format.

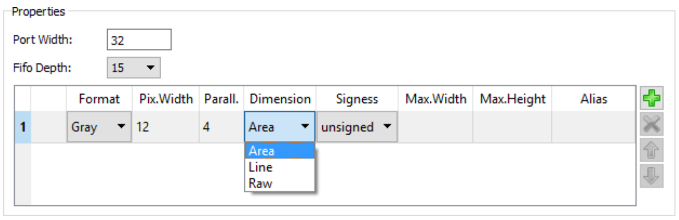
The parallelism defines the number of pixels which are contained in a single data word at the interface port. It must be chosen from following set of allowed values: ***P*** = {1, 2, 4, 8, 16, 32, 64}. Packing of image data into words of a given interface width ***N*** (specified under ***Port Width***) must follow certain rules:

- The data of all ***P*** pixels must fit in a single word of length ***N***. The data is stored LSB aligned which means that for a pixel width ***Z*** (***Z***=***X*** for grey, ***Z***=***Y*** for color) data is distributed as follows: Pixel[0]->Bits[0..***Z***-1] .. Pixel[***P***-1]->Bits[(***P***-1)****Z***..***P*******Z***-1].
- For RGB images the three color components are packed LSB aligned into a sub word[0..Y-1] in the following order: red uses the bits [0..***Y***/3-1], green the bits [***Y***/3..2****Y***/3-1] and blue the bits [2****Y***/3..3****Y***/3-1].
- For HSI color images the same rules than for RGB applies where H takes the role ofred, S that of green and I the role of blue.
- For HSL color images the same rules than for RGB applies where H takes the role ofred, S that of green and L the role of blue.
- For HSV color images the same rules than for RGB applies where H takes the role ofred, S that of green and V the role of blue.
- For YUV color images the same rules than for RGB applies where Y takes the role

ofred, U that of green and V the role of blue.

- For YCrCb color images the same rules than for RGB applies where Y takes the roleof red, Cr that of green and Cb the role of blue.
- For LAB color images the same rules than for RGB applies where L takes the role ofred, A that of green and B the role of blue.
- For XYZ color images the same rules than for RGB applies where X takes the role ofred, Y that of green and Z the role of blue.

10. Under ***Dimension***, specify if the protocol supports 2D (Area), 1D (Line), or 0D (Raw) images.



Properties

Port Width: 32

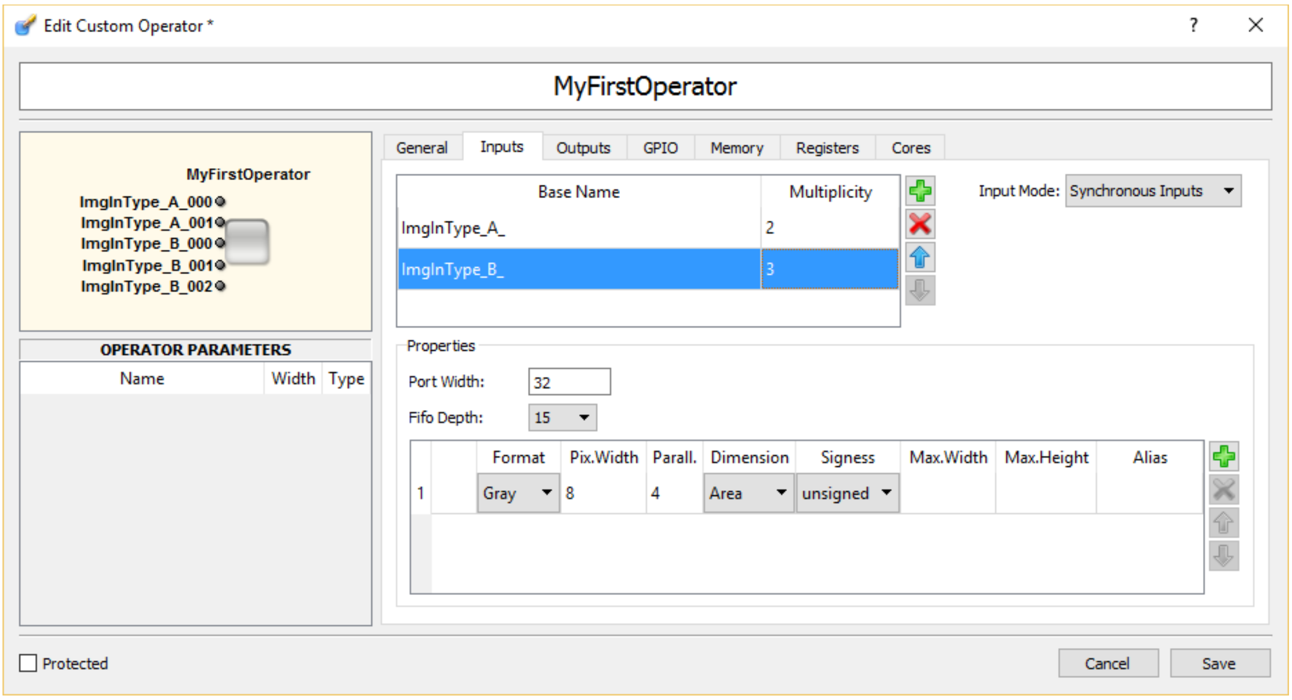
Fifo Depth: 15

	Format	Pix.Width	Parall.	Dimension	Signess	Max.Width	Max.Height	Alias
1	Gray	12	4	Area	unsigned			

11. ***Max.Width/Max.Height***: Using these optional fields you can define constraints for the image width and image height.

12. Repeat steps 7 to 11 to define as many protocols as you want the ImgIn port to support.

13. Repeat steps 2 to 12 to define as many ImgIn ports you want your custom operator to provide.



Edit Custom Operator *

MyFirstOperator

Base Name	Multiplicity
ImgInType_A_	2
ImgInType_B_	3

Input Mode: Synchronous Inputs

Properties

Port Width: 32

Fifo Depth: 15

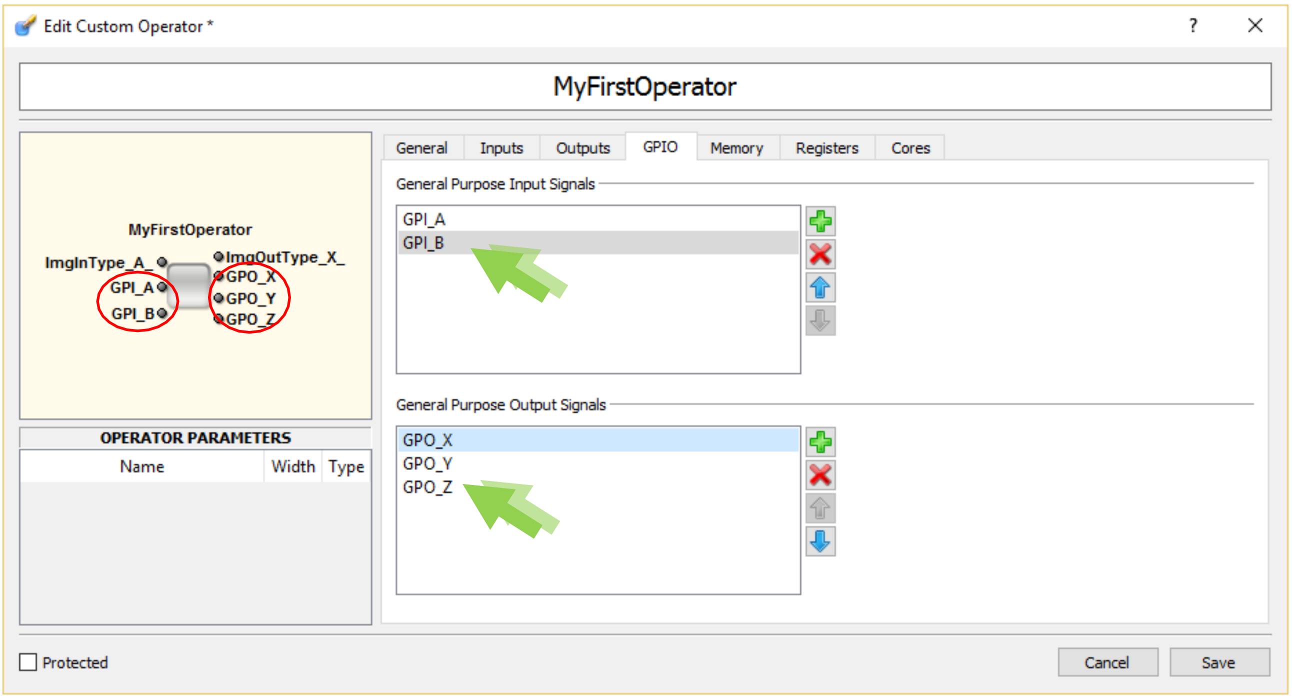
	Format	Pix.Width	Parall.	Dimension	Signess	Max.Width	Max.Height	Alias
1	Gray	8	4	Area	unsigned			

3.5 Defining the GPIO Ports

The General Purpose I/O interface allows connecting dedicated signal pins of the custom operator. Every GPIO port maps to a pin of the custom operator which is either an input or an output. Bidirectional pins are not supported. In VisualApplets, the corresponding operator ports are of type SIGNAL.

1. Go to tab GPIO.
2. Add as many GPIs and GPOs as you want, using the plus button ➕.
3. Double-click into the field to give a name to a specific GPI or GPO.

The defined GPIs and GPOs are immediately displayed in the depiction of the custom operator in the upper left hand panel of the program window:


Edit Custom Operator *

MyFirstOperator

General | Inputs | Outputs | GPIO | Memory | Registers | Cores

General Purpose Input Signals
- GPI_A
- GPI_B

General Purpose Output Signals
- GPO_X
- GPO_Y
- GPO_Z

OPERATOR PARAMETERS

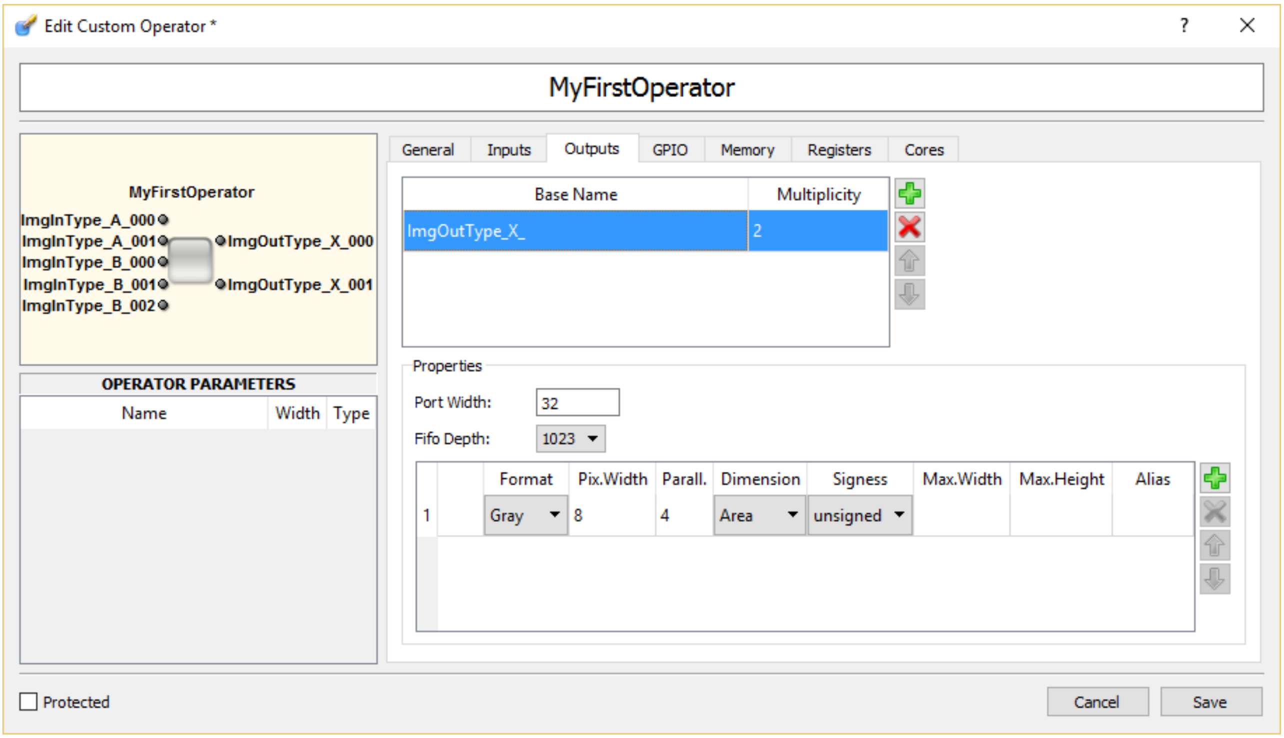
Name	Width	Type

Bidirectional Pins not Supported

The pins are either an input or an output. Bidirectional pins are not supported.

3.6 Defining the Image Output Ports

Under tab *Outputs*, you describe the properties of the image output ports.



Edit Custom Operator *

MyFirstOperator

Base Name	Multiplicity
ImgOutType_X_	2

Properties

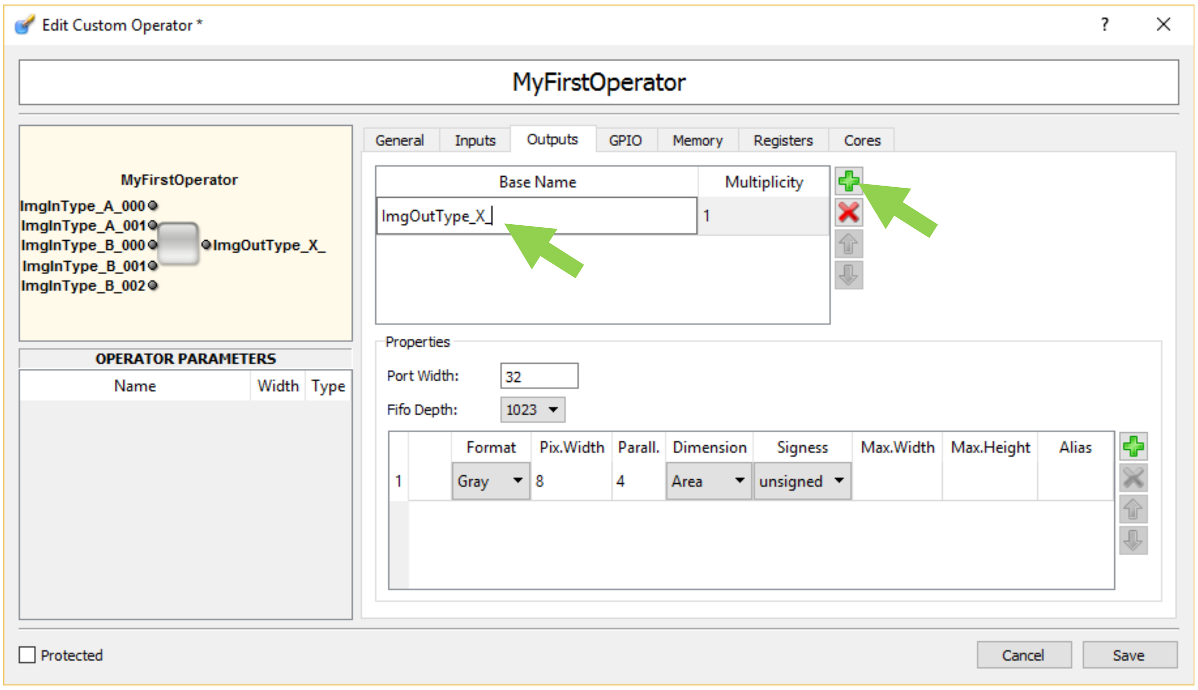
Port Width: 32

Fifo Depth: 1023

	Format	Pix.Width	Parall.	Dimension	Signess	Max.Width	Max.Height	Alias
1	Gray	8	4	Area	unsigned			

You can define one or more image output ports (ImgOut). Each ImgOut port may be used as often as you specify.

1. Click on the plus button ➕ to create a first image out (ImgOut) port.
2. Give a name to the ImgOut port.



Base Name	Multiplicity
ImgOutType_X_	1

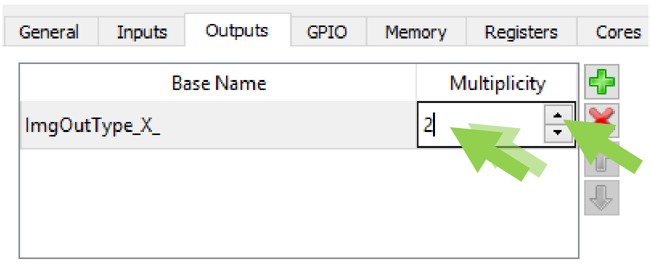
Properties

Port Width: 32

Fifo Depth: 1023

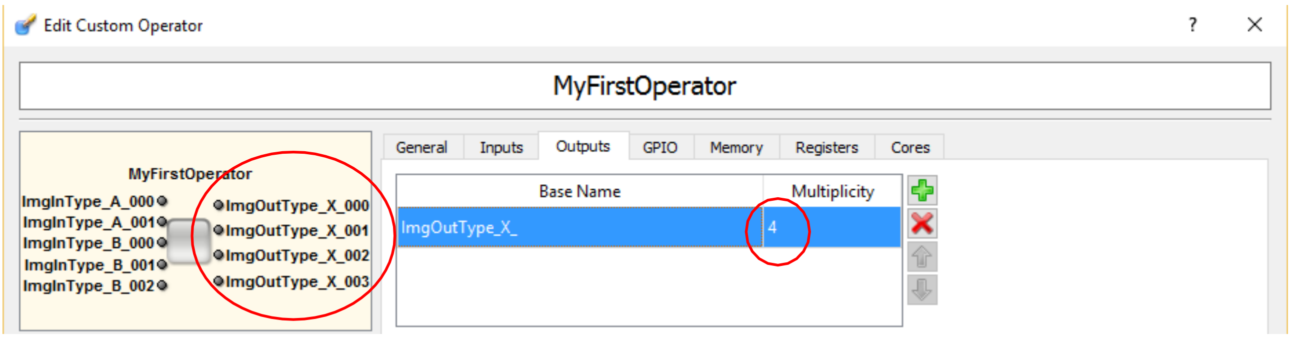
	Format	Pix.Width	Parall.	Dimension	Signess	Max.Width	Max.Height	Alias
1	Gray	8	4	Area	unsigned			

3. Double-click in the field of the *Multiplicity* column to create an array of ports. Multiplicity >1 defines an array of ports with a name consisting of the base name and an index.



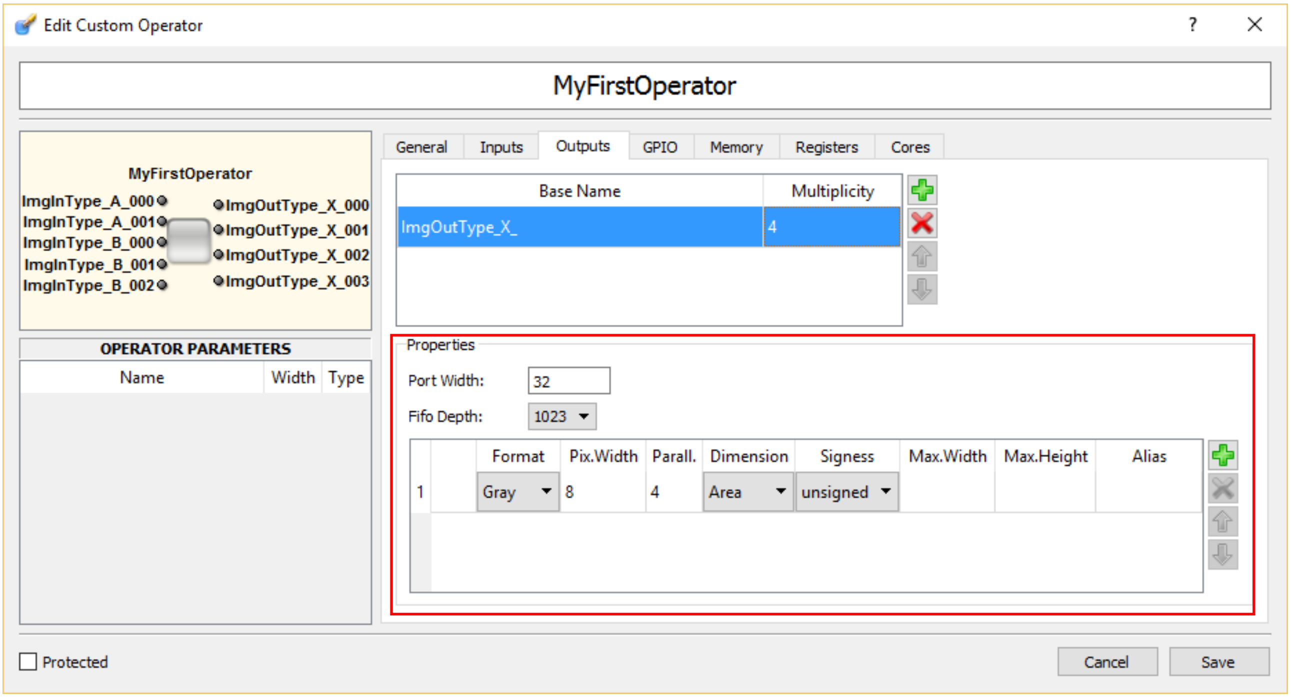
Base Name	Multiplicity
ImgOutType_X_	2

Immediately, the operator depiction in the program window displays the entered array of ImgOut ports:



Base Name	Multiplicity
ImgOutType_X_	4

In the ***Properties*** panel, you specify the properties of the protocols that are supported by this ImgOut port.



MyFirstOperator

ImgInType_A_000, ImgInType_A_001, ImgInType_B_000, ImgInType_B_001, ImgInType_B_002

ImgOutType_X_000, ImgOutType_X_001, ImgOutType_X_002, ImgOutType_X_003

OPERATOR PARAMETERS

Name	Width	Type

Base Name	Multiplicity
ImgOutType_X_	4

Properties

Port Width: 32

Fifo Depth: 1023

	Format	Pix.Width	Parall.	Dimension	Signess	Max.Width	Max.Height	Alias
1	Gray	8	4	Area	unsigned			

4. Under ***Port Width***, specify the width of the ImgOut port.

5. Under ***Fifo Depth***, specify the depth of the buffer FIFO for output data which at least needs to be provided by the VA core. The value must be a power of two minus 1 between 15 and 1023.

For an image interface port, you define a list of allowed protocols. A protocol can be described by the following properties:

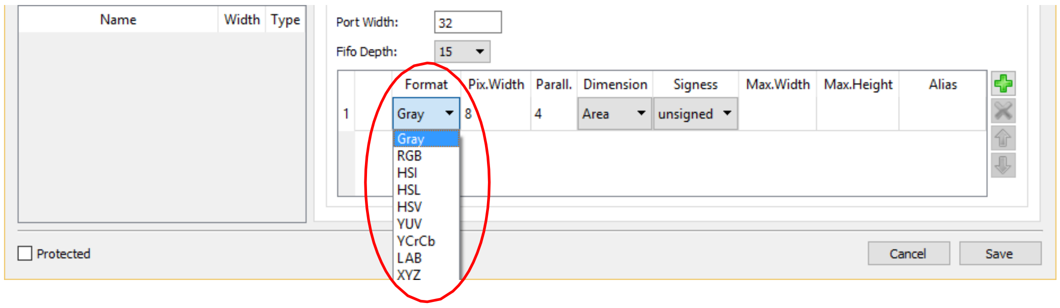
- Gray or color format (single or three data components with aggregated width N)
- Flavor of color format (RGB, HSI, HSL, HSV, YUV, YCrCb, LAB, XYZ)
- Pixel data bit width N = [1..64]
- Parallelism P = {1,2,4,8,16,32,64}
- 2D (Aray), 1D (Line), or 0D (Raw)
- Data type uint or int
- Max. image dimensions

Implicitly it is assumed that the kernel size is 1x1.

The listed protocols are numbered starting from zero and therewith define an ID (in the image below visible in the left hand column of the table in the *Properties* panel).

If you specify more than one protocol, you design the custom operator to support configuration of its input channel(s). In this case, several different protocols can be driven through a single port. The user of your custom operator can select the protocol he wants to use on a specific ImgOut port. According to the selection made by the VA user, the corresponding ID will be output to the related custom operator port. This enables the custom operator to adapt its behavior to the selected protocol.

6. Under ***Format***, specify the color format of the protocol.



Port Width: 32

Fifo Depth: 15

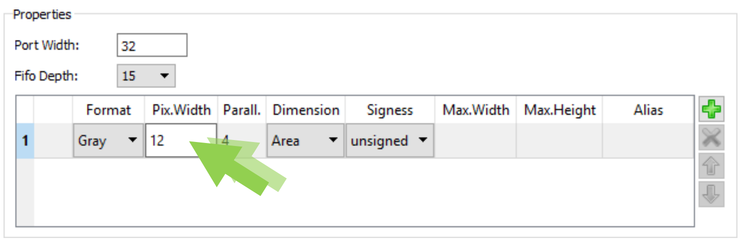
	Format	Pix.Width	Parall.	Dimension	Signess	Max.Width	Max.Height	Alias
1	Gray	8	4	Area	unsigned			

The following color formats are allowed:

- **gray*XxP***: gray image with ***X*** bits per pixel and parallelism ***P***
- **rgb*YxP*:** color image with ***Y***/3 bits per color component (red, green, blue) and parallelism ***P***
- **hsi*YxP*:** color image with ***Y***/3 bits per color component (HSI color model) and parallelism ***P***
- **hsl*YxP*:** color image with ***Y***/3 bits per color component (HSL color model) and parallelism ***P***
- **hsv*YxP*:** color image with ***Y***/3 bits per color component (HSV color model) and parallelism ***P***
- **yuv*YxP*:** color image with ***Y***/3 bits per color component (YUV color model) and parallelism ***P***
- **ycrcb*YxP*:** color image with ***Y***/3 bits per color component (YCrCb color model) and parallelism ***P***
- **lab*YxP*:** color image with ***Y***/3 bits per color component (LAB color model) and parallelism ***P***
- **xyz*YxP*:** color image with ***Y***/3 bits per color component (XYZ color model) and parallelism ***P***

7. Double-click in the field of column ***Pix.Width*** and specify the pixel data width for the

specific format:


Properties

Port Width: 32

Fifo Depth: 15

	Format	Pix.Width	Parall.	Dimension	Signess	Max.Width	Max.Height	Alias
1	Gray	12	4	Area	unsigned			

The value range of ***Pix.Width*** depends on your choice under ***Format***:

> **Gray**: The pixel data width (in the following referred to as ***X***) is limited to 64 bit**.**
>
> **All color formats**: The pixel data width (in the following referred to as ***Y***) must be a multiple of 3 and is limited to 63 bit.

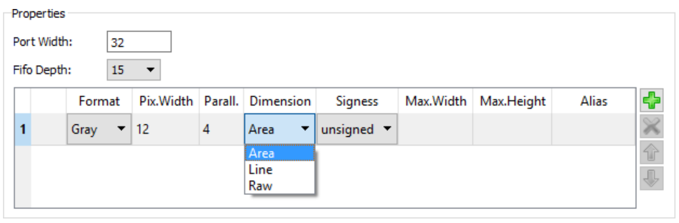
8. Double-click in the field of column ***Parall.*** and specify the parallelism for the specific format.

The parallelism defines the number of pixels which are contained in a single data word at the interface port. It must be chosen from following set of allowed values: ***P*** = {1, 2, 4, 8, 16, 32, 64}. Packing of image data into words of a given interface width ***N*** (specified under ***Port Width***) must follow certain rules:

- The data of all ***P*** pixels must fit in a single word of length ***N***. The data is stored LSB aligned which means that for a pixel width ***Z*** (***Z***=***X*** for grey, ***Z***=***Y*** for color) data is distributed as follows: Pixel[0]->Bits[0..***Z***-1] .. Pixel[***P***-1]->Bits[(***P***-1)****Z***..***P*******Z***-1].
- For RGB images the three color components are packed LSB aligned into a sub word[0..Y-1] in the following order: red uses the bits [0..***Y***/3-1], green the bits [***Y***/3..2****Y***/3-1] and blue the bits [2****Y***/3..3****Y***/3-1].
- For HSI color images the same rules than for RGB applies where H takes the role ofred, S that of green and I the role of blue.
- For HSL color images the same rules than for RGB applies where H takes the role ofred, S that of green and L the role of blue.
- For HSV color images the same rules than for RGB applies where H takes the role ofred, S that of green and V the role of blue.
- For YUV color images the same rules than for RGB applies where Y takes the role ofred, U that of green and V the role of blue.
- For YCrCb color images the same rules than for RGB applies where Y takes the roleof red, Cr that of green and Cb the role of blue.

- For LAB color images the same rules than for RGB applies where L takes the role ofred, A that of green and B the role of blue.
- For XYZ color images the same rules than for RGB applies where X takes the role ofred, Y that of green and Z the role of blue.

9. Under ***Dimension***, specify if the protocol supports 2D (Area), 1D (Line), or 0D (Raw) images.



Properties

Port Width: 32

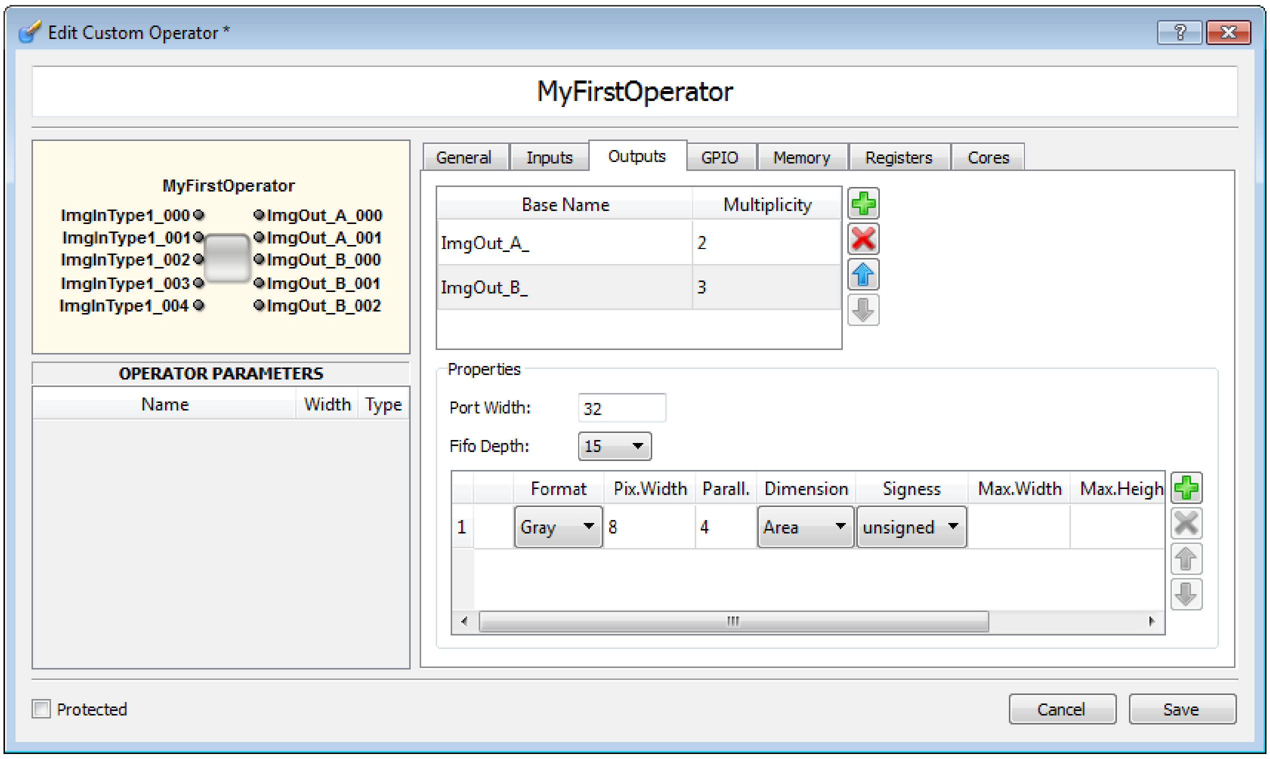
Fifo Depth: 15

	Format	Pix.Width	Parall.	Dimension	Signess	Max.Width	Max.Height	Alias
1	Gray	12	4	Area	unsigned			

10. ***Max.Width/Max.Height***: Using these optional fields you can define constraints for the image width and image height.

11. Repeat steps 7 to 11 to define as many protocols as you want the ImgOut port to support.

12. Repeat steps 2 to 12 to define as many ImgOut ports you want your custom operator to support.



Edit Custom Operator *

MyFirstOperator

OPERATOR PARAMETERS

Name	Width	Type

Base Name	Multiplicity
ImgOut_A_	2
ImgOut_B_	3

Properties

Port Width: 32

Fifo Depth: 15

	Format	Pix.Width	Parall.	Dimension	Signess	Max.Width	Max.Heigh
1	Gray	8	4	Area	unsigned		

☐ Protected

3.7 Defining the Memory Ports

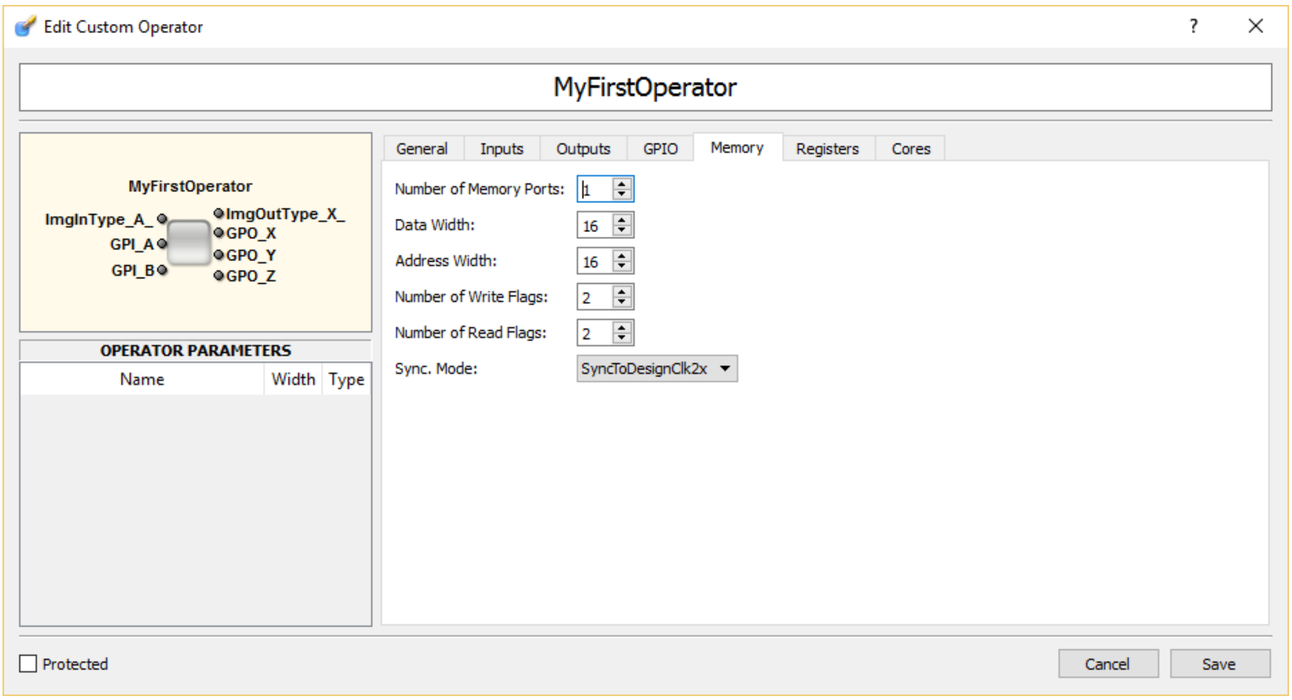
A custom operator may be set up for accessing one or more banks of memory (DRAM, SRAM, …).

All memory ports have a FIFO-like interface for write and read commands. The FIFOs reside in the VA part of the custom operator, so that you only need to implement a flow control, but not the FIFO. The timing of forwarding the FIFO content to the memory controller attached to the custom operator is fully controlled by VisualApplets.

Under the ***Memory*** tab, you can define that your operator gets access to external memory. You can specify up to 4 ports. You can specify the memory interface properties the operator needs.

Comply with Memory Layout of Target Platforms

Keep in mind the memory layout of potential target platforms (on which the applets containing the custom operator will run).



Number of Memory Ports: 1
Data Width: 16
Address Width: 16
Number of Write Flags: 2
Number of Read Flags: 2
Sync. Mode: SyncToDesignClk2x

Parameter name	Type	Description
Data Width	Integer	Data width
Address Width	Integer	Address width

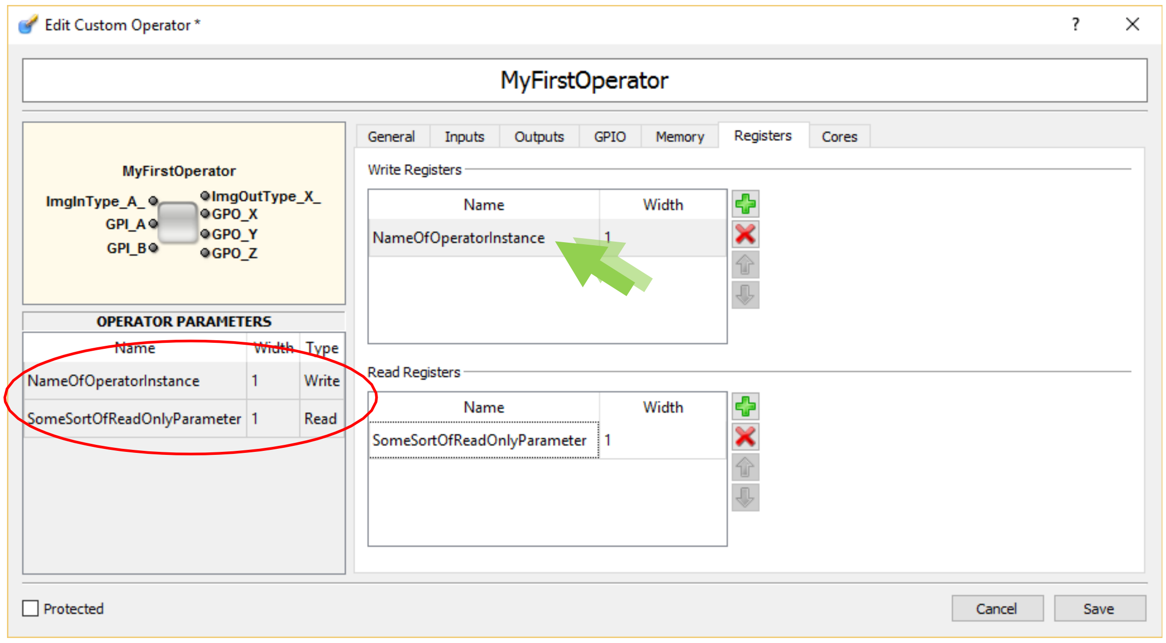
Number of Write Flags (Width)	Integer	Width of flag for marking write accesses. This parameter must be >= 1.
Number of Read Flags (Width)	Integer	Width of flag for marking read accesses. This parameter must be >= 8.
SyncMode	String	This parameter signals the relation of the memory interface clock and the design clock. Following values are possible: "SyncToDesignClk" – memory interface ports are synchronous to iDesignClk. "SyncToDesignClk2x" – memory interface ports are synchronous to iDesignClk2x.

3.8 Defining the Registers of the Custom Operator

Under the *Registers* tab, you can define the write and read registers your custom operator will provide. Each of this registers is accessed in VisualApplets via a dedicated operator parameter. (The parameter name is the same as the register name.)

1. Go to the ***Registers*** tab.
2. Under ***Write Registers***, define the write registers you want your custom operator to have.
3. Define a specific width for each write register.
4. Under ***Read Registers***, define the read register you want your custom operator to have.
5. Define a specific width for each read register.

The related operator parameters are immediately displayed in the left hand lower panel of the dialog window:


Edit Custom Operator *

MyFirstOperator

OPERATOR PARAMETERS

Name	Width	Type
NameOfOperatorInstance	1	Write
SomeSortOfReadOnlyParameter	1	Read

Write Registers

Name	Width
NameOfOperatorInstance	1

Read Registers

Name	Width
SomeSortOfReadOnlyParameter	1

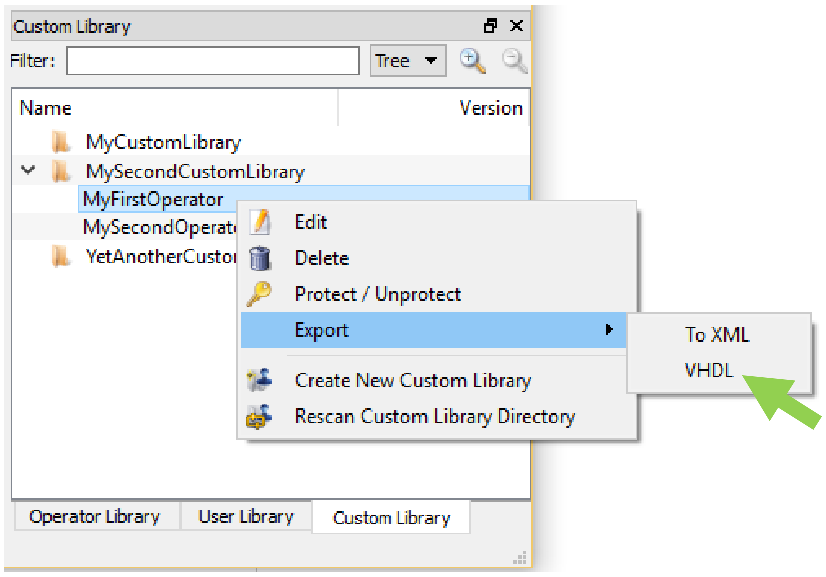
6. Click **Save.**

4 Generation of VHDL Black Box and Test Bench

After you have entered all details as described in section 3, you are ready for the actual VHDL coding. First of all, you need VisualApplets to generate the VHDL black box and test bench.

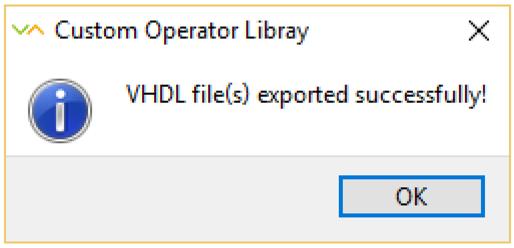
To trigger VHDL black box and test bench generation:

1. In the *Library* panel of the VisualApplets program window, go to the *Custom Library* tab.
2. Open the custom library and select the custom operator you want to implement.
3. Right-click on the operator name, and from the sub menu, select **Export -> VHDL**.



4. Specify the folder where you want the created VHDL files to be stored.

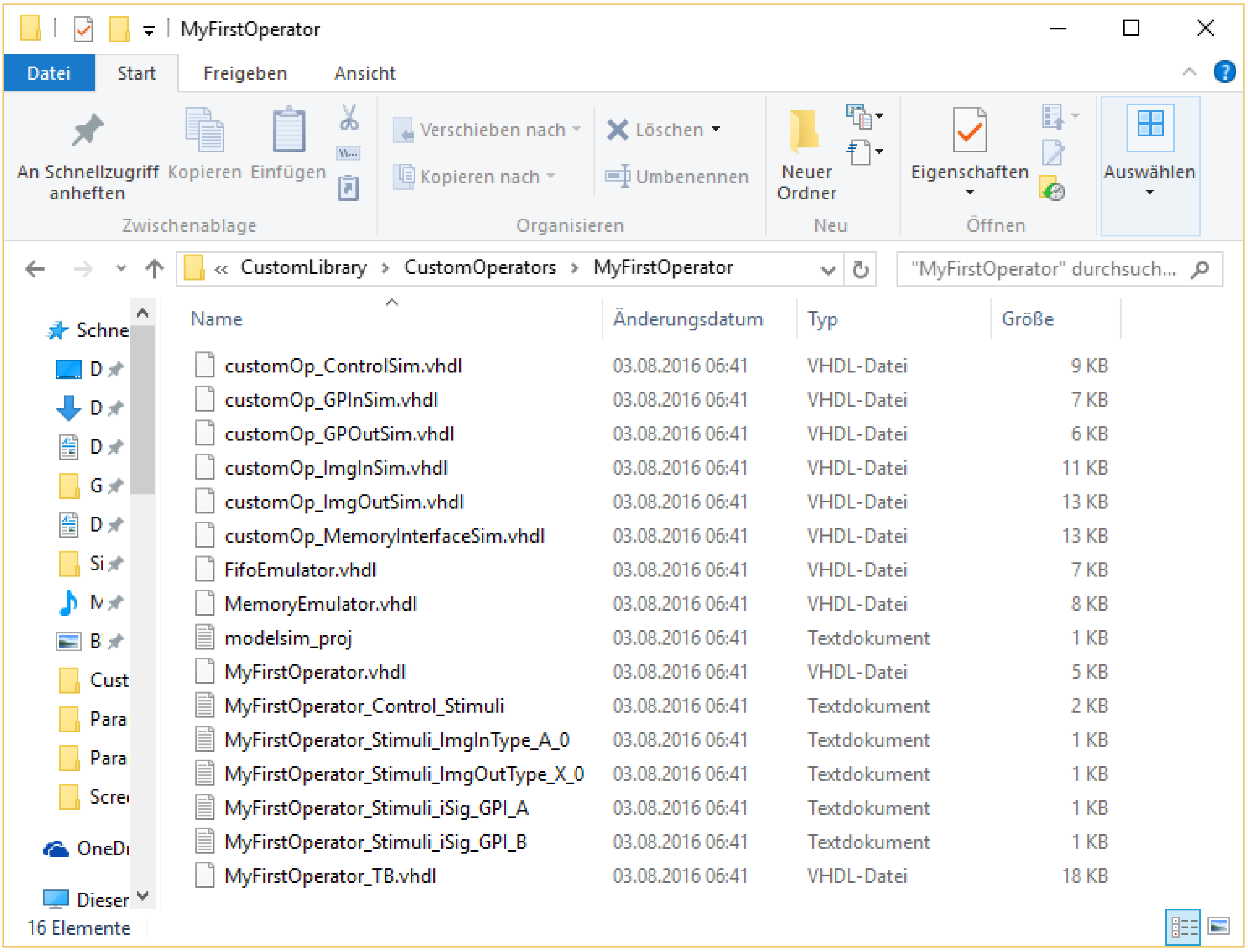
Now, the generation starts. After successful generation, you get the following message:


Custom Operator Libray

VHDL file(s) exported successfully!

OK

You find all generated files in the folder you specified:



Name	Änderungsdatum	Typ	Größe
customOp_ControlSim.vhdl	03.08.2016 06:41	VHDL-Datei	9 KB
customOp_GPInSim.vhdl	03.08.2016 06:41	VHDL-Datei	7 KB
customOp_GPOutSim.vhdl	03.08.2016 06:41	VHDL-Datei	6 KB
customOp_ImgInSim.vhdl	03.08.2016 06:41	VHDL-Datei	11 KB
customOp_ImgOutSim.vhdl	03.08.2016 06:41	VHDL-Datei	13 KB
customOp_MemoryInterfaceSim.vhdl	03.08.2016 06:41	VHDL-Datei	13 KB
FifoEmulator.vhdl	03.08.2016 06:41	VHDL-Datei	7 KB
MemoryEmulator.vhdl	03.08.2016 06:41	VHDL-Datei	8 KB
modelsim_proj	03.08.2016 06:41	Textdokument	1 KB
MyFirstOperator.vhdl	03.08.2016 06:41	VHDL-Datei	5 KB
MyFirstOperator_Control_Stimuli	03.08.2016 06:41	Textdokument	2 KB
MyFirstOperator_Stimuli_ImgInType_A_0	03.08.2016 06:41	Textdokument	1 KB
MyFirstOperator_Stimuli_ImgOutType_X_0	03.08.2016 06:41	Textdokument	1 KB
MyFirstOperator_Stimuli_iSig_GPI_A	03.08.2016 06:41	Textdokument	1 KB
MyFirstOperator_Stimuli_iSig_GPI_B	03.08.2016 06:41	Textdokument	1 KB
MyFirstOperator_TB.vhdl	03.08.2016 06:41	VHDL-Datei	18 KB

5 Operator Interface Ports

The generated black box provides all ports you specified via the GUI (see section3).

In this chapter, you find a detailed description of how these ports look like in the generated VHDL black box.

5.1 Clock System, Reset and Enable

VisualApplets supports two clock domains. There is a base design clock and one derived clock which is in phase with that clock and has double frequency. Accordingly, there are two clock inputs to the Custom Operator. Additionally, there is a Reset and Enable input as described above.

Port	Direction	Width	Description
iDesignClk	In	1	Base design clock
iDesignClk2x	In	1	Clock sync. to iDesignClk but double frequency
iReset	In	1	Reset of operator
iEnable	In	1	Enable processing

5.2 Parameter Interface

The definition of write register ports as described in section 3.8 (XML[1]: **`Operator/IO/RegInInfo)`** leads to an interface as follows where **`PORTID`** is the register name and **`PORTIDWidth`** is the defined register width (in XMl defined in the corresponding entry of *Operator/RegIn)*.

[1] All entries you make to specify the interface of your custom operator (section 3) are written into an XML file.Information about structure and syntax of this XML file is provided in the Appendix, section 12.1 .

Port	Direction	Width	Description
ivReg_***PORTID***_D	In	***PORTID****Width*	Register data
iReg_***PORTID***_Wr	In	1	Signal write access

The definition of read register ports as described in section 3.8 (XML: **`Operator/IO/RegOutInfo`**) leads to the following interface, accordingly:

Port	Direction	Width	Description
ovReg_***PORTID***_D	Out	***PORTID****Width*	Register data
iReg_***PORTID***_Rd	In	1	Signal read access

5.3 Image Communication Interfaces

For communication of data between the VisualApplets core and a Custom Operator, image communication ports as described in section 3.4 may be configured. Communication is done via a simple FIFO interface and an additional format identifier port.

5.3.1 Interfaces of Type ImgIn

An **`ImgIn`** channel for transferring data from the VisualApplets core to a Custom Operator leads to an interface as follows where **`PORTID`** is the name of the corresponding port type name (XML: attribute *Operator/ImgIn/name* referenced by *Operator/IO/ImgInInfo*) and **`X`** is a port number for differentiating several ports of the same kind:

Port	Direction	Width	Description
iv***PORTIDX***Data	In	***PORTID****Width*	Data entering the Custom Operator
o***PORTIDX***Read	Out	1	Accept input data

Port	Direction	Width	Description
i*PORTIDX*EndOfLine	In	1	Signal end of line. If this flag is activated data doesn't contain pixel values.
iPORTIDXEndOfFrame	In	1	Signal end of frame. If this flag is activated data doesn't contain pixel values. This flag is only asserted when end of line is signaled as well.
i***PORTIDX***FIFOEmpty	In	1	Buffer FIFO is empty
iv***PORTIDX***FIFOCnt	In	Ceil Log2(***PORTID***FIFODepth)	Number of words in buffer FIFO. This signal can be used to generate FIFO flags like Almost Empty.
iv***PORTIDX***_FID_D	In	Ceil Log2(N)	Predefined parameter which notifies about the current image data format. N is the number of image formats specified for this port.

Figure 3 illustrates the data flow at an **`ImgIn`** port. The port name component **`PORTIDX`** has been substituted by 'ImgIn'. The waveform shows the input of a two dimensional frame of size 3x2. When the **`ImgIn`** port is part of several O-synchronous input ports, all of them must consume the FIFO data simultaneously. In that case the FIFO fill level of all ports will exactly match so the operator only needs to implement flow control according to the fill level of one out of several O-synchronous inputs.

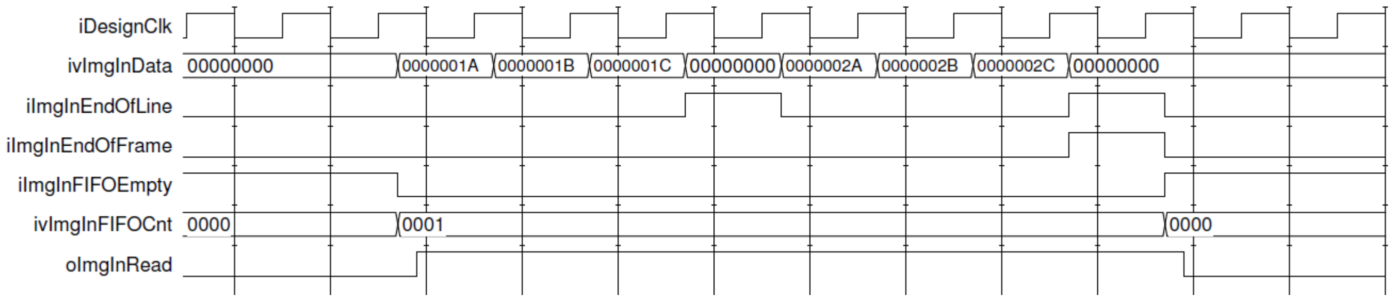


Figure 3: Waveform illustrating the protocol on an image input port.

5.3.2 Interfaces of Type ImgOut

An **`ImgOut`** channel for transferring data from a Custom Operator to the VisualApplets core leads to an interface as follows where **`PORTID`** is the name of the corresponding port type name (XML: attribute *Operator/ImgOut/name* referenced by *Operator/IO/ImgOutInfo*) and **`X`** is a port number for differentiating several ports of the same kind:

Port	Direction	Width	Description
ov***PORTIDX***Data	Out	***PORTID****Width*	Output data
o***PORTIDX***Valid	Out	1	Output data valid
o***PORTIDX***EndOfLine	Out	1	Signal current write access as end of line notification. Write data is then not interpreted as pixel data.
o***PORTIDX***EndOfFrame	Out	1	Signal current write access as end of frame notification. Write data is then not interpreted as pixel

Port	Direction	Width	Description
			data. This flag needs to be correlated with an end of line strobe at the same time.
i***PORTIDX***FIFOFull	In	1	Buffer FIFO is full, no further data is accepted
iv***PORTIDX***FIFOCnt	In	Ceil Log2(***PORTID***FIFODepth)	Number of words in buffer FIFO. This signal can be used to generate FIFO flags like Almost Full.
iv***PORTIDX***_FID_D	In	Ceil Log2(N)	Predefined parameter which notifies about the current image data format. N is the number of image formats specified for this port.

Figure 4 illustrates the data flow at an `ImgOut` port. The waveform shows the output of a two dimensional frame of size 3x2. When the `ImgOut` port is part of several O-synchronous output ports all of them must emit data simultaneously.

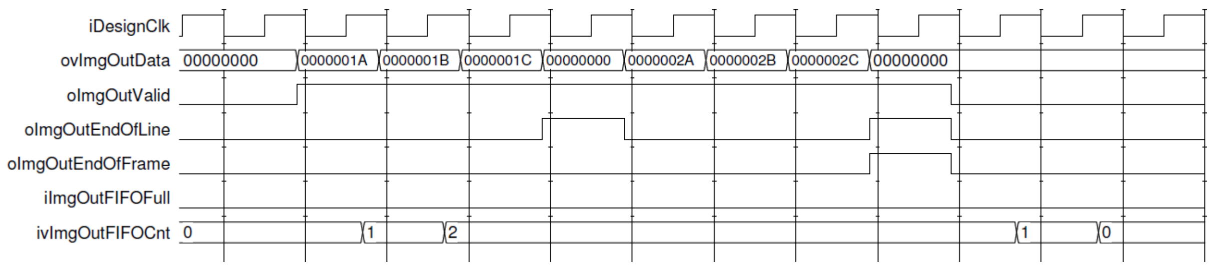


Figure 4: Waveform illustrating the protocol on an image output port.

5.4 Memory Interfaces

A Custom Operator may be set up for having up to four memory ports. The I/O ports of the generated interface get a suffix ***X*** where ***X*** is the index of the memory port (XML: *Operator/IO/MemInfo)*.

Name	Direction	Width	Description
ovMemWrData***X***	Out	*MemDataWidth**X***	Write data output to memory via VisualApplets core
ovMemWrFlag***X***	Out	*MemWrFlagWidth**X***	Write flag output
ovMemWrAddr***X***	Out	*MemAddrWidth**X***	Write address
oMemWrAddrValid***X***	Out	1	Emit write command
oMemWrPriority***X***	Out	1	Request priority for this write port
iMemWrAlmostFull***X***	In	1	Only single further write command may be accepted
iMemWrFull***X***	In	1	No write command is accepted as concerning FIFO is full
iMemWrEmpty***X***	In	1	FIFO for write commands is empty
ivMemWrCnt***X***	In	4	Number of buffered write commands
ivMemWrFlag***X***	In	*MemWrFlagWidth**X***	Write flag output from the VisualApplets core

Name	Direction	Width	Description
iMemWrFlagValid***X***	In	1	Write flag input valid – signals that iMemWrFlag***X*** is valid, which means that write access which had been marked with corresponding oMemWrFlag***X*** has been executed.
ovMemRdFlag***X***	Out	*MemRdFlagWidth**X***	Read flag
ovMemRdAddr***X***	Out	*MemAddrWidth**X***	Read address
oMemRdAddrValid***X***	Out	1	Emit read command
oMemRdPriority***X***	Out	1	Request priority for this read port
iMemRdAlmostFull***X***	In	1	Only single further read command may be accepted
iMemRdFull***X***	In	1	No read command is accepted as concerning FIFO is full
iMemRdEmpty***X***	In	1	FIFO for read commands is empty
ivMemRdCnt***X***	In	4	Number of buffered read commands
ivMemRdFlag***X***	In	*MemRdFlagWidth**X***	Read flag input – only valid when iMemRdDataValid***X*** is asserted
ivMemRdData***X***	In	*MemDataWidth**X***	Read data input

Name	Direction	Width	Description
iMemRdDataValid***X***	In	1	Read data valid

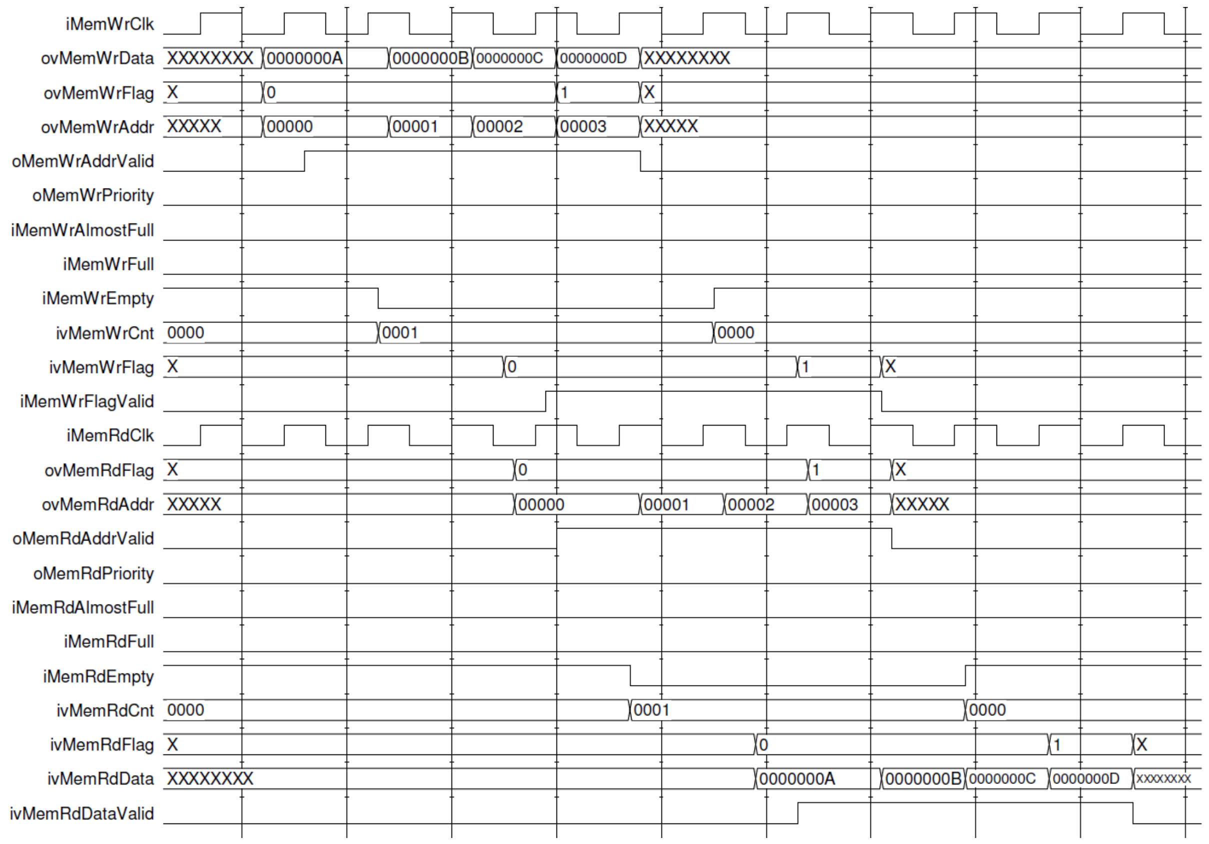


Figure 5: Waveform illustrating the memory interface protocol.

5.5 General Purpose I/O pins

Any GPIO input or output signal which has been defined in the interface description of the custom operator (section 3) has a corresponding input or output port in the resulting operator interface. The following ports will be created when the general purpose pins are declared (in XML with the name ***NAME*** within *Operator/IO/RegInInfo* or *Operator/IO/RegOutInfo*):

- iSig_***NAME*** for a GPIO input signal called *NAME*
- oSig_***NAME*** for a GPIO output signal called *NAME*

6 VHDL Simulation and Verification

For emulating a VisualApplets design which contains a custom operator module, VisualApplets creates a simulation test bench for the interfaces connecting to the custom operator.

Each interface port is emulated independently, driven by File I/O. The simulation entity shall consist of following elements:

- Emulation of register access. According to a stimuli file a set of registers can be written and read.
- Emulator for frame source connected to ports of type ImgIn. Stimulated by file these kinds of modules output frame data to ImgIn.
- Emulator for frame sink connected to ports of type ImgOut. This kind of module emulates an operator which is connected to ImgOut. The module writes the received data to file.
- Memory port emulator.
- GPIO emulator. Each GPIO signal for input is driven by a signal generator which is configured by a file. Each GPIO signal output is monitored and changes of the signal are written to a report file.
-

6.1 Simulation Framework

For RTL level simulation, VisualApplets creates a VHDL file containing a package with the name Cus t omOper at or _***<OPERATORNAME>*** where ***<OPERATORNAME>*** is the given operator name.

This package contains the components ***<OPERATORNAME>*** and ***<OPERATORNAME>***_TB where the latter is a test bench of the interface between the VisualApplets design and the Custom Operator. The following shows the resulting code for a simple Custom Operator called *RegExample* consisting only of a read and write register port (‘Ctrl’ and ‘Status’), each 4 bit wide:

```
component
RegExampleport(
  iDesignClk: in std_logic := '0';
  iDesignClk2x: in std_logic :=
  '0';iReset: in std_logic := '0';
  iEnable: in std_logic := '0';
  ivReg_Ctrl_D: in std_logic_vector(3 downto
  0);iReg_Ctrl_Wr: in std_logic;
  ovReg_Status_D: out std_logic_vector(3 downto 0);
  iReg_Status_Rd: in std_logic
);
end component;

component
RegExample_TBgeneric(
  DesignClkPeriod: time := 16 ns;
  Register_StimuliFileName: string := ""
);
end component;
```

The test bench creates an instance of the custom operator and connects protocol emulation modules to each interface ports. The following sections describe the different kinds of emulators, how they may be controlled via stimuli files, and how output files are generated.

6.2 Emulation of Register Interface

The generated test bench implements an emulator for a register access interface. The emulator is configured for addressing a design with a single process. Addresses of write and read registers start from 0x4 where addresses for registers are counted up with an increment of 1 according to the sequence of the register interface ports in the given custom operator component (like the above example component *RegExample*). Register addresses for reading and writing are counted independently. The emulator is driven by a text file which is set by the entity parameter *Register_StimuliFileName* as provided in the above VHDL code.

The following commands may be present in the stimuli file:

REM	Rest of line is comment
GRS	Emulate global reset
PRS	Emulate process reset. This command has the following syntax, **`PRS <procNr>`** where the parameter **`<procNr>`** must always be 0.

PEN	Enable process. The syntax is as follows, **`PEN <procNr> <value>`** with **`<procNr>`** being always 0 and **`<value>`** signaling the enable state.
WCK	Wait for a number of clock cycles. The syntax is as follows, **`WCK <clock_ticks>`** with **`<clock_ticks>`** giving the number of clock ticks in hexadecimal format
WRR	Write to register, **`WRR <wrRegAddr> <value>`** With the parameters: **`<wrRegAddr>:`** address of register (hex) **`<value>`**: hexadecimal register value
RDR	Read from register, **`RDR <rdRegAddr>`** with **`<wrRegAddr>`** being the register address (hex).

After the last parameter of any command, a comment may be added preceded by '#'.

The following code is an example stimuli file which accesses the registers according to the above given test bench RegExample_TB:

```
REM
************************************************************
REM Command formats:
REM  GRS                    -> Global reset
REM  GEN <value>            -> Set global enable to <value>
REM  PRS <procID>           -> Reset process <procID> (0 ..
F) REM  PEN <procID> <value>   -> Set enable of process
<procID> to
<value>
REM  WCK <clk_ticks>        -> Wait for <clk_ticks> clock
```

```
cycles REM WRR <wrRegAddr> <value> -> Write <value> to register
<wrRegAddr>REM  RDR <rdRegAddr>     -> Read from register
<rdRegAddr>
REM ************************************************************

WCK 0004          # wait for 4 clock
                    cycles
GRS               # global reset
GEN 1             # set global enable
WCK 0001          # wait for 1 clock tick
PRS 0             # reset process 0
PEN 0 1           # set enable of process
                    0
WCK 0002          # wait for 2 clock ticks
WRR 0004 0000000A # write 0xA to address
                    0x4
WCK 0002          # wait for 2 clock ticks
RDR 0004          # read from address 0x4
WCK FFFF
```

6.3 Emulation of ImgIn Interface

The emulation of image communication interfaces of type ImgIn is driven by a stimuli file providing information about the sequence of data which enters the Custom Operator. For any present ImgIn port the test bench has a generic ***<PORTIDX>**_StimuliFileName* where ***<PORTIDX>*** is the name of the corresponding image input port type followed by the port number. Each line within the given file must follow the syntax,

```
<Command> <Data> <EndOfLine> <EndOfFrame> <DataValid>
```

where **`<Command>`** is a three letter command, **`<Data>`** provides an hexadecimal data word, and the three remaining parameters correspond to the image protocol flags.

The following table describes the available commands:

DAT	Data command. This command provides data which will become input at the port **`ivPORTIDXData`** and the associated image protocol flag ports.
WCK	Wait command. The parameter **`<Data>`** provides the number of clock ticks for which the command interpreter pauses.
FI D	Set FID input. The parameter **`<Data>`** provides the value to which the port **`ivPORTIDX_FID_D`** will be set.

To any command line a comment may be added, preceded by '#'.

The following code is an example stimuli file which causes the input of an 3x2-image:

```
FID 00000001 0 0 0 #Format: Cmd Data(hex) EndOfLine EndOfFrame DataValid
DAT 00000000 0 0 0
DAT 0000001a 0 0 1
DAT 0000001b 0 0 1
DAT 0000001c 0 0 1
DAT 00000000 1 0 1
WCK 00000004 0 0 0
DAT 0000002a 0 0 1
DAT 0000002b 0 0 1

DAT 0000002c 0 0 1
DAT 00000000 1 1 1
WCK 0000FFFF 0 0 0
```

6.4 Emulation of ImgOut Interface

The emulation of image communication interfaces of type ImgOut is driven by a stimuli file where information is provided about the sequence of FID states. For any present ImgOut port the VA_Design_Emulator entity has a generic ***<PORTIDX>**_StimuliFileName* where ***<PORTIDX>*** is the name of the corresponding image output port type followed by the port number. The syntax is exactly the same as in the case of the stimuli for ImgIn interfaces except that no DAT command is available. A simple stimuli file may look like,

```
WCK 00000010 0 0 0     #Format: Command Data(hex)
FID 00000001 0 0 0
WCK 0000FFFF 0 0 0
```

where the parameters **`<EndOfLine>`**,**`<EndOfFrame>`** and **`<DataValid>`** are actually meaningless.

The ImgOut interface emulator present in the generated test bench writes the received data to file. For that purpose the test bench entity has a generic ***<PORTIDX>***_*DumpFileName*. During simulation a file with the given name is created and the data is written using *DAT* and *WCK* commands in a format, which exactly corresponds to the stimuli file format for an ImgIn interface emulator.

6.5 Emulation of Memory Communication

When the Custom Operator implements an interface to memory the test bench connects a memory emulation module to the corresponding interface ports. The Custom Operator may not rely on a certain timing of the memory interface (like time until read data is returned) as this is fully controlled by VisualApplets and may vary between platforms and even between different designs.

6.6 GPIO Emulation

The emulation of dedicated input signals is done for each signal independently, driven by a stimuli file. There information is provided about the sequence of signal states. The stimuli file may consist of a number of commands which are described below. For any present output signal port the test bench entity has a generic *iSig_**<NAME>**_StimuliFileName* where ***<NAME>*** is the concerning port name.

The following table describes the available commands:

SET	Set signal. This command provides the signal state to which the output at the port i Si g_***NAME*** will be set. The next command will be executed one clock tick later. It has the syntax, **`SET <value>`** where **`<value>`** may be 0 or 1.

WCK	Wait command. It has the syntax, `WCK <ticks>` where the parameter **`<ticks>`** provides the number of clock ticks for which the signal will be held constant.
RST	Restart from begin. The command interpreter will start again from the first line of the stimuli file. This command does not have any parameters. The command will execute the first command of the file at the same clock tick allowing assembling a loop without a gap.
STP	Stop at current state. The command interpreter will stop and the current signal state will be held constant until end of simulation. This command does not have any parameters.

To any command line a comment may be added, preceded by '#'.

The following code is an example stimuli file which causes the Custom Operator input signaltoggling being low for 5 clock cycles and high for 7 clock cycles (synchronous to iDesignClk):

```
SET 0               # deassert output
WCK 0004            # wait for 4 clock cycles
SET 1               # assert output
WCK 0006            # wait for 6 clock cycles
RST                 # restart from begin
```

Dedicated output signals are monitored writing a dump file *oSig_**<NAME>**_DumpFileName* where ***<NAME>*** is the concerning port name. The file is composed of SET and WCK commands exactly corresponding to the commands of the stimuli file for an dedicated input signal.

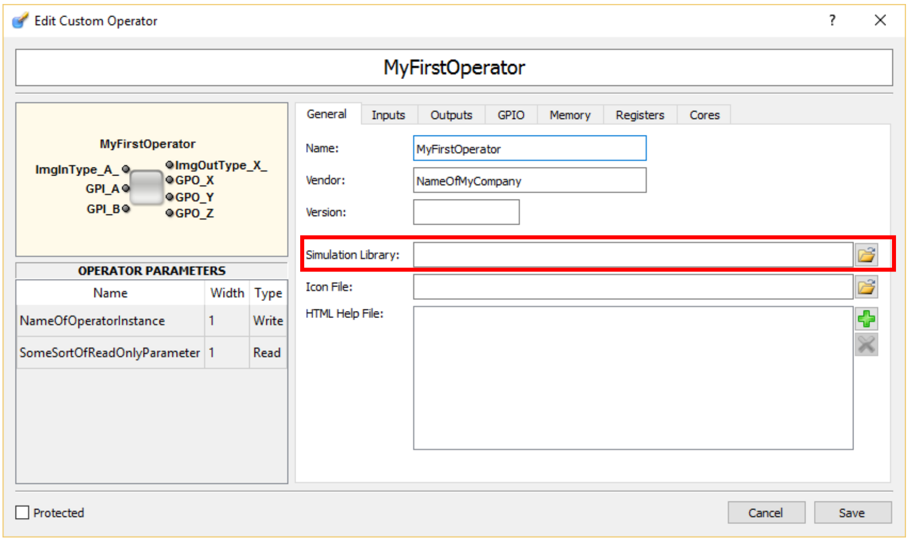
7 Defining the Custom Operator’s Software Interface

The following software components must be provided for fully integrating a custom operator to VisualApplets:

1. High-level simulation component
2. Throughput analysis

The software components need to be compiled to a dynamic link library with a predefined set of exported C-Functions.

You add this file to the operator specification under tab *General / Simulation Library*:



Name: MyFirstOperator

Vendor: NameOfMyCompany

Version:

Simulation Library:

Icon File:

HTML Help File:

OPERATOR PARAMETERS

Name	Width	Type
NameOfOperatorInstance	1	Write
SomeSortOfReadOnlyParameter	1	Read

☐ Protected

(In XML, the entry Operator/Info/LibraryFile points to this file.)

7.1 High-level Simulation

7.1.1 Overview

For High-level simulation within VisualApplets the following function must be exported,

```
int SimulateOPNAME (va_custom_op_sim_handle simHandle)
```

where ***OPNAME*** is the name of the Custom Operator.

High-level simulation must be done according to following requirements:

- **Frame based simulation** - On each image input port it can be queried whether one or more frames are available. If all ports which are required for starting simulation are able to provide a frame then the concerning output frames need to be computed and emitted via calls of appropriate functions. For one dimensional image data the data stream is automatically split into frames and simulated just like2D-data.
- **Bit accurate simulation** – The calculation of resulting frames must be bit accurate, i.e. the output data must be exactly equal to the data generated by the hardware implementation.
- **Keeping consistency of flow** – When operator input ports are synchronous to eachother input images must be fetched accordingly. When several outputs are definedimages must be output simultaneously. For the simulation function this means thatwhen a frame is output to one output link it must also output a frame to all other output links before the simulation function is returning.

As the behavior of the operator typically depends on the set of operator parameters these parameters may be queried via the following interface:

Nr.	**Function**	**Description**
1	`vaSi_CustomOp_GetParamValue()`	Get value of operator parameter.

A number of functions are provided by VisualApplets for getting, generating and storing image data for the Custom Operator:

Nr.	**Function**	**Description**
1	`vaSi_CustomOp_GetInputImage()`	Get image available at an ImgIn port.
2	`vaSi_CustomOp_PutOutputImage()`	Output image to ImgOut port.
3	`vaSi_CustomOp_InputHasImage()`	Query whether ImgIn port may deliver an image.

Nr.	Function	Description
4	`vaSi_CustomOp_OutputReady()`	Query whether ImgOut port may take an image.
5	`vaSi_CustomOp_CreateImage()`	Create new image.
6	`vaSi_CustomOp_DeleteImage()`	Delete image.
7	`vaSi_CustomOp_StoreImage()`	Store image in local storage of operator instance providing a name whereby the image may later be referenced.
8	`vaSi_CustomOp_GetStoredImagesCount()`	Query number of images stored within operator instance.
9	`vaSi_CustomOp_GetStoredImage()`	Get stored image by index.
10	`vaSi_CustomOp_GetNameOfStoredImage()`	Get name of stored image by index.
11	`vaSi_CustomOp_GetStoredImageByName()`	Get stored image by name.
12	`vaSi_CustomOp_CreateImageFormat()`	Create new image format handle which becomes initialized by the format associated with the given port.
13	`vaSi_CustomOp_CopyImageFormat()`	Create new image format which isa copy of given format.
14	`vaSi_CustomOp_DeleteImageFormat()`	Delete image format handle created earlier.

For manipulating images via image handles the following functions are available:

Nr.	Function	Description
1	`vaSi_Image_GetFormat()`	Get image format.
2	`vaSi_Image_SetProperty()`	Set property of frame (e.g. height).
3	`vaSi_Image_GetProperty()`	Get property of frame.
4	`vaSi_Image_SetPixelValue()`	Set pixel component value
5	`vaSi_Image_GetPixelValue()`	Get pixel component value
6	`vaSi_Image_SetLineLength ()`	Set individual length of a line.
7	`vaSi_Image_GetLineLength ()`	Get length of individual line.

Image formats may be manipulated via the following functions:

Nr.	Function	Description
1	`vaSi_ImageFormat_SetProperty()`	Set image format property (e.g. maximum width).
2	`vaSi_ImageFormat_GetProperty()`	Get image format property.

The simulation function may inject an status message (i.e., error message) into the VisualApplets simulation system using the following functions:

Nr.	Function	Description
1	`vaSi_CreateStatusMessage()`	Create status message.
2	`vaSi_SetStatusMessageProperty()`	Set property of status message (like severity).
3	`vaSi_SendStatusMessage()`	Submit the status message to the simulation engine.

7.1.2 Communicating Data

For querying information and configuring parameters data must be exchanged through the software interface. In order to keep the interface functions simple but providing a type save interface an abstraction mechanism for data is implemented. Whenever data of different types needs to be communicated a data structure called `va_data` is used, containing a reference to the data and information about the underlying data type. This data structure is created by the user but configured by dedicated functions listed below. The following table shows the data types which are handled by this method:

VA_ENUM	enum entry given as 32-Bit integer
VA_INT32	32-Bit signed integer
VA_UINT32	32-Bit unsigned integer
VA_INT64	64-Bit signed integer
VA_UINT64	64-Bit unsigned integer
VA_DOUBLE	Floating-point number, double precision
VA_INT32_ARRAY	Array of 32-Bit signed integer numbers
VA_UINT32_ARRAY	Array of 32-Bit unsigned integer numbers
VA_INT64_ARRAY	Array of 64-Bit signed integer numbers
VA_UINT64_ARRAY	Array of 64-Bit unsigned integer numbers
VA_DOUBLE_ARRAY	Array of double numbers
VA_STRING	String given as const char*

Configuring an earlier created **va_data** structure (**vaData**) for setting up data communication is done via the following functions:

```
va_data* va_data_enum(va_data* vaData, int32_t *data)
va_data* va_data_int32(va_data* vaData, int32_t *data)
va_data* va_data_uint32(va_data* vaData, uint32_t
*data) va_data* va_data_int64(va_data* vaData, int64_t
*data) va_data* va_data_uint64(va_data* vaData,
uint64_t *data) va_data* va_data_double(va_data*
vaData, double *data) va_data*
va_data_int32_array(va_data* vaData, int32_t *data,
                          size_t elementCount)
va_data* va_data_uint32_array (va_data* vaData, uint32_t *data,
                            size_t elementCount)
va_data* va_data_int64_array (va_data* vaData, int64_t *data,
                           size_t elementCount)
va_data* va_data_uint64_array (va_data* vaData, uint64_t *data,
                            size_t elementCount)
va_data* va_data_double_array (va_data* vaData, double *data,
                            size_t elementCount)
va_data* va_data_string(va_data* vaData, char data*, size_t
strSize)va_data* va_data_const_string(va_data* vaData, const
char **data)
```

For strings there are two options how strings are communicated:

1. Providing a char array via `va_data_string()`. Then queried string data will be copied to that array.
2. Providing a pointer to const char*. Then a pointer to an internal string representation is returned when information of type VA_STRING is queried. When you use this approach check the livetime of the returned string.

Example Code:

The following example shows code for querying the image width.

```
uint32_t imgWidth;
va_data
va_imgWidth;
va_data_double(&va_imgWidth,&imgWidth);

vaSi_Image_GetProperty(imageHandle, "Width", &va_imgWidth);
```

After that the variable **`imgWidth`** will contain the requested information.

7.1.3 Detailed Description of Interface Functions

The following gives a detailed description of parameters and returned values for the specified simulation interface functions.

Function	`int vaSi_CustomOp_GetParamValue (va_custom_op_sim_handle simHandle, const char* paramName, va_data *value)`
Parameter 1	Simulation handle provided to the operator simulation function.
Parameter 2	Name of parameter.
Parameter 3	Return parameter for queried value.
Description	Returns the value of the parameter with the given name.
Return value	0 : Value is queried data <0: Cannot query parameter

Function	`int vaSi_CustomOp_GetInputImage (va_custom_op_sim_handle simHandle, const char* portName, va_image_handle *image)`
Parameter 1	Simulation handle provided to the operator simulation function.
Parameter 2	Name of operator port.
Parameter 3	Return parameter for image handle.
Description	Take an image which enters the operator at the given port and return a handle referencing that image. Before returning from the simulation function this image must either be stored by calling vaSi_CustomOp_StoreImage() or deleted by calling vaSi_CustomOp_DeleteImage().
Return value	0: OK <0 : Cannot get image

Function	`int vaSi_CustomOp_PutOutputImage (va_custom_op_sim_handle simHandle, const char* portName, va_image_handle imageHandle)`
Parameter 1	Simulation handle provided to the operator simulation function.
Parameter 2	Name of operator port.
Parameter 3	Image handle.
Description	Outputs image to the given port.
Return value	0 : Operation has been completed successfully <0: Cannot output image

Function	`bool vaSi_CustomOp_InputHasImage (va_custom_op_sim_handlesimHandle, const char* portName)`
Parameter 1	Simulation handle provided to the operator simulation function.
Parameter 2	Name of operator input port.
Description	Returns whether there is an input image available at the port with the given name.
Return value	true : Image is available false : No image available

Function	`bool vaSi_CustomOp_OutputReady (va_custom_op_sim_handle simHandle, const char* portName)`
Parameter 1	Simulation handle provided to the operator simulation function.
Parameter 2	Name of operator output port.
Description	Returns whether the output port with the given name may take an image.
Return value	true : Output ready for next image false : Output not ready for taking image

Function	`int vaSi_CustomOp_CreateImage (va_custom_op_sim_handle simHandle, va_image_format_handle format, va_image_handle * newImage)`
Parameter 1	Simulation handle provided to the operator simulation function.
Parameter 2	Image format of the new image.
Parameter 3	Return parameter for image handle.
Description	Creates a blank image based on the format given by parameter 2. Before returning fromthe simulation function this image must either be stored by calling vaSi_CustomOp_StoreImage() or deleted by calling vaSi_CustomOp_DeleteImage().
Return value	0 : OK <0 : Could not create image

Function	`int vaSi_CustomOp_DeleteImage (va_custom_op_sim_handle simHandle, va_image_handle imageHandle)`
Parameter 1	Simulation handle provided to the operator simulation function.
Parameter 2	Handle of image which shall be deleted.
Description	Deletes image referenced by given image handle.
Return value	0 : Operation has been completed successfully <0: Error during deleting image

Function	`int vaSi_CustomOp_StoreImage (va_custom_op_sim_handle simHandle, va_image_handle imageHandle, const char* storeName)`
Parameter 1	Simulation handle provided to the operator simulation function.
Parameter 2	Image handle.
Parameter 3	Name as which the image shall be stored. The image may later be queried by this name.
Description	Stores image in local storage of the operator simulation instance.
Return value	0 : Operation has been completed successfully VA_SIM_CANNOT_STORE_IMAGE: Cannot create storage for image VA_SIM_STORE_NAME_ALREADY_USED: Name ‘storeName’ is already in use for currentlystored image

Function	`int vaSi_CustomOp_GetStoredImagesCount (va_custom_op_sim_handle simHandle, unsigned int *count)`
Parameter 1	Simulation handle provided to the operator simulation function.
Parameter 2	Return parameter for image count.
Description	Returns the number of images which are stored within the operator simulation instance.
Return value	0: OK <0: Can't query information.

Function	`int vaSi_CustomOp_GetStoredImage (va_custom_op_sim_handle simHandle, unsigned int index, va_image_handle *retImage)`
Parameter 1	Simulation handle provided to the operator simulation function.
Parameter 2	Index within the array of stored images.
Parameter 3	Return parameter for image handle.
Description	Get image which has been stored before. The image is removed from the image storage. Before returning from the simulation function this image must either be stored again by calling vaSi_CustomOp_StoreImage() or deleted by calling vaSi_CustomOp_DeleteImage().
Return value	0 : OK <0 : Could not get image

Function	`const char* vaSi_CustomOp_GetNameOfStoredImage (va_custom_op_sim_handle simHandle, unsigned int index)`
Parameter 1	Simulation handle provided to the operator simulation function.
Parameter 2	Index within the array of stored images.
Description	Returns a string of the image name.
Return value	Not NULL : Value is image name string NULL : Could not query name

Function	`int vaSi_CustomOp_GetStoredImageByName (va_custom_op_sim_handlesimHandle, const char* storeName, va_image_handle *retImage)`
Parameter 1	Simulation handle provided to the operator simulation function.
Parameter 2	Name under which the image has been stored.
Parameter 3	Return parameter for image handle.
Description	Get image which has been stored before with the given storage name. The image is removed from the image storage. Before returning from the simulation function this image must either be stored again by calling vaSi_CustomOp_StoreImage() or deleted by calling vaSi_CustomOp_DeleteImage().
Return value	0 : OK <0 : Could not get image

Function	`int vaSi_CustomOp_CreateImageFormat (va_custom_op_sim_handle simHandle, const char* portName, va_image_format_handle *createdFormat)`
Parameter 1	Simulation handle provided to the operator simulation function.
Parameter 2	Name of operator port.
Parameter 3	Return pointer for format handle.
Description	Creates a new image format object and returns a corresponding handle. The format is initialized by the format of the port with the given name. Before returning from the simulation function the format must become deleted by calling vaSi_CustomOp_DeleteImageFormat().
Return value	0 : OK <0 : Could not create format

Function	`int vaSi_CustomOp_CopyImageFormat (va_custom_op_sim_handle simHandle, va_image_format_handle formatHandle, va_image_format_handle *createdFormat)`
Parameter 1	Simulation handle provided to the operator simulation function.
Parameter 2	Handle of format which is being copied.
Parameter 3	Return parameter for format handle.
Description	Creates a new image format object and returns a corresponding handle. The format is initialized by the provided format. Before returning from the simulation function the formatmust become deleted by calling vaSi_CustomOp_DeleteImageFormat().
Return value	Not NULL : Value is format handle NULL : Could not create format

Function	`int vaSi_CustomOp_DeleteImageFormat (va_custom_op_sim_handle simHandle, va_image_format_handle formatHandle)`
Parameter 1	Simulation handle provided to the operator simulation function.
Parameter 2	Handle of format which is being deleted.
Description	Deletes the image format object referenced by the given format handle.
Return value	0: OK <0 : Could not delete format

Function	`int vaSi_Image_GetFormat (va_image_handle imageHandle, va_image_format_handle formatHandle)`
Parameter 1	Image handle.
Parameter 2	Handle of earlier created format which will be set to format of image.
Description	Queries the format of the image referenced by the image handle.
Return value	0 : Operation has been completed successfully <0: Cannot query format

Function	`int vaSi_Image_SetProperty (va_image_handle imageHandle, constchar* propType, const va_data* propData)`
Parameter 1	Image handle.
Parameter 2	String identifying the property which shall be set.
Parameter 3	Pointer to data structure which will be used for setting the new property.
Description	Set property of the image referenced by the image handle. Following properties may be setvia this function: "ImgWidth" : Set image width (propData has type VA_UINT32) "ImgHeight": Set image height (propData has type VA_UINT32)
Return value	0 : Property has been set successfully VA_SIM_INVALID_PARAMETER : Cannot identify propertyVA_SIM_INVALID_TYPE: Property data has wrong format VA_SIM_INVALID_VALUE : Property data has invalid value

Function	`int vaSi_Image_GetProperty (va_image_handle imageHandle, constchar* propType, va_data* propData)`
Parameter 1	Image handle.
Parameter 2	Enum value identifying the property which shall be queried.
Parameter 3	Pointer to data structure which will be used for data communication.
Description	Queries the properties of the image referenced by the image handle. Following propertiesare available: "ImgWidth" : Get image width (propData has type VA_UINT32) "ImgHeight": Get image height (propData has type VA_UINT32)

Return value	0 : Property has been queried successfully VA_SIM_INVALID_PARAMETER : Cannot identify property VA_SIM_INVALID_TYPE: Property data has wrong format VA_SIM_INVALID_VALUE : Property data has invalid value

Function	`int vaSi_Image_SetLineLength (va_image_handle imageHandle, unsignedint line, unsigned int length)`
Parameter 1	Image handle.
Parameter 2	Line number.
Parameter 3	Line length.
Description	Sets the length of the referenced line to an individual value which may differ to the overallimage width (not exceeding the maximum image width defined by the image format).
Return value	0 : Operation has been completed successfully <0: Cannot set line length to the given value

Function	`int vaSi_Image_GetLineLength (va_image_handle imageHandle, unsignedint line, unsigned int *length)`
Parameter 1	Image handle.
Parameter 2	Line number.
Parameter 3	Return parameter for line length.
Description	Returns the length of the referenced line.
Return value	0: OK <0: Cannot query line length

Function	`int vaSi_Image_SetPixelValue (va_image_handle imageHandle, uint64_t imagePos, unsigned int compIndex, int64_t value)`
Parameter 1	Image handle.
Parameter 2	Position within the frame.
Parameter 3	Component index.
Parameter 4	Pixel component value.
Description	Sets the corresponding pixel component to the given value.
Return value	0 : Operation has been completed successfully <0: Error setting the pixel component value

Function	**`int vaSi_Image_GetPixelValue (va_image_handle imageHandle, uint64_t imagePos, unsigned int compIndex, int64_t *value)`**
Parameter 1	Image handle.
Parameter 2	Position within the frame.
Parameter 3	Component index.
Parameter 4	Return parameter for pixel component value
Description	Returns the corresponding pixel component value.
Return value	0 : Operation has been completed successfully <0: Error getting the pixel component value

Function	`int vaSi_ImageFormat_SetProperty (va_image_format_handle formatHandle, const char* propType, const va_data* propData)`
Parameter 1	Image format handle.
Parameter 2	Enum value identifying the property which shall be set.
Parameter 3	Pointer to data structure which holds the new property.
Description	Sets properties of the image format referenced by the handle. Following properties may beset via this function: “Protocol”: Set image protocol where *propData has the type VA_ENUM and is set to oneof the following values: VALT_IMAGE 2D VALT_LINE1D “ColorFormat”: Set image protocol where *propData has the type VA_ENUM and is set toone of the following values: VAF_GRA Y VAF_COL OR “ColorFlavor”: Set image protocol where *propData has the type VA_ENUM and is set toone of the following values: FL_NON E FL_RGB FL_HSI

	FL_YUV FL_LAB FL_XYZ "Parallelism": Set parallelism (type VA_INT32) "ComponentCount": Set number of pixel components (type VA_INT32) "ComponentWidth": Set pixel component width (type VA_INT32) "Arithmetic": Set pixel component arithmetic where *propData has the type VA_ENUM andis set to one of the following values: UNSIGNE DSIGNED "MaxImgHeight": Set max. image height (type VA_INT32) "MaxImgWidth": Set max. image width (type VA_INT32)
Return value	0 : Property has been set successfully VA_SIM_INVALID_PARAMETER : Cannot identify propertyVA_SIM_INVALID_TYPE: Property data has wrong format VA_SIM_INVALID_VALUE : Property data has invalid value

Function	`int vaSi_ImageFormat_GetProperty (va_image_format_handle formatHandle, VAImageFormatProperty propType, va_data* propData)`
Parameter 1	Image format handle.
Parameter 2	Enum value identifying the property which shall be queried.
Parameter 3	Pointer to data structure which will be overwritten by the queried property.
Description	Queries properties of the image format referenced by the handle. The properties whichmay be queried are identical to the ones which can be set through the function vaSi_ImageFormat_SetProperty().
Return value	0 : Property has been queried successfully VA_SIM_INVALID_PARAMETER : Cannot identify property VA_SIM_INVALID_TYPE: Property data has wrong format

Function	`int vaSi_CreateStatusMessage (va_custom_op_sim_handle simHandle, va_status_handle *newMessage)`
Parameter 1	Simulation handle.
Parameter 2	Return parameter for created error message.
Description	Create a status message which may be submitted to the simulation engine.
Return value	0 : OK <0: Can't create message

Function	`int vaSi_SetStatusMessageProperty (va_custom_op_sim_handle simHandle, va_status_handle message, const char* propName, const va_data* propValue)`
Parameter 1	Simulation handle.
Parameter 2	Status message handle.
Parameter 3	Name of property which shall be set.
Parameter 4	New property value
Description	Alter status message property. Following properties may be set via this function:“Code”: Set error code (type VA_INT32) “Severity”: Set severity level where the data (type VA_ENUM) must be one of the followingvalues: VA_INFO VA_WARNING VA_ERROR “Description”: Set string description of status (type VA_STRING)

Function	`int vaSi_SendStatusMessage (va_custom_op_sim_handle simHandle, va_status_handle message)`
Parameter 1	Simulation handle.
Parameter 2	Handle of status message which shall be submitted.
Description	Submitted status message to the simulation engine.
Return value	0 : OK <0: Can’t submit message

7.2 Throughput Analysis

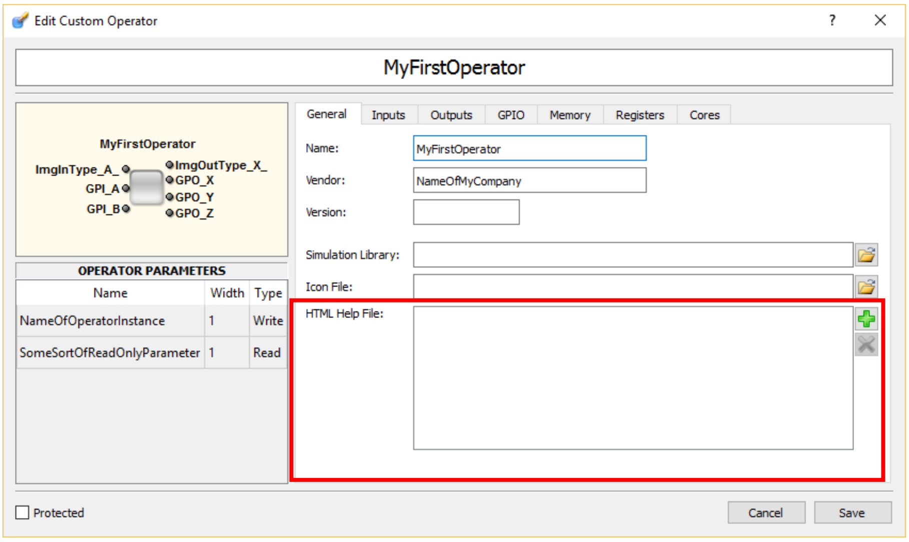
For throughput analysis within VisualApplets the following function must be exported:

```
int AnalyzeThroughputOfOPNAME (va_custom_op_sim_handle simHandle,
                               const char* inPort, const char* outPort,
                               double* throughputRatio)
```

where ***OPNAME*** is the name of the Custom Operator. The function must return a factor **`throughputRatio`** which is the ratio of data rate between input **`inPort`** and output **`outPort`** of the operator. When there is no direct relation between **`inPort`** and **`outPort`** the function must return -1 otherwise 0 is returned.

8 Creating Custom Operator Documentation

Documentation of the operator should be provided as an HTML file. When available, all files which make up the documentation need to be specified under the *General* tab / *HTML Help Files*:



Name: MyFirstOperator
Vendor: NameOfMyCompany
Version:

Name	Width	Type
NameOfOperatorInstance	1	Write
SomeSortOfReadOnlyParameter	1	Read

The first file that is specified is interpreted to be the starting point of the operator's documentation. The naming convention for this file is: <NameOfCustomOperator>.htm.

Make sure you provide a CSS file. Make sure you also provide all related image files.

(In XML, the entry Operator/Info/HtmlHelpFile points to these files.)

You can use the operator template provided in the VisualApplets install directory in subdirectory Examples/CustomLibrary/OperatorTemplate.

9 Completing the Custom Operator

When you have wrapped your HDL code so that its interface matches the generated black box, you need to proceed some last steps for completing your custom operator:

1. Create a netlist out of your implementation.

	Set Add IO Buffer = NO When creating the netlist, make sure that your synthesis tool doesn't automatically add IO buffers. In case you use XST for netlist synthesis you set `Add IO Buffer = NO` Otherwise, the resulting NGC file will cause errors during the VisualApplets build flow.

	Warnings During Netlist Generation When generating the net list, warnings may be output concerning unused IO Ports of the custom operator interface. Unused IO Ports are all ports that were generated according to your operator definition, but are not connected with your IP core. You may ignore these warnings. Examples of this behavior are all custom operator examples you find in the *Examples* directory of your VisualApplets installation: \Examples\CustomLibrary

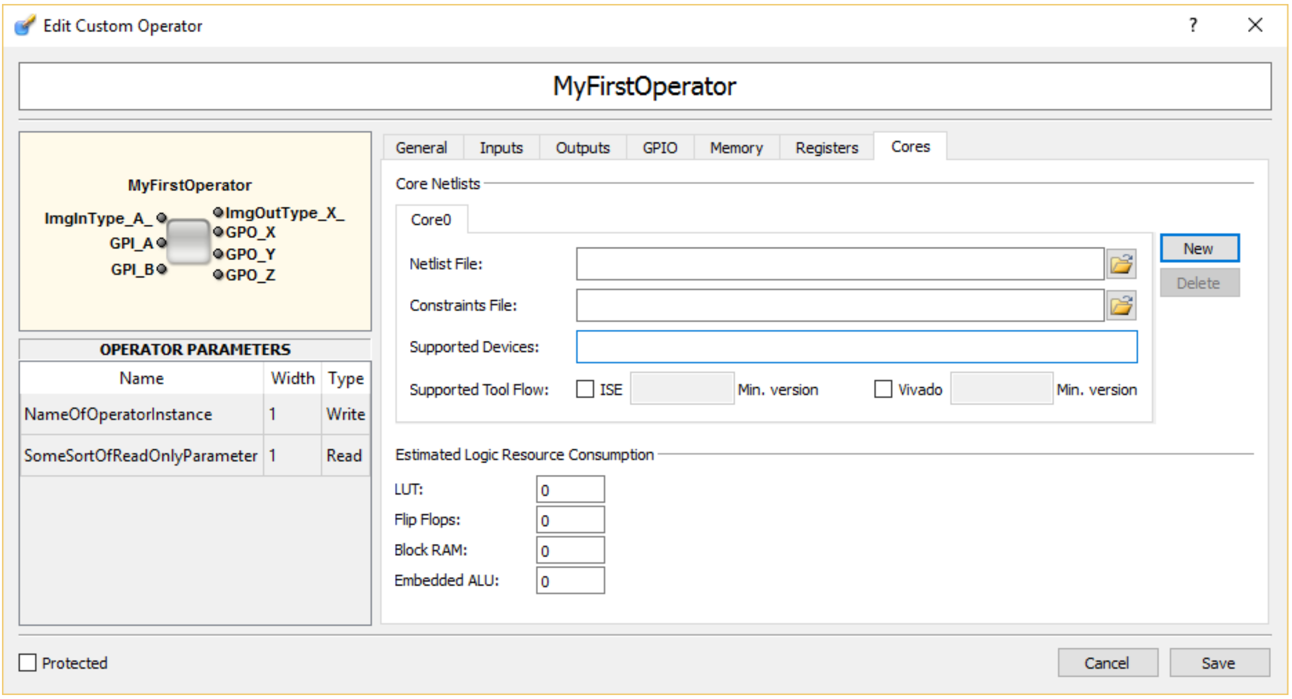
2. If required, also define a constraints file (*.ucf format if you use Xilinx ISE, *.xdc format if you use Xilinx Vivado).
3. Optionally, set up the operator's software interface as described in section 7.
4. Optionally, create the operator documentation as described in section 8.

Now, you need to complete the operator definition in VisualApplets.

To do so, proceed as follows.

Required steps:

1. Go to the *Core* tab.



Core Netlists: Core0

Netlist File:

Constraints File:

Supported Devices:

Supported Tool Flow: ☐ ISE Min. version ☐ Vivado Min. version

OPERATOR PARAMETERS

Name	Width	Type
NameOfOperatorInstance	1	Write
SomeSortOfReadOnlyParameter	1	Read

Estimated Logic Resource Consumption

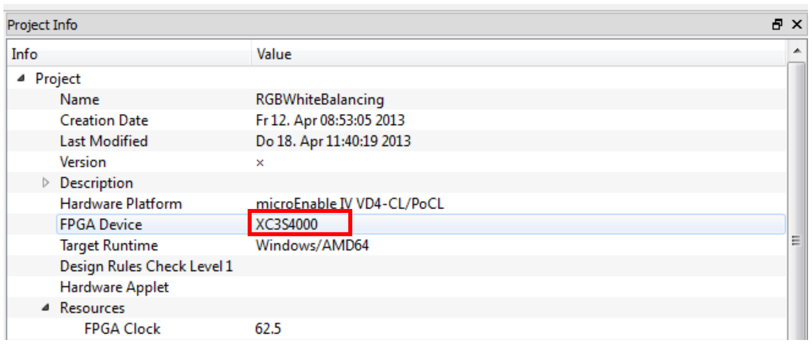
LUT:	0
Flip Flops:	0
Block RAM:	0
Embedded ALU:	0

☐ Protected

2. Specify the netlist file you generated.

3. Specify the constraints file if you defined constraints.

4. Specify the supported devices: The device name is the name of the FPGA type of the target platform. Please use exactly the same spelling as provided in the project info box of VisualApplets. If several FPGA types are supported, use a space separated list of names.



Info	Value
Project	
Name	RGBWhiteBalancing
Creation Date	Fr 12. Apr 08:53:05 2013
Last Modified	Do 18. Apr 11:40:19 2013
Version	x
Description	
Hardware Platform	microEnable IV VD4-CL/PoCL
FPGA Device	XC3S4000
Target Runtime	Windows/AMD64
Design Rules Check Level 1	
Hardware Applet	
Resources	
FPGA Clock	62.5

If a design uses the custom operator, but the FPGA on the target platform is not in this list, the DRC will report an error that the operator is not supported by the target platform.

5. Specify the supported Xilinx Tool(s). You can check the boxes for both ISE and Vivado. Netlists generated with ISE are usually also compatible with the Vivado build flow but you should check whether this is the case for your operator implementation. You need to

define the minimal version number of the tool which supports the given netlist. Typically this would be the version which you used for creating the net list.
If a design uses the custom operator, but the specified tools are not used for building the design, the DRC will report an error that the operator is not supported by the target platform.

Defining Multiple Cores

You can define multiple cores for the same custom operator. This will allow to use device and tool specific implementations of the custom operator so for different target platforms the appropriate implementation is chosen for building an applet.

Optional steps:

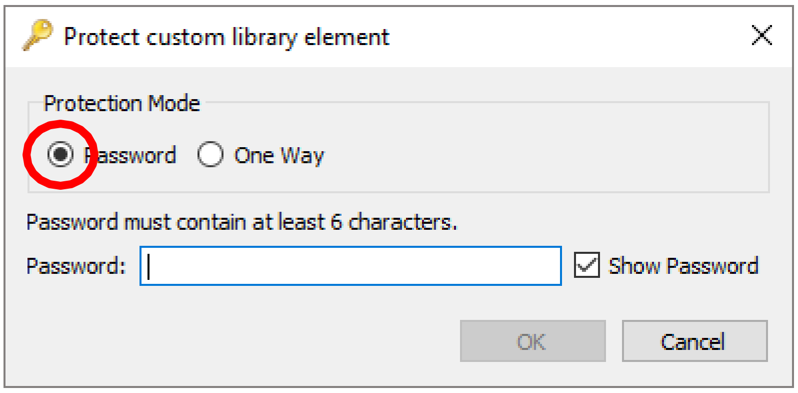
6. Optionally, enter the consumption of logic resources by the operator. Simply enter the values estimated by the Xilinx tools during generation of netlist.

7. Under the General tab, specify the path to your simulation library (the custom operator's software interface).

8. Under the *General* tab, specify the path to the icon file. This is the file that contains the icon that will be used when your custom operator is displayed in VisualApplets.

9. Under the *General* tab, specify all files that make up your custom operator documentation. Make sure you also provide a CSS file and all related image files.

10. If you want to protect your operator design: In the left bottom corner, activate the option "Protected". In the dialog that opens:
    a) Make sure protection mode **Password** is activated.
    b) Enter your password.
    c) Click **OK**.


Protect custom library element

Protection Mode

[✓ Password] One Way

Password must contain at least 6 characters.

Password:

☑ Show Password

You can always protect your custom operator design also at a later point of time, using the context menu of the custom library element.

Protecting Options

After protection has been enabled, the custom operator is made a “black box”.There are two ways to protect a custom operator design:

Protection via password: The custom operator design can afterwards be opened and edited via password. Users that do **not** have the password will notbe able to see any details of the custom operator (black box).

- **Irreversible protection**: If you select protection mode *One-Way*, the customoperator is made a black box forever and cannot be re-opened, not even byyourself.

"One-Way" protection is irreversible: If you select protection mode *One Way* (instead of *Password*), the user library element can never be re-opened, not even by yourself. If you plan to enhance the element at a later point of time, make sure you select protection mode **Password** instead. Alternatively, you can save a copy of the element (as a hierarchical box or a non-protected operator) before enabling this protection mode.

11. Click **Save**.

Now, your new custom operator is ready for being used in designs.

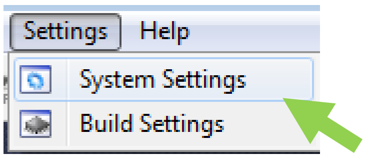
10 Using New Custom Operators

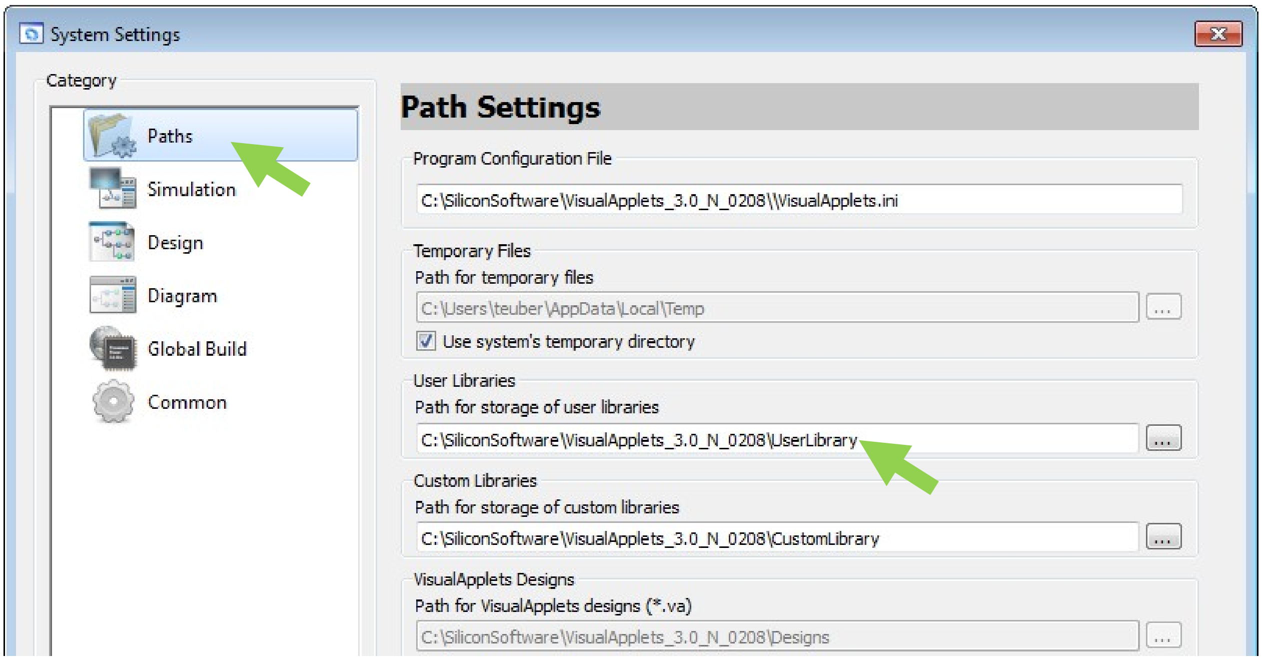
10.1 Distributing the Custom Library or the Individual Custom Operator

A custom library with all contained operators is stored as one single <LibaryName>.vl file. <LibaryName> is the name of the custom library.

This file can be distributed and directly applied in VisualApplets. It simply needs to be copied into the Custom Library directory which is specified in the VisualApplets settings (Settings -> System Settings -> Paths -> Custom Libraries).

1. Copy the new <LibaryName>.vl file to the Custom Library directory of your VisualApplets installation.





Path Settings

Program Configuration File

C:\SiliconSoftware\VisualApplets_3.0_N_0208\\VisualApplets.ini

Temporary Files

Path for temporary files

C:\Users\teuber\AppData\Local\Temp

[✓] Use system's temporary directory

User Libraries

Path for storage of user libraries

C:\SiliconSoftware\VisualApplets_3.0_N_0208\UserLibrary

Custom Libraries

Path for storage of custom libraries

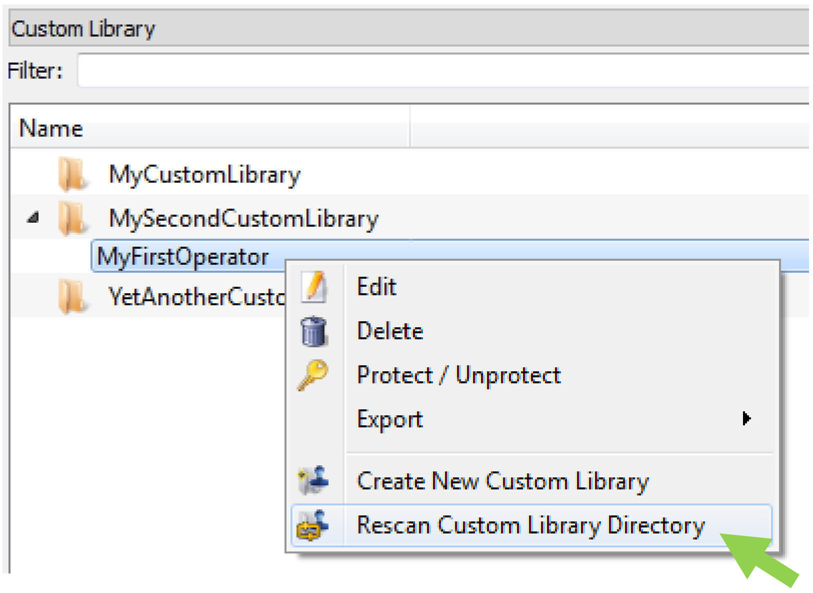
C:\SiliconSoftware\VisualApplets_3.0_N_0208\CustomLibrary

VisualApplets Designs

Path for VisualApplets designs (*.va)

C:\SiliconSoftware\VisualApplets_3.0_N_0208\Designs

2. Re-scan the custom library in the VisualApplets GUI: Right-click on the library name and from the sub menu select **Rescan Custom Library Directory**.



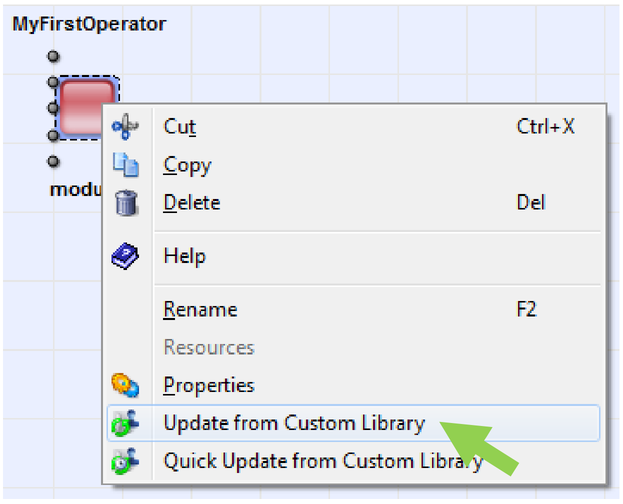
In the VisualApplets examples directory, you find a ready-to-use library called *CustomLibrary.vl* which contains all example operators.

10.2 Update from Custom Library

When you make changes to a custom operator, these changes are not reflected in the designs where you already use the custom operator. Therefore, you need to update the custom operator instances in the designs.

1. Right-click on the operator.

2. From the sub-menu, select **Update from Custom Library** or **Quick Update from Custom Library**.



The update mechanism for Custom Libraries is exactly the same as for User Libraries.

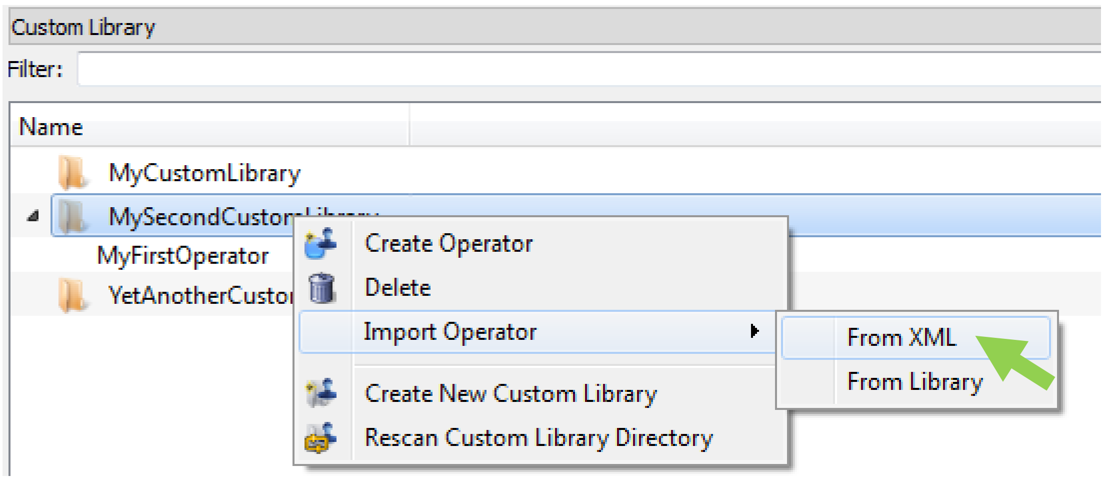
10.3 Importing and Exporting Individual Custom Operators

You can import and export individual custom operators by importing/exporting the XML definition of the operator.

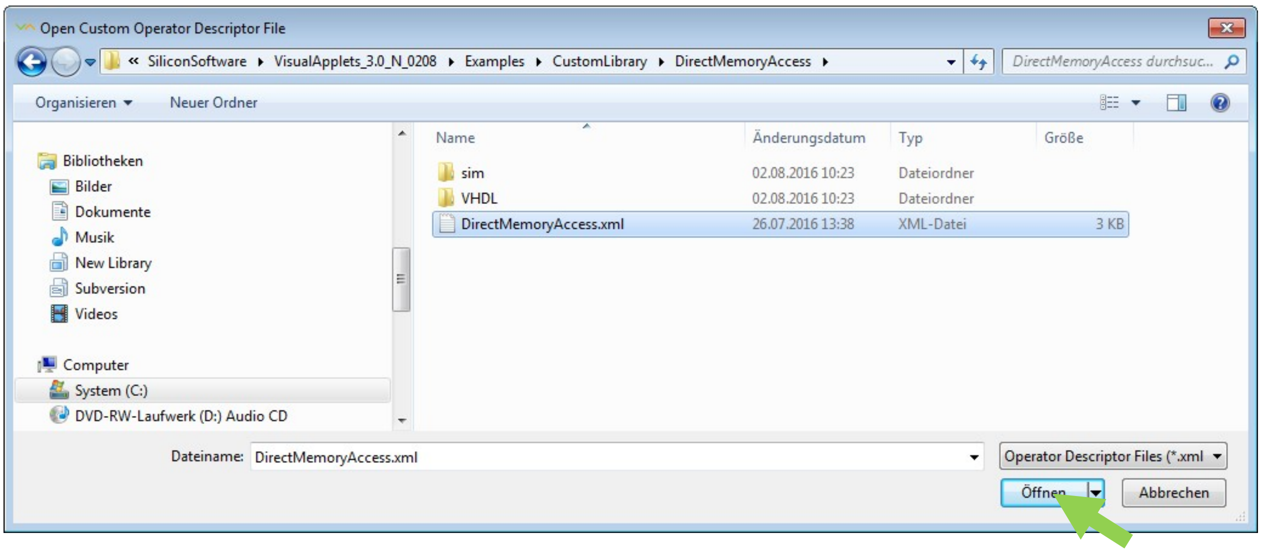
To import a custom operator:

1. Right-click on the custom library where you want to import the custom operator to.

2. From the sub-menu, select **Import Operator** -> **From XML**.



3. Specify the path to the custom operator's XML definition:


Open Custom Operator Descriptor File
« SiliconSoftware ▸ VisualApplets_3.0_N_0208 ▸ Examples ▸ CustomLibrary ▸ DirectMemoryAccess ▸
DirectMemoryAccess durchsuc...
Organisieren
Neuer Ordner
Bibliotheken
Bilder
Dokumente
Musik
New Library
Subversion
Videos
Computer
System (C:)
DVD-RW-Laufwerk (D:) Audio CD
Öffnen
Abbrechen

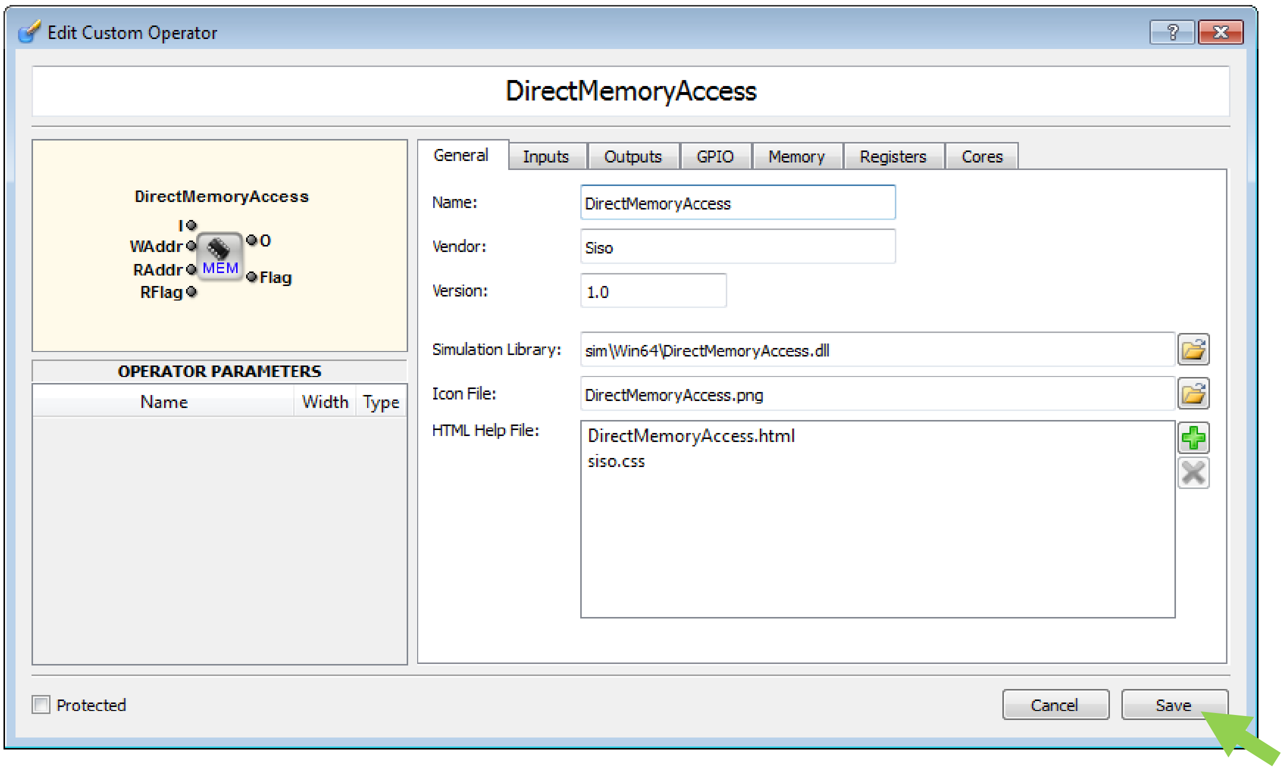
Name	Änderungsdatum	Typ	Größe
sim	02.08.2016 10:23	Dateiordner	
VHDL	02.08.2016 10:23	Dateiordner	
DirectMemoryAccess.xml	26.07.2016 13:38	XML-Datei	3 KB

Dateiname: DirectMemoryAccess.xml

Operator Descriptor Files (*.xml)

4. Click **Open**.

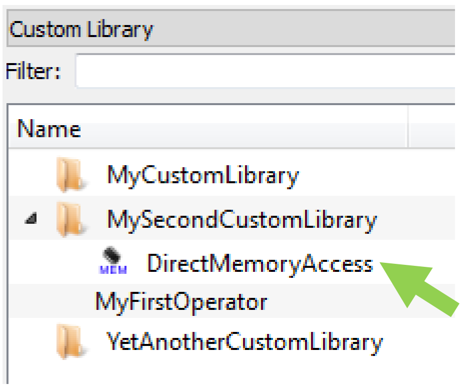
Immediately, the *Edit Custom Operator* dialog opens:



Name: DirectMemoryAccess
Vendor: Siso
Version: 1.0
Simulation Library: sim\Win64\DirectMemoryAccess.dll
Icon File: DirectMemoryAccess.png
HTML Help File: DirectMemoryAccess.html
siso.css

5. Click **Save**.

After saving, the imported operator is directly available in the custom library:



11 Operator Template and Examples

11.1 Examples

In the install directory, you find three completed custom operators which you can use as reference.

You find the examples here:

\Examples\CustomLibrary

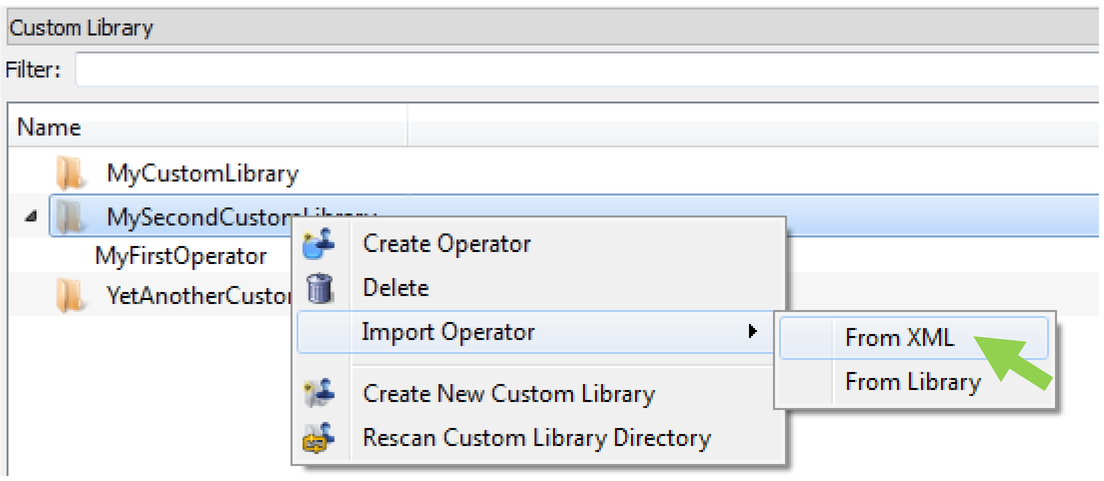
11.2 Custom Operator Template

In the install directory, you find a custom operator template which you can use for defining your custom operators.

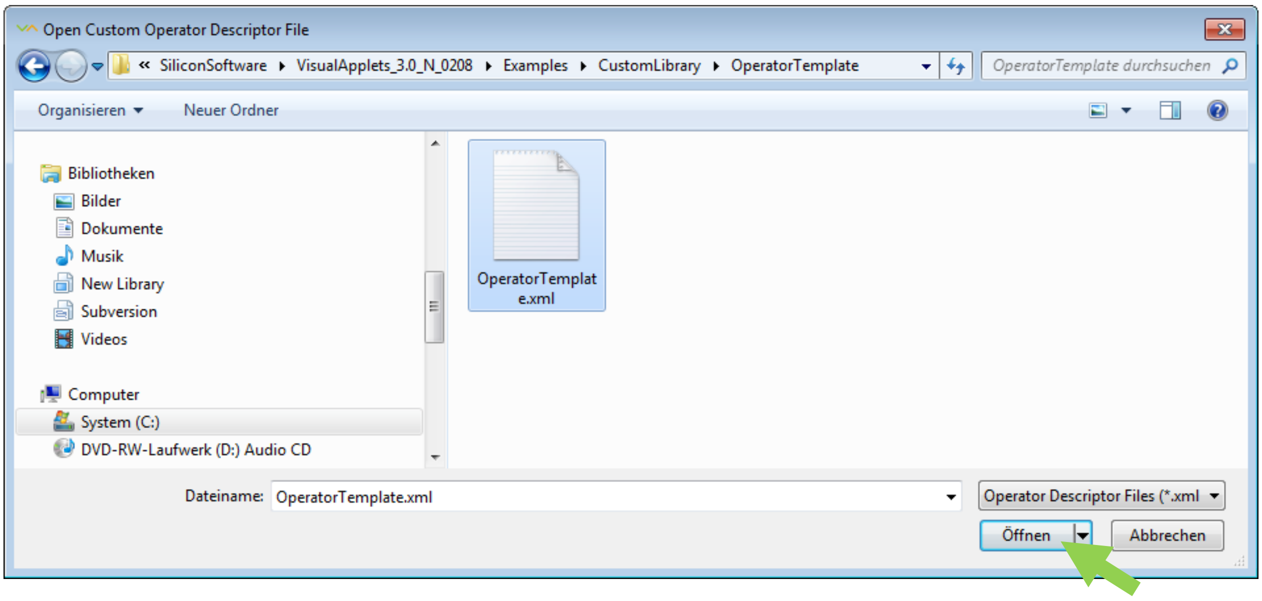
\Examples\CustomLibrary To use the custom operator

template:

1. Right-click on the custom library where you want to create the new custom operator in.
2. From the sub-menu, select **Import Operator** -> **From XML**.



3. Specify the path to the operator template:


Open Custom Operator Descriptor File

« SiliconSoftware ▸ VisualApplets_3.0_N_0208 ▸ Examples ▸ CustomLibrary ▸ OperatorTemplate

OperatorTemplate durchsuchen

Organisieren ▾ Neuer Ordner

Bibliotheken, Bilder, Dokumente, Musik, New Library, Subversion, Videos, Computer, System (C:), DVD-RW-Laufwerk (D:) Audio CD

OperatorTemplate.xml

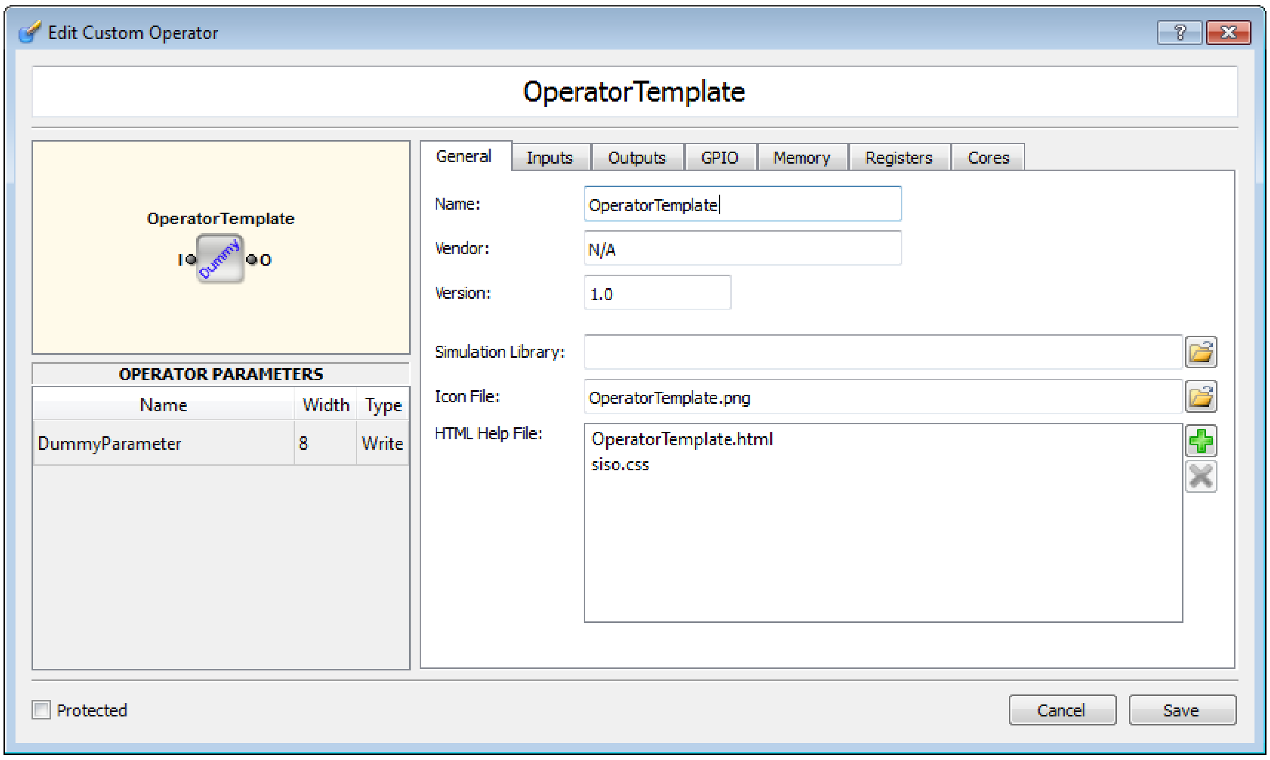
Dateiname: OperatorTemplate.xml

Operator Descriptor Files (*.xml

Öffnen | Abbrechen

4. Click **Open**.

Immediately, the *Edit Custom Operator* dialog opens:


Edit Custom Operator

OperatorTemplate

General | Inputs | Outputs | GPIO | Memory | Registers | Cores

Name: OperatorTemplate

Vendor: N/A

Version: 1.0

Simulation Library:

Icon File: OperatorTemplate.png

HTML Help File: OperatorTemplate.html
siso.css

OPERATOR PARAMETERS

Name	Width	Type
DummyParameter	8	Write

Protected

Cancel | Save

5. Give a name to your new custom operator and proceed as described in section 3.

12 Appendix

12.1 XML Format for Custom Operator Specification

The definition of a custom operator is stored in XML format. A concerning XML file can be exported from the operator library or an operator can be imported using an earlier exported XML file.

In the following, we describe the required parameters where the parameter name is related to an XML tag with the same name. A parameter like **`ImgInInfo`** will translate to an XML entry like: **`<ImgInInfo> ImgInPortNames </ImgInInfo>`** where **`ImgInPortNames`** is the value which in this case would be a sequence of port names. The parameters are hierarchically ordered. In the following tables, lines with gray background will notify the hierarchy position where the parameters are expected.

Simple parameter values can be of following types:

- Choice: the allowed values are **`YES`** or **`NO`**
- String: an ASCII string without whitespace
- Integer
- Floating-point

Some parameters are composed as a structure of values where arrays or records are possible elements for structuring. Arrays are entered by a list of values separated by white space where the values themselves may be structured. Records are entered by a scheme like follows where **`RecordName`** is the record identifier, **`attrX`** are the identifiers for the record entries and **`attrXValue`** are the values:

```
<RecordName attr1=”attr1Value” .. attrN=”attrNValue”/>
```

An example would be providing a record called **`port`** with entries for name and width:

```
<port name=”flag” width=”4”/>
```

The root tag of the XML format is “Operator” with an attribute “name” where the Custom Operator name should be provided:

```
<Operator name=”XYZ”>
     …
</Operator>
```

ⓘ	**Comply with VHDL Naming Conventions** When defining the operator name in the VA GUI, make sure you conform to the VHDL naming conventions. VHDL valid names are defined as follows: “A valid name for a port, signal, variable, entity name, architecture body, or similar object consists of a letter followed by any number of letters or numbers, without space. A valid name is also called a named identifier. VHDL is not case sensitive. However, an underscore may be used within a name, but may not begin or end the name. Two consecutive underscores are not permitted.“

Parameter Name	Type	Description
	Operator/Info	
Vendor	String	Name of Vendor.
Version	String	Version number of the operator. The value can be freely chosen and is intended for version identification by the user.
Cores	Array of String	List of core netlists for the operator. The first string must be *Core0* and must always be there. If more than one core is available the naming convention for them is Core<*N*> where <*N*> is a integer number incremented with every core.
LibraryFile	String	Quoted name of file containing software library (dynamic link library) containing the high-level simulation model for the operator.

Parameter Name	Type	Description
IconFile	String	Quoted name of file containing the operator icon.
HtmlHelpFiles	Array of String	List of quoted file names which contain help content (html + images). The first file is considered as the main HTML file.
Operator/IO		
RegInInfo	Array of String	List of names of later defined info structures (Operator/RegIn) describing write register ports.
RegOutInfo	Array of String	List of names of later defined info structures (Operator/RegOut) describing read register ports.
ImgInSyncMode	String	String defining whether the inputs at the ImgIn ports are synchronous or asynchronous to each other. This string may either be “Sync” or “Async”.
ImgInInfo	Array of String	List of names of later defined info structures (Operator/ImgIn) describing the properties of the image input ports. Several list entries may refer to the same structure which then means that several ports of the same kind of image input interface are available.
ImgOutInfo	Array of String	List of names of later defined info structures (Operator/ImgOut) describing the properties of the image output ports. Several list entries may refer to the same structure which then means that several ports of the same kind of image output interface are available.
GPIn	Array of String	List of pin names for general purpose signal inputs.

Parameter Name	**Type**	**Description**
GPOut	Array of String	List of pin names for general purpose signal outputs.
MemInfo	Array of String	List of names of later defined info structures (Operator/Mem) describing the properties of the memory interface ports. Several list entries may refer to the same structure which then means that several ports of the same kind of memory interface are available.
	Operator/Properties	
NrLut	Integer	Number of FPGA LUT elements consumed by the operator
NrRegs	Integer	Number of FPGA registers consumed by the operator
NrBlockRam	Integer	Number of block ram elements consumed by the operator
NrEmbeddedMult	Integer	Number of embedded multipliers consumed by the operator

Image input port specification is done by following syntax within the configuration file:

```
<ImgIn name=”IMG_IN_IDENTIFIER”> Parameters </ImgIn>;
```

Here **`IMG_IN_IDENTIFIER`** is one of the image input port names which have been provided in the above parameter *Operator/IO/ImgInInfo*. The content *Parameters* is specifying the properties of the image interface port:

Parameter name	Type	Description
Operator/ImgIn		
Width	Integer	Width of the image data port
FIFODepth	Integer	Depth of the buffer FIFO for input data which at least needs to be provided by the VA core. The value must be a power of two minus 1 between 15 and 1023.
Formats	Array of Record	List of image format records *ImgFormat* which are supported by the port. For the naming scheme of image formats see below.

The image format records have the following structure:

```
          <ImgFormat name="FORMAT" maxWidth="X1"
                      maxHeight="Y1"alias="NAME1"/>
```

The entry **FORMAT** is a String value for an image format coded by the below discussed naming scheme for image formats. The attributes **maxWidth** and **maxHeight** are optional and fix the limits of image size. If they are not present, the image size constraints can be freely chosen by the user within VisualApplets later on. The attribute **alias** is optional as well and, if present, defines the name under which the format will be displayed in the GUI.

Image output port specification is done by following syntax within the configuration file:

```
          <ImgOut name="IMG_OUT_IDENTIFIER"> Parameters </ImgOut>;
```

Here **IMG_OUT_IDENTIFIER** is one of the image output port names which have been provided in the above parameter *Operator/IO/ImgOutInfo*. The content *Parameters* is specifying the properties of the image interface port:

Parameter Name	Type	Description
Operator/ImgOut		
Width	Integer	Width of the image data port
FIFODepth	Integer	Depth of the buffer FIFO for output data which at least needs to be provided by the VA core. The value must be a power of two minus 1 between 15 and 1023.
Formats	Array of Record	List of image format records *ImgFormat* which are supported by the port. For the naming scheme of image formats see below.

Image formats are coded by the following naming scheme:

```
{BaseFormat}{BitsPerPixel}x{Parallelism}
```

Optionally there can be suffixes for image dimension and the notification of signed component data:

```
{BaseFormat}{BitsPerPixel}x{Parallelism}x{Dimension}{Sign}
```

The meaning of the dimension is as follows:

- ***Dimension* = 2** – a two-dimensional image means that the image is structured bothby end-of-line and end-of-frame markers.
- ***Dimension* = 1** – a one-dimensional image means that there are no end-of-frame markers which divide the incoming lines into frames.

When no dimension is specified a value of two is assumed. The suffix *{Sign}* can be **s** for signed pixel components or **u** for unsigned values where the default value is **u** when no such suffix is provided. Supported color formats are **rgb**, **yuv**, **hsi**, **lab** and **xyz**.

Examples are:

- **gray8x4** – gray format with 8-bit pixel and parallelism 4
- **rgb24x2** – rgb color format with 3x8-bit pixel and parallelism 2

- **gray16x1** – gray format with 16-bit pixel, only single pixel in a data word
- **gray8x4x1** – one dimensional gray image with 8-bit per pixel and parallelism 4
- **gray16x1s** – gray image with signed 16-bit components, only single pixel in a data word

Register input port specification is done by following syntax within the configuration file:

```
<RegIn name="REG_IN_IDENTIFIER"> Parameters </RegIn>;
```

Here **`REG_IN_IDENTIFIER`** is one of the register input port names which have been provided in the above parameter *Operator/IO/RegInInfo*. The content *Parameters* is specifying the properties of the register interface port:

Parameter Name	**Type**	**Description**
	Operator/RegIn	
Width	Integer	Width of the register port

Register output port specification is done by following syntax within the configuration file:

```
<RegOut name="REG_OUT_IDENTIFIER"> Parameters </RegOut>;
```

Here **`REG_OUT_IDENTIFIER`** is one of the register output port names which have been provided in the above parameter *Operator/IO/RegOutInfo*. The content *Parameters* is specifying the properties of the register interface port:

Parameter name	**Type**	**Description**
	Operator/RegOut	
Width	Integer	Width of the register port

Memory interface specification is done by sections with following syntax within the configuration file:

```
<Mem name="MEM_IDENTIFIER"> Parameters </MEM>
```

Here **`MEM_IDENTIFIER`** is one of the memory port names which have been provided in the above described parameter *Operator/IO/MemInfo*. The content *Parameters* is specifying the properties of the memory interface:

Parameter name	**Type**	**Description**
	Operator/Mem	
DataWidth	Integer	Data width
AddrWidth	Integer	Address width
WrFlagWidth	Integer	Width of flag for marking write accesses. This parameter must be >= 1.
RdFlagWidth	Integer	Width of flag for marking read accesses. This parameter must be >= 8.
WrCntWidth	Integer	Width of port for communicating the number of available write commands
RdCntWidth	Integer	Width of port for communicating the number of available read commands
SyncMode	String	This parameter signals the relation of the memory interface clock and the design clock. Following values are possible: "SyncToDesignClk" – memory interface ports are synchronous to iDesignClk. "SyncToDesignClk2x" – memory interface ports are synchronous to iDesignClk2x.

Specification of IP core netlists is done by sections with following syntax within the configuration file:

```
<Core name="CORE_IDENTIFIER"> Parameters </Core>
```

Here **`Core_IDENTIFIER`** is one of the core names which have been provided in the above described parameter *Operator/Cores*. The content *Parameters* is specifying the properties of the IP core:

Parameter name	**Type**	**Description**
	Operator/Core	
Devices	Array of String	List of FPGA device names which are supported by the core (Example: “XC3S1600E XC3S4000”).
NetlistFile	String	Quoted UTF-8 encoded file name for the net list.
ConstraintsFile	String	Quoted UTF-8 encoded file name for an optional constraints file.
MinVersionISE	String	Minimum version number of ISE tool flow which can use the given netlist (Example: “14.6” for ISE 14.6). If ISE is not supported this string is empty.
MinVersion	String	Minimum version number of Vivado tool flow which can use the given netlist (Example: “2014.4” for Vivado 2014.4). If Vivado is not supported this string is empty.

Contact Details

Europe, Middle East, Africa	The Americas	Asia-Pacific
Basler AG Konrad-Zuse-Ring 28 68163 Mannheim Germany	Basler Inc. 855 Springdale Drive, Suite 203 Exton, PA 19341 USA	Basler Asia Pte. Ltd. 35 Marsiling Industrial Estate Road 3 #05–06 Singapore 739257
Tel.: +49 (0) 621 789507 0 Fax: +49 (0) 621 789507 10	Tel. +1 610 280 0171 Fax +1 610 280 7608	Tel. +65 6367 1355 Fax +65 6367 1255
support.europe@baslerweb.com	support.usa@baslerweb.com	support.asia@baslerweb.com

https://www.baslerweb.com/en/sales-support/support-contact/

Disclaimer