See all versions of this document

Model Composer User Guide

UG1262 (v2020.1) June 3, 2020

XILINX®

Revision History

The following table shows the revision history for this document.

Section	Revision Summary
06/03/2020 Version 2020.1	
Generating C++ Code	Updated for release 2020.1

Supported MATLAB versions and Operating Systems

Xilinx Model Composer supports MATLAB® versions:

- 2019a
- 2019b
- 2020a

The following operating systems are supported on x86 and x86-64 processor architectures:

- Microsoft Windows 10.0 1809 Update; 10.0 1903 Update; 10.0 1909 Update (64-bit), English/Japanese
- Red Hat Enterprise Workstation/Server 7.4, 7.5, 7.6, 7.7 (64-bit)
- Ubuntu Linux 16.04.5 LTS;16.04.6 LTS; 18.04.1 LTS; 18.04.2 LTS, 18.04.3 LTS; 18.04.4 LTS (64-bit)

***Note*:** Xilinx Model Composer with MATLAB version R2020a does not support RHEL 7.4 OS.

Table of Contents

Chapter 1

Introduction

What is Model Composer

Model Composer is a Model-Based design tool that enables rapid design exploration within the MathWorks Simulink® environment and accelerates the path to production for Xilinx® programmable devices through automatic code generation.

Simulink, an add-on product to MATLAB®, provides an interactive, graphical environment for modeling, simulating, analyzing and verifying system-level designs. Model Composer is built as a Xilinx toolbox that fits into the MathWorks Simulink environment and allows algorithm developers to fully leverage all the capabilities of Simulink's graphical environment for algorithm design and verification.

You can express your algorithms in Simulink using blocks from the Model Composer library as well as custom user-imported blocks. Model Composer transforms your algorithmic specifications to production-quality IP implementations using automatic optimizations and leveraging the high-level synthesis technology of Vivado® HLS. Using the IP integrator in the Vivado Design Suite you can then integrate the IP into a platform that, for example, may include a Zynq® device, DDR3 DRAM, and a software stack running on the Arm® processor.

Model Composer provides a library of over 80 optimized blocks for use within the Simulink environment. These include basic functional blocks for expressing algorithms like Math, Linear Algebra, Logic and Bit-wise operations and others. It also includes a number of application-specific blocks for Image Processing and Computer Vision. The Xilinx Model Composer block library contains the following categories of elements.

Table 1: **Xilinx Model Composer Block Library**

Library	Description
Computer Vision	Blocks that support the analysis, manipulation and optimization of a digitized image.
Logic and Bit Operations	Blocks that supports the compound logical operations and bit-wise operations.
Lookup Tables	Block set that performs a one dimensional lookup operation with an input index.
Math Functions	Blocks that implement mathematical functions.

Table 1: **Xilinx Model Composer Block Library** *(cont'd)*

Library	Description
Ports and Subsystems	Blocks that allow creation of subsystems and input/output ports.
Relational Operations	Block set to define some kind of relation between two entities (e.g., Numerical Equality and inequalities).
Signal Attributes	Includes block which helps to maintain the compatibility between input type and output type (e.g., Type casting).
Signal Operations	Blocks that support simple modifications to the time variable of the signal to generate new signals (e.g., Unit Delay).
Signal Routing	Blocks that supports the setup to track signal sources and destinations (e.g., Bus selector).
Sinks	Include blocks that receive physical signal output from other blocks.
Source	Include blocks that generate or import signal data.
Tools	Include blocks that controls the implementation/Interface of the Model.

For information on specific blocks in the Xilinx Model Composer block library, see Appendix B: Model Composer Block Library.

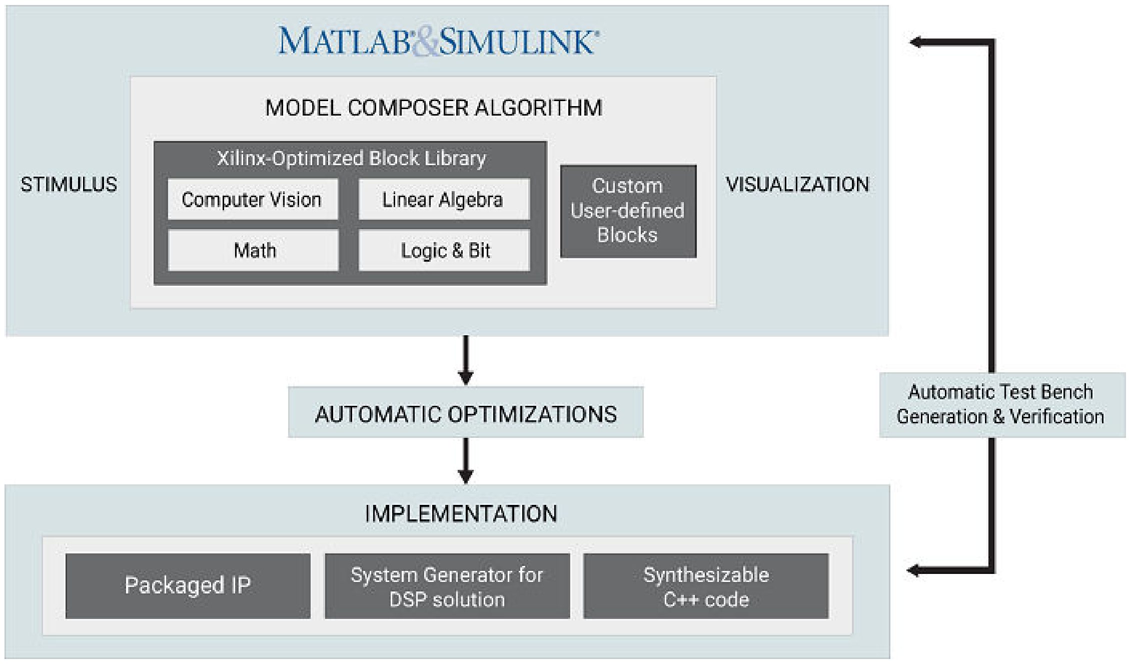
The Model Composer block library is compatible with the standard Simulink block library, and these blocks can be used together to create models that can be simulated in Simulink. However, only certain Simulink blocks are supported for code generation by Model Composer. The Simulink blocks compatible with output generation from Model Composer can be found in the Xilinx Model Composer block library.

Model Composer also lets you create your own custom blocks from existing C/C++ code for use in models. Refer to Chapter 3: Importing C/C++ Code as Custom Blocks for more information.

Your desgin in Model Composer is bit-accurate with regard to the final implementation in hardware, though untimed. You can compile the design model into C++ code for synthesis in Vivado HLS, create System Generator blocks, or create packaged IP to be used in the Vivado Design Suite.

The general tool flow is shown in the following diagram.

Figure 1: **Model Composer Tool Flow**



To help you learn Model Composer, the *Model Composer Tutorial: Model-Based Design Using Model Composer* (UG1259) includes labs and data to walk you through the tool.

The rest of this document discusses the following topics:

- Creating a Model Composer model using block libraries.
- Importing existing C-code into the Model Composer block library for use in your models.
- Compiling the model for use in downstream design tools.
- Verifying your Model Composer model, C++, and RTL outputs.

What's New and Limitations

For information related to what is new for a specific release of Model Composer, refer to *What's New* at https://www.xilinx.com/products/design-tools/vivado/integration/model-composer.html#new.

In addition, while Model Composer is a toolbox built onto the MathWorks Simulink environment, there are certain features of Simulink that are not supported in Model Composer. The following is a list of some of the unsupported features:

- Simulink Performance Advisor.
- Model referencing.

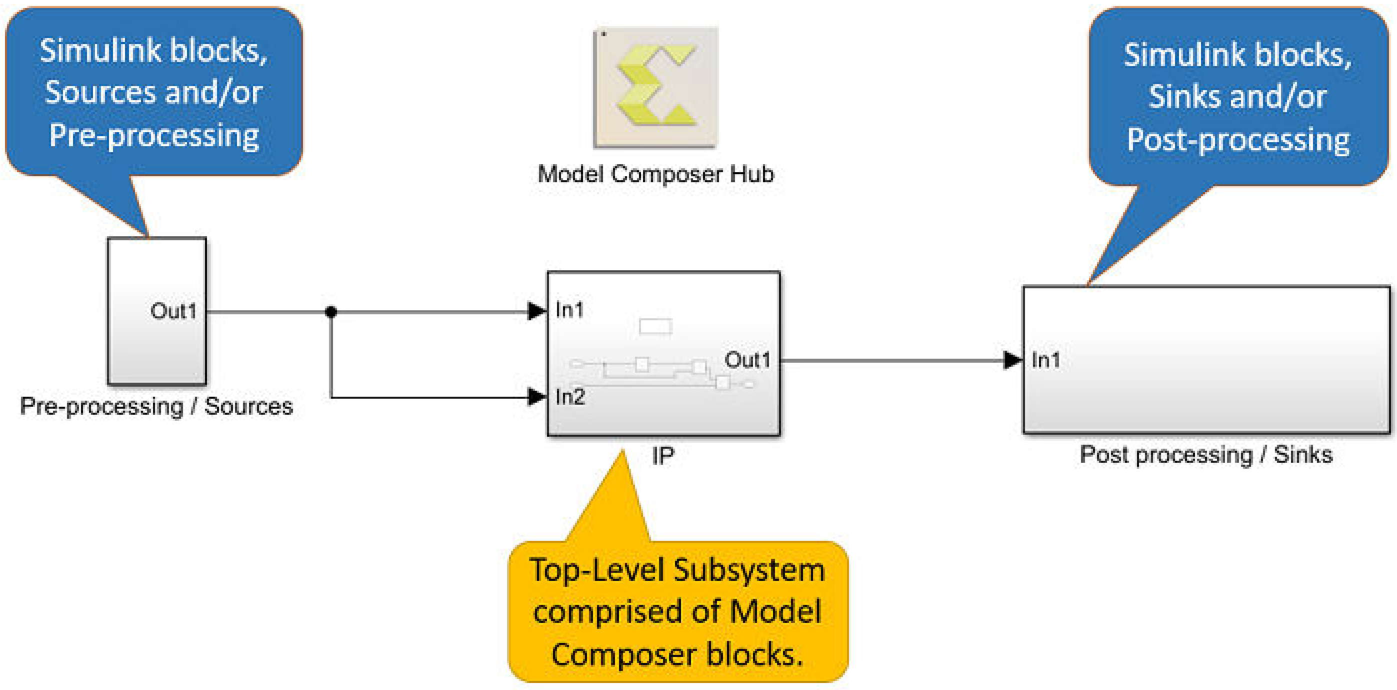
- Variant subsystems.
- Model Composer blocks do not support Simulink fixed-point types and only support Xilinx® fixed-point types.
- Fixed-point designer does not integrate with Model Composer.
- Accelerator mode and Rapid Accelerator mode.

Creating a Model Composer Design

As shown in the image below, a Model Composer design includes the following elements:

1. Simulink® blocks that determine inputs, and provide source signals. These blocks are used in simulation of the design, but do not affect the output generated by Model Composer.
2. A top-level subsystem block, as described in Creating a Top-Level Subsystem Module, that encapsulates the algorithm defined by the Model Composer model. This subsystem module can contain:
   - Blocks from the Model Composer block library to define your algorithm, as listed in Appendix B: Model Composer Block Library.
   - Custom imported functions as described in Chapter 3: Importing C/C++ Code as Custom Blocks.
   - An Interface Spec block that defines the hardware interfaces as described in Defining the Interface Specification.
3. The Model Composer Hub block that controls throughput of the design, and output generation through a series of options as described in Adding the Model Composer Hub.
4. The output signals, or sinks that process the output in Simulink. Again, these blocks are used during Simulink simulation as described in Chapter 5: Simulating and Verifying Your Design, but do not affect the output generated by Model Composer.

Figure 2: **Elements of a Model Composer Design**


Simulink blocks, Sources and/or Pre-processing
Model Composer Hub
Simulink blocks, Sinks and/or Post-processing
Out1
Pre-processing / Sources
In1
In2
Out1
IP
In1
Post processing / Sinks
Top-Level Subsystem comprised of Model Composer blocks.

Launching Model Composer

You can launch Model Composer directly from the desktop icon, or from the command line. Double-click the Model Composer icon, launch it from the Start Menu in Windows operating system, or use the following command from the command prompt:

```
model_composer
```

TIP: *The command-line use of Model Composer requires that the command shell has been configured as follows. You must change directory to* `<install_dir>/Model_Composer/<version>` *and run the* `settings64-Model_Composer.bat` *(or* `.sh`*) file. Where* `<install_dir>` *is the installation folder and* `<version>` *is the specific version of the tool.*

MATLAB opens, and the Model Composer library and features are overloaded onto this environment. At this point you can click the Simulink icon from the main toolbar menu, or type Simulink from the MATLAB command prompt:

```
>> simulink
```

The Simulink **Start Page** is displayed. Model Composer features are loaded onto the Simulink platform.

Model Composer supports the latest releases of MATLAB. For more information on Supported MATLAB Versions and operating systems, refer to Supported MATLAB versions and Operating Systems. If you have multiple versions of MATLAB installed on your system, the first version found in your PATH will be used by Model Composer. You can edit the PATH to move the preferred version of MATLAB to precede other versions. You can also direct Model Composer to open a specific version of the tool using the `-matlab` option as follows:

```
model_composer -matlab C:\Progra~1\MATLAB\R2018b
```

TIP: *When you specify the path to the MATLAB version, do not specify the full path to the executable (`bin/MATLAB`). The string "C:\Progra~1\" is a shortcut to "C:\Program Files\" which eliminates spaces from the command path. The command-line use of Model Composer requires that the command shell has been properly configured as previously discussed.*

Creating a New Model

You create a new model by adding blocks from the Library Browser into the Simulink Editor. You then connect these blocks with signal lines to establish relationships between blocks. The Simulink Editor manages the connections with smart guides and smart signal routing to control the appearance of your model as you build it. You can add hierarchy to the model by encapsulating a group of blocks and signals as a subsystem within a single block. Model Composer provides a set of predefined blocks that you can combine to create a detailed model of your application.

In the Simulink start page, select **Blank Model** to open a new model.

TIP: *You can also open an existing Model Composer template if any have been defined. Model templates are starting points to reuse settings and block configurations. To learn more about templates, see Create a Template from a Model in the Simulink documentation.*

The Simulink start page also lists the recent models that you have opened on the left-hand column. You can open one of these recent models if you prefer.

The blank model opens, and you will create the Model Composer model by adding blocks, specifying block parameters, and using signal lines to connect the blocks to each other.

IMPORTANT! *Model Composer only supports one sample time for the model, and does not support multi-time systems. All Model Composer blocks inherit the sample time from the source block of the model. See What is Sample Time for more information.*

To save the model select **File→Save** from the main menu. The **Save As** dialog box is opened, with a file browser. Navigate to the appropriate folder or location, and enter a name for the model in the **File Name** field. Click **Save**. The model is saved with the file extension `.slx`.

Model Composer also includes example models that can be accessed from the *Model Composer Examples* section of the *Xilinx Model Composer* documentation available from the **Help** menu in the tool, or by typing the `xmcOpenExample` command from the MATLAB command prompt:

```
>> xmcOpenExample
```

This command returns a list of available examples that can be opened. The `xmcOpenExample` command copies the example model to a specified target directory, or to a temp directory if no target is specified, and opens the model to explore the design in Model Composer. The following shows how to specify an example and target directory:

```
xmcOpenExample('importing_c_code','C:\Data\importing_code')
```

TIP: *If the specified target directory does not exist, Model Composer will create it.*

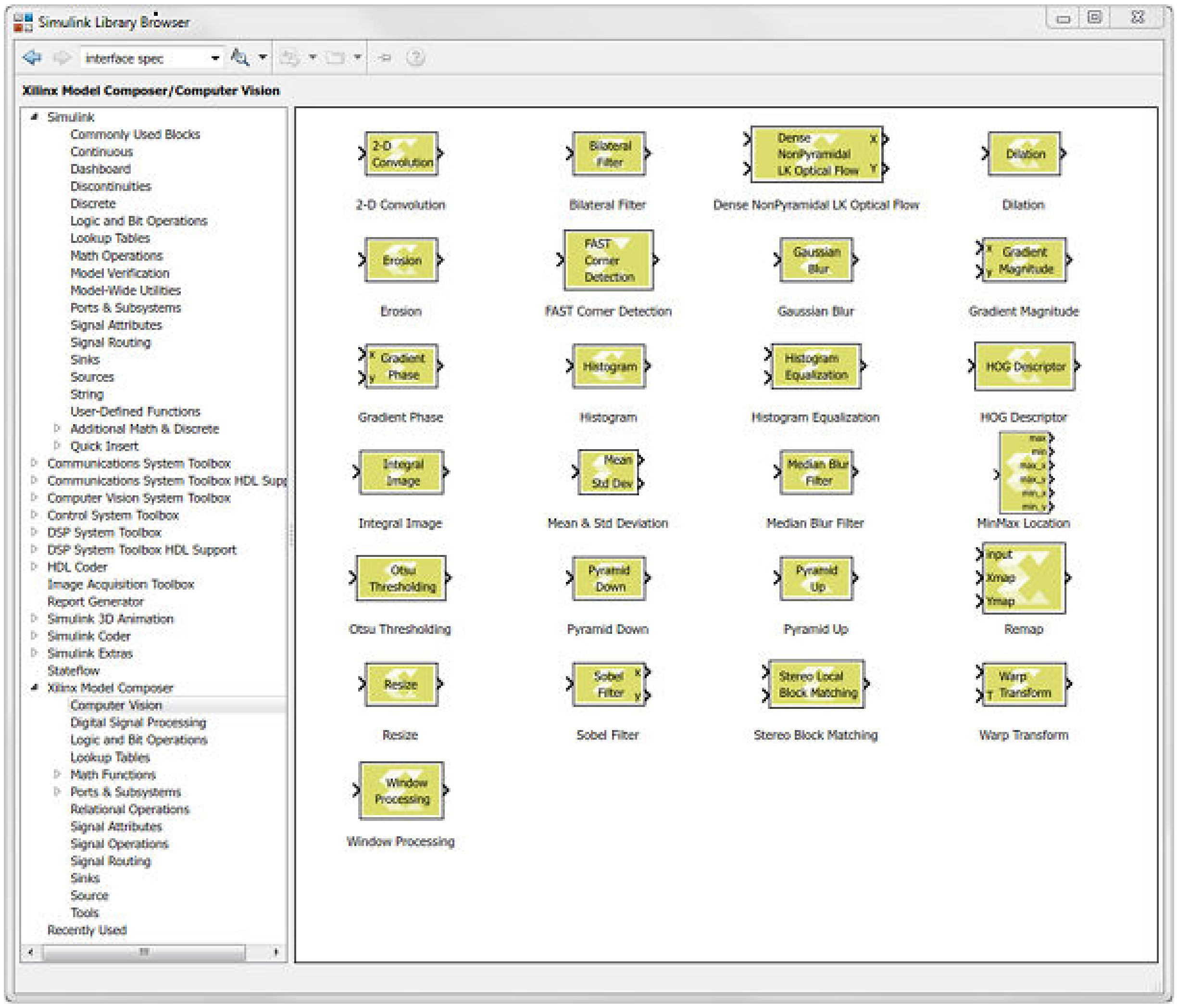
Adding Blocks to a Model

You can add blocks to the current model by opening the Library Browser and dragging and dropping the block onto the design canvas of the Simulink Editor. Open the Library Browser by clicking the button, or by selecting the **View → Library Browser** command from the main menu. You will see the standard Simulink library of blocks, as well as the Xilinx Model Composer library.

TIP: *You can also open the Library browser by typing the slLibraryBrowser command from the command prompt.*

The Xilinx Model Composer blocks are organized into sub-categories based on functionality. The figure below shows the **Xilinx Model Composer → Computer Vision** block library in the Library Browser.

Figure 3: **Library Browser**



Double-clicking a block in the Library Browser, opens the Block Parameter dialog box displaying the default values for the various parameters defined on the selected block. While the block is in the library, you can only view the parameters. To edit the parameters, you must add the block to the design canvas.

To get additional information about a block you can right-click a block in the Library Browser and select the **Help** command. Alternatively, you can double-click the block in the Library Browser and click the **Help** button from the block dialog box. The Help browser opens with specific information for the block.

When you drag and drop the block onto the canvas, the block is added to the model with the default parameter values defined.

TIP: *You can also quickly add blocks to the current model by single-clicking on the design canvas of the Simulink Editor and typing the name of a block. Simulink displays possible matches from the libraries, and you can select and add the block of interest.*

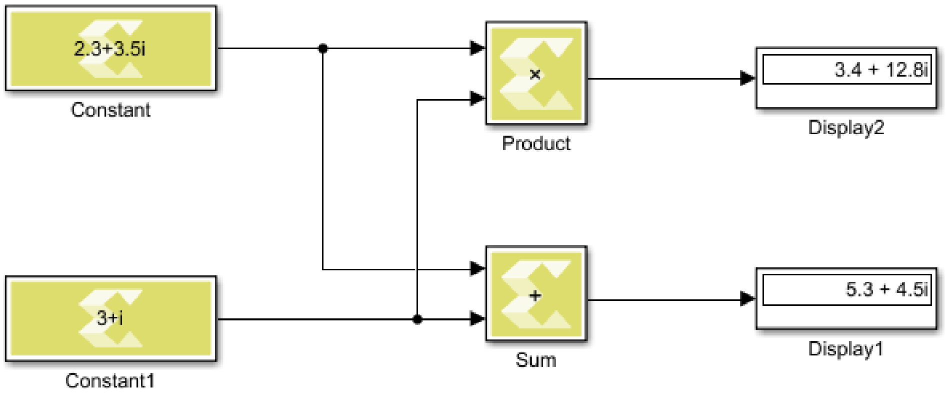
Simulink models contain both signals and parameters. Signals are represented by the lines connecting blocks. Parameters are coefficients that define key characteristics and behavior of a block.

Connecting Blocks

You can connect the output ports on blocks to the input ports of other blocks with signal lines. Signal lines define the flow of data through the model. Signals can have several attributes:

- Data type: Defines the type of data carried by the signal. Values can range from integer, to floating point, to fixed point data types. See Working with Data Types for more information.
- Signal dimension: Defines the values as being scalar, vector, or matrices. See Signal Dimensions and Matrices, Vectors, and Scalars for more information.
- Complexity: Defines a value as being a complex or real number. See Signal Values for more information. The figure below shows complex numbers propagating through a model.

Figure 4: **Complex Signal Values**



To add a signal line, position the cursor over an input or output port of a Simulink block. The cursor changes to a cross hair (+). Left-click and drag the mouse away from the port. While holding down the mouse button, the connecting line appears as a dotted line as you move across the design canvas. The dotted line represents a signal that is not completely connected.

Release the mouse button when the cursor is over a second port to be connected. If you start with an input port, you can stop at an output port, or connect to another signal line; if you start at an output you can stop at an input. Simulink connects the ports with a signal line and an arrow indicating the direction of signal flow.

You can connect into an existing line by right-clicking and dragging the mouse. This creates a branching line connected to the existing signal line at the specified location. The branch line can connect to an input or output as appropriate to the connected signal.

TIP: *You can also connect blocks by selecting them sequentially while holding the Ctrl key. This connects the output of the first block into the input of the second block. Keeping the Ctrl key pressed and selecting another block continues the connection chain.*

Simulink updates the model when you **Run** simulation. You can also update the model using the **Simulation → Update Diagram** menu command, or by typing `Ctrl+D` at any point in the design process. You will see the Data Types, Signal Dimensions and Sample Times from the source blocks propagate through the model.

TIP: *You can use the* ***Display → Signals and Ports*** *menu command to enable the various data that you want displayed in your model, such as* ***Signal Dimensions*** *and* ***Port Data Types****.*

You cannot specify sample times on Model Composer blocks, except for the Constant block from the Source library of the Xilinx Model Composer block library. Model Composer infers the sample time from the source blocks connected at the input of the model, and does not support multiple sample times in the Model Composer design.

By updating the diagram from time to time, you can see and fix potential design issues as you develop the model. This approach can make it easier to identify the sources of problems by limiting the scope to recent updates to the design. The **Update Diagram** is also faster than running simulation.

Working with Data Types

Data types supported by Model Composer include the following:

Table 2: **Model Composer Data Types**

Name	Description
double	Double-precision floating point
single	Single-precision floating point
half*	Half-precision floating point
int8	Signed 8-bit integer
uint8	Unsigned 8-bit integer
int16	Signed 16-bit integer
uint16	Unsigned 16-bit integer
int32	Signed 32-bit integer
uint32	Unsigned 32-bit integer
fixed*	Signed and unsigned fixed point

Table 2: **Model Composer Data Types** *(cont'd)*

Name	Description
boolean	For this data type, Simulink represents real, nonzero numeric values as TRUE (1)

IMPORTANT! *Data types marked with '*' are specific to Model Composer, and are not naturally supported by Simulink. While Simulink does support fixed point data types, you must have the Fixed-Point Designer™ product installed and licensed. In addition, the fixed point data type supported by Xilinx Model Composer is not compatible with the fixed point data type supported by Simulink although it uses a similar notation.*

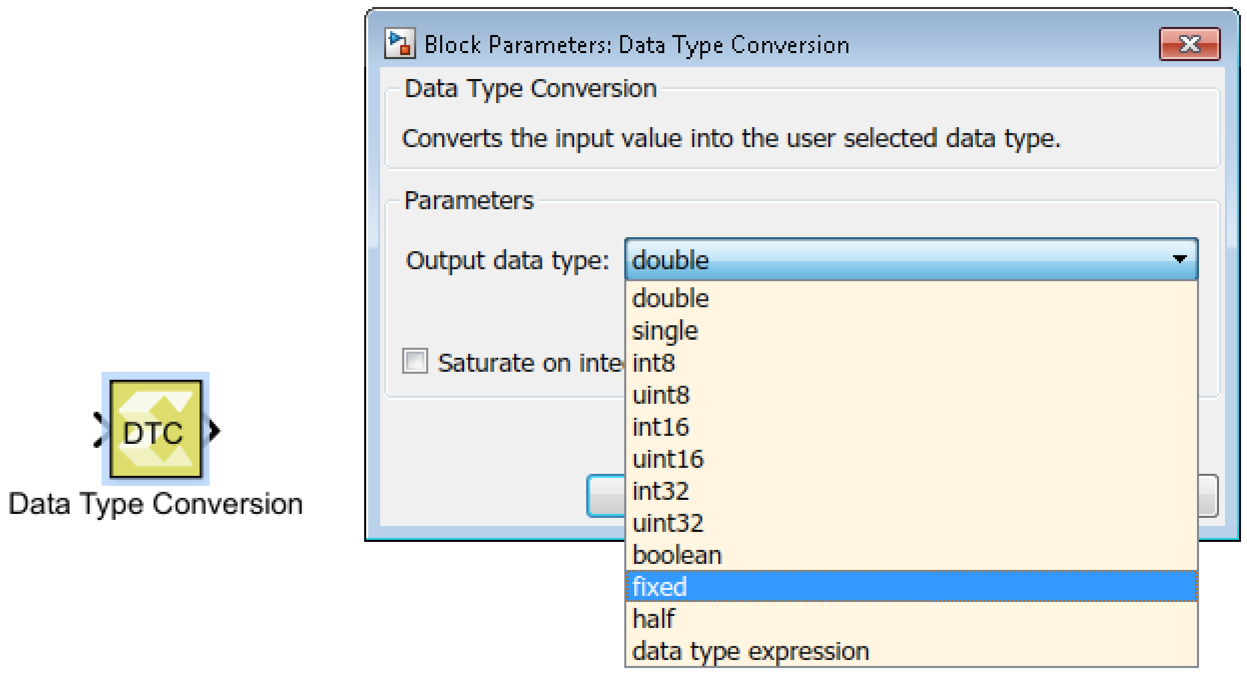
Notice in the preceding table there are some data types that are supported by Model Composer that are not supported by default in Simulink. If you connect blocks from the Model Composer block library, with `fixed` or `half` data types, to Simulink native blocks, you will see an error when running simulation in Simulink, or when using the **Update Diagram** command, or pressing `Ctrl+D`.

```
RelationalOperator does not accept signals of data type 'x_sfix16'.
'ConstRE_or_IMpartBug/Relational Operator' only accepts numeric and
enumerated data types.
```

This error indicates that Simulink could not cast the signal value from the Xilinx Model Composer fixed data type to a double precision floating point data type.

In cases of mismatched data types, Xilinx Model Composer recommends that you use a Data Type Conversion block to specify the behavior of the model, and indicate the conversion of one data type to another. The Data Type Conversion block (DTC) is found in the Xilinx Model Composer Library under the Signal Attributes library.

Figure 5: **Data Type Conversion Block**



Converts the input value into the user selected data type.

Output data type: double

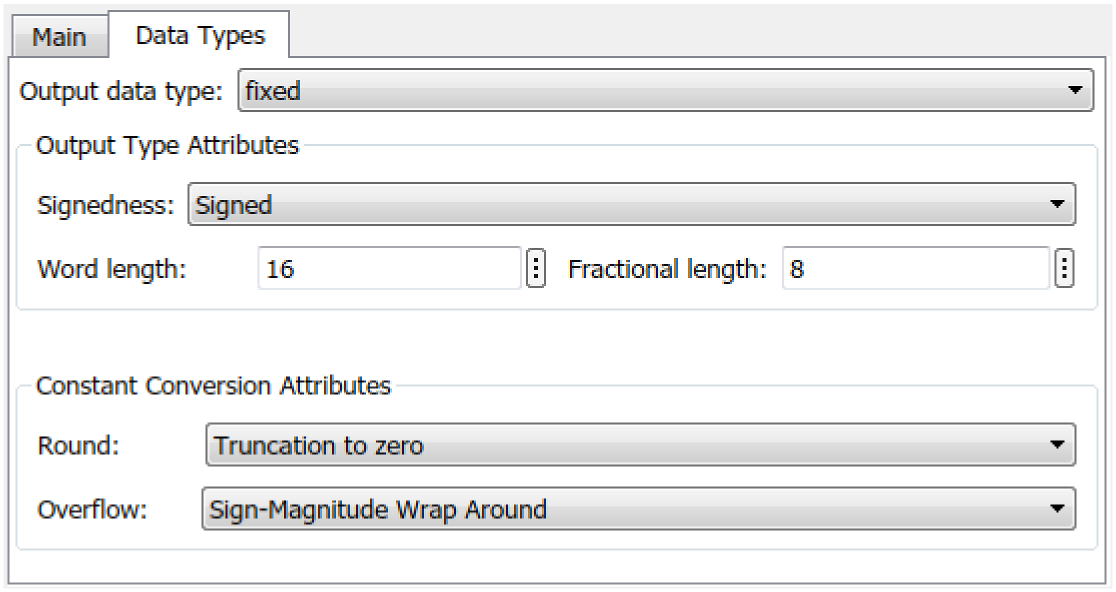
The DTC block lets you specify the **Output data type**, while the input data type is automatically determined by the signal connected to the input port. Using the DTC block, you can convert from single precision floating point to double precision for example, or from double precision to single precision.

IMPORTANT! *You must exercise caution when casting from a higher precision data type to a lower precision data type, as loss of precision can lead to rounding or truncation and loss of data can occur.*

Working with Fixed-Point Data Types

As indicated earlier, Simulink® provides support for fixed-point data types through the Fixed-Point Designer™ product. However, the format of the `fixed` data type supported by Xilinx® Model Composer and Simulink are not compatible.

Figure 6: **Fixed-Point Data Type**



Output data type: fixed

Output Type Attributes

Signedness: Signed

Word length: 16 Fractional length: 8

Constant Conversion Attributes

Round: Truncation to zero

Overflow: Sign-Magnitude Wrap Around

The format used to display the Xilinx Model Composer fixed-point data types is as follows: `x_[u/s]fix[wl]_E[n][fl]`

Where:

- `x_`: Is the prefix indicating the Xilinx fixed data type.
- `[u/s]`: Represents signed or unsigned data.
- `fix`: Indicates the fixed-point data type.
- `[wl]` Specifies the word length of the data.
- `E`: Prefix for the fractional portion of the fixed-point data type. Does not display if the fractional length is 0.
- `n`: Displays 'n' if the binary point is to the left of the right-most bit in the word; or displays no 'n' if the binary point is to the right of the right-most bit in the word.
- `[fl]`: Specifies the fractional length of the fixed-point data type, indicating the position of the binary point with respect to the right-most bit in the word.

For example, `x_sfix16_En6` represents a signed 16-bit fixed-point number, with 6-bits allocated to the right of the binary point.

Notice the fixed-point data type also lets you specify what happens in the case of data overflow, or the need to do rounding or truncation. For more information refer to Data Type Conversion.

You must use a DTC block to convert the Xilinx Model Composer fixed-point data type into the Simulink fixed-point data type. However, there is no direct conversion between Model Composer fixed-point and Simulink fixed-point data types, so you can use the following method:

1. Convert Xilinx Model Composer fixed-point data type to the `double` data type using the DTC block from the **Xilinx Model Composer** library in the Library Browser.
2. Convert the `double` data type to Simulink format fixed-point data type using the Simulink Data Type Conversion block from the Simulink **Signal Attributes** library in the Library Browser.
3. Match the signedness, word length, and fractional length between the two fixed-point data types.

TIP: *Converting between the Xilinx Model Composer fixed data type and the Simulink fixed data type is not recommended unless necessary for your design. You can convert from the Model Composer fixed-point data type to double, as shown in the first step, and this should be sufficient for most applications.*

Although handling fixed-point data type in a design is more time consuming, the value of using fixed-point data types for applications targeted at implementation in an FPGA is worth the challenge. It is widely accepted that designing in floating point leads to higher power usage for the design. This is true for FPGAs where floating-point DSP blocks have been hardened onto the FPGA and users must implement a floating point solution using DSP blocks and additional device resources. Floating-point implementations require more FPGA resources than an equivalent fixed-point solution. With this higher resource usage comes higher power consumption and ultimately increased overall cost of implementing the design. For more information refer to the white paper *Reduce Power and Cost by Converting from Floating Point to Fixed Point* (WP491).

Working with Half Data Types

Xilinx Model Composer also supports a `half` precision floating point data type, which is 16 bits wide instead of 32 bits, because it consumes less real estate on the target device when implemented on the FPGA. This is an important consideration when designing in Model Composer. However, Simulink does not support the `half` data type, and this can lead to errors in simulation. The solution is to use the Model Composer DTC block to convert the `half` into the `single` data type supported by Simulink, for those portions of the design that are not in the Model Composer sub-module and will not be part of the generated output.

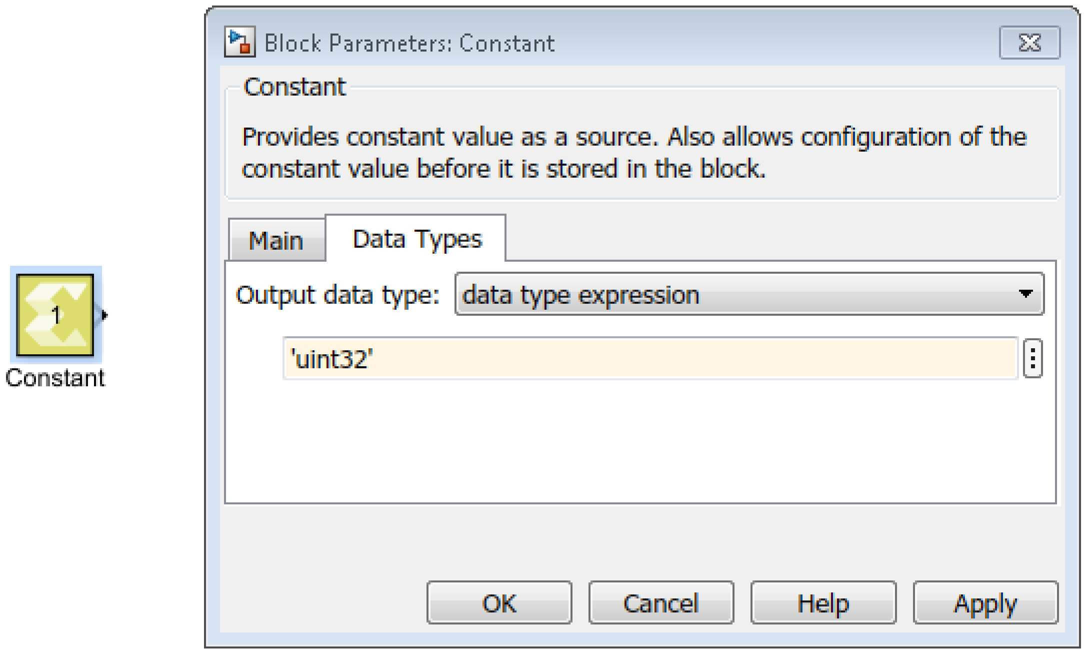
Working with Data Type Expression

Model Composer lets you specify data types as an expression. Currently the following Model Composer library blocks support data type expression:

- Constant
- Data Type Conversion
- Gain
- Look-Up Table
- Reinterpret

To specify a data type expression open one of the block types that support it, and edit the data type and value. The following shows a data type expression being defined for the Constant block. The **Output data type** is specified as an expression, and a string is specified to indicate the data type value, in this case 'uint32'.

Figure 7: **Data Type Expression**



Constant

Provides constant value as a source. Also allows configuration of the constant value before it is stored in the block.

Output data type: data type expression

'uint32'

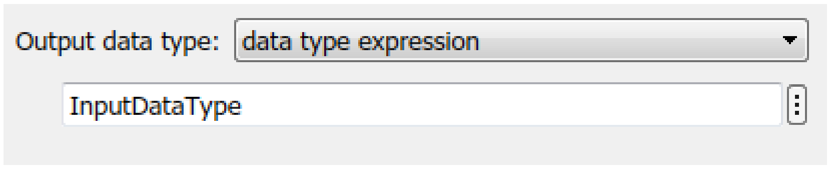
The data type value can be specified as a string representing any of the supported data types shown in the Model Composer Data Types table in the Working with Data Types section. The exception to this rule is for fixed point data types, which are not specified with the `fixed` string, but are defined according to the display format discussed under Working with Fixed-Point Data Types (e.g. 'x_sfix16_En8').

The real benefit of defining a data type as an expression is the ability to programmatically determine the data type value using a variable from the model. For instance, if you define a variable from the MATLAB® command line:

```
>> InputDataType = 'x_ufix8_En7';
```

You can use the variable in defining the data type expression.

Figure 8: **Variable Data Type**



Output data type: data type expression

InputDataType

You can specify variables from the MATLAB command line, or define variables within the model using the **Tools → Model Explorer** menu command, or simply pressing `Ctrl+H`. From the Model Explorer you can create, edit, and manage variables that the model or a block uses.

TIP: *You can also enable the **View → Property Inspector** command to display the variables for currently selected objects.*

Managing Overflow

Sometimes the data type definition for a block will not support the incoming data value on a signal. In these cases, an overflow condition can occur if the value on the signal is too large or too small to be represented by the data type of the block. The two types of overflow that can occur are wrap overflow, and saturation overflow:

- **Wrap on Overflow**: This is the default overflow mechanism, and causes the bit value to wrap at the overflow point. For instance, when an arithmetic operation such as multiplying two numbers produces a result larger than the maximum value for the data type, effectively causing a wrap around.
- **Saturate on Overflow**: This is an overflow mechanism in which all operations such as addition and multiplication are limited to a fixed range between the minimum and maximum values supported by the data type. In essence, the value will reach the maximum or minimum value and stop.

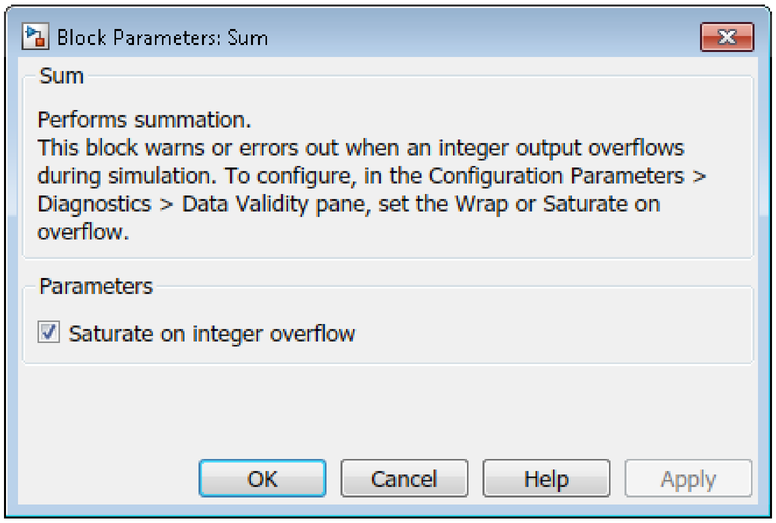
Wrapping is the default method for handling overflow, as it occurs naturally as the value overruns the data type. There is no checking required. However, the **Saturate on Overflow** option requires some additional logic in order to check the data value against the permitted maximum or minimum value to prevent wrapping. This additional logic consumes available resources on the target device.

Saturate on Overflow

Saturate on Integer Overflow

Model Composer currently supports overflow detection of integer data values on signals. As previously indicated, the default overflow mechanism is to wrap on overflow. Specific blocks in the standard Simulink library of blocks, and in the Xilinx Model Composer library, have an option to **Saturate on integer overflow**, which can be enabled on the Block Parameters dialog box. This parameter applies only if the output is an integer (int8, int16, int32, uint8, uint16, uint32). Refer to Appendix B: Model Composer Block Library for information specific to a block.

Figure 9: **Saturate on Integer Overflow**



Performs summation.
This block warns or errors out when an integer output overflows during simulation. To configure, in the Configuration Parameters > Diagnostics > Data Validity pane, set the Wrap or Saturate on overflow.

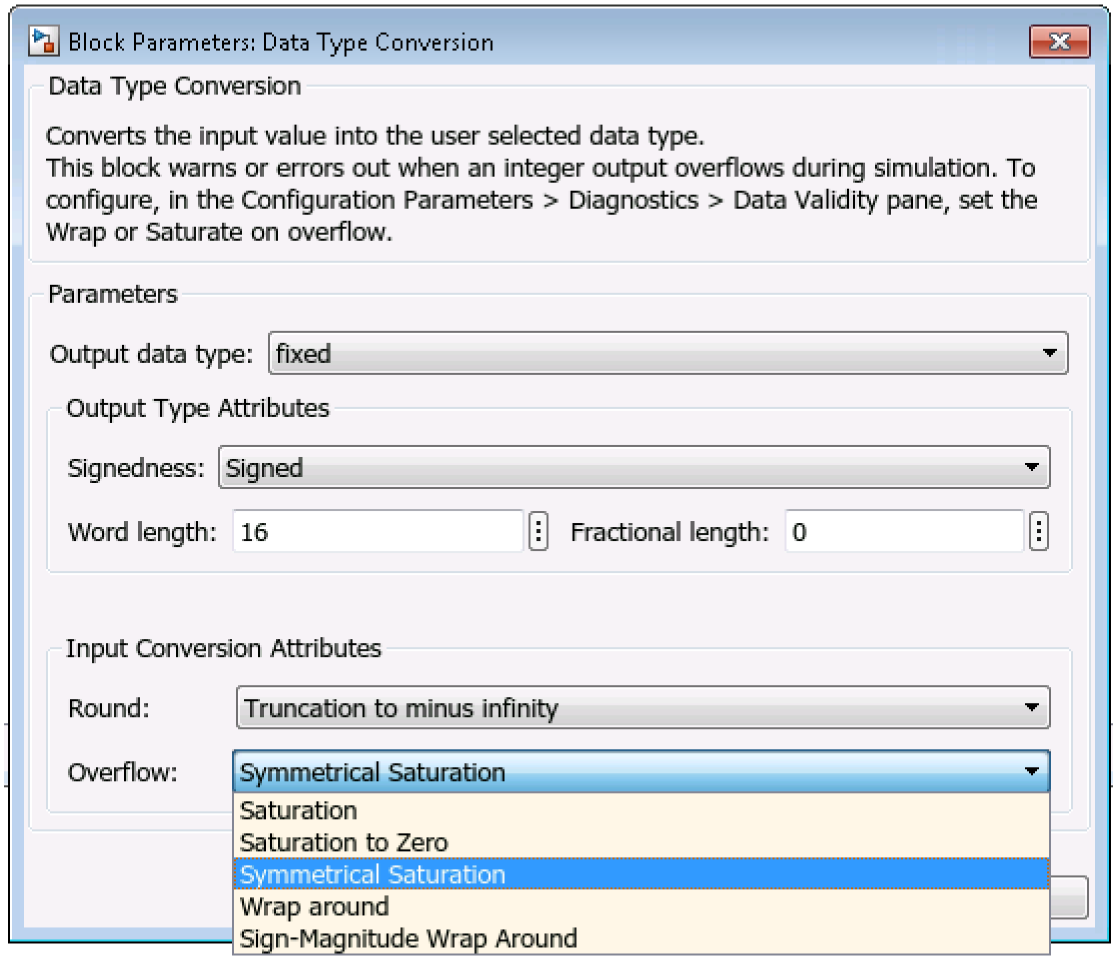
[✓] Saturate on integer overflow

Saturate on integer overflow simply means that when the input value exceeds the range of values supported by the output, either too great or too small, the value simply sits at the max or min supported value. The value is saturated, and does not change.

Saturate on Fixed-Point Overflow

For fixed-point data types, as supported on the Data Conversion Block (DTC) for example, the overflow modes offer more control than the Saturate on integer overflow option, as shown in the following figure.

Figure 10: **Fixed-Point Overflow**



Block Parameters: Data Type Conversion

Data Type Conversion

Converts the input value into the user selected data type.
This block warns or errors out when an integer output overflows during simulation. To configure, in the Configuration Parameters > Diagnostics > Data Validity pane, set the Wrap or Saturate on overflow.

Parameters

Output data type: fixed

Output Type Attributes

Signedness: Signed

Word length: 16

Fractional length: 0

Input Conversion Attributes

Round: Truncation to minus infinity

Overflow: Symmetrical Saturation

A description of the different fixed-point overflow modes is provided below, with a graph to illustrate the condition.

Table 3: **Fixed-Point Overflow Modes**

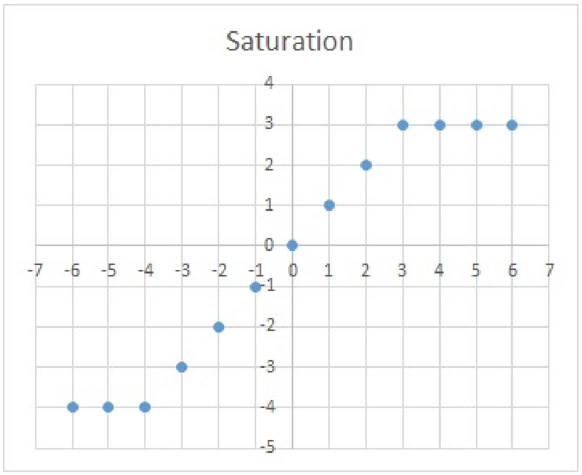
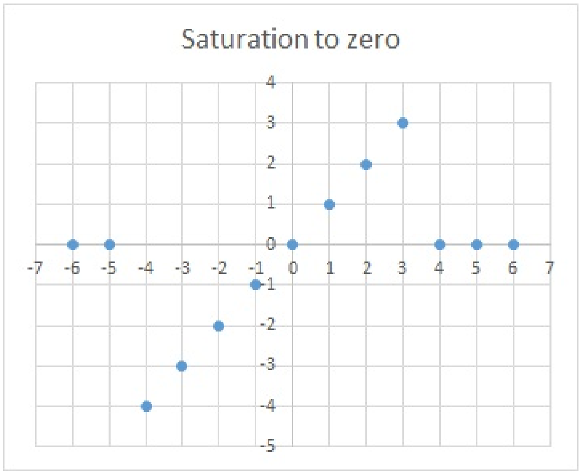
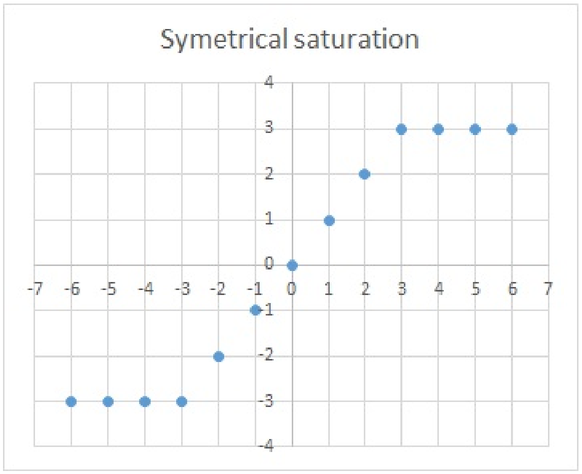
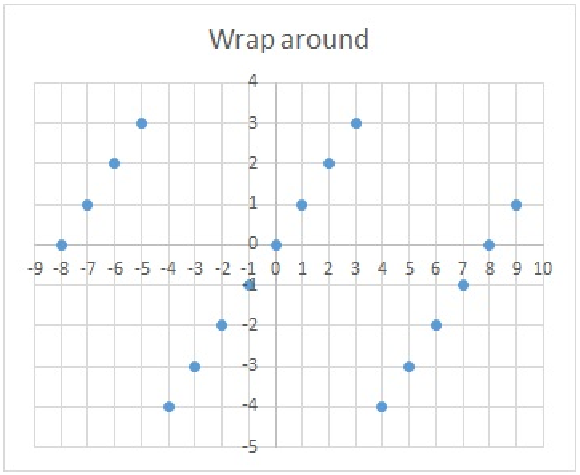
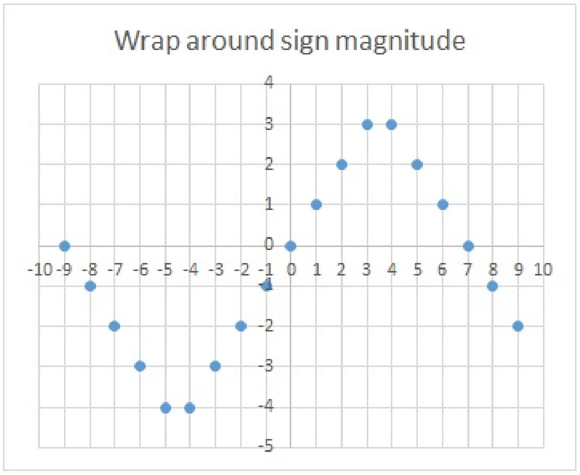
Mode	Description	Image
Saturation	When the input value overflows the output data type, the output value reaches saturation at the min or max value, and does not change.	
Saturation to Zero	When the input value overflows the output data type, the output value reaches saturation at the min or max value, and returns to zero.	
Symmetrical Saturation	Like Saturation to Zero, except the min and max values are symmetrical, or equal in size though opposite in value.	

Table 3: **Fixed-Point Overflow Modes** *(cont'd)*

Mode	Description	Image
Wrap around	When the input value exceeds the output data type, the output value is wrapped from the maximum value to the minimum value, or from the minimum value to the maximum value, thus cycling through the range of permitted values.	
Sign-Magnitude Wrap Around	When the input value exceeds the output data type, the output value reaches the maximum value, and then begins decreasing to return to the minimum value. In an underflow situation, the minimum value is reached, and begins increasing to return to the maximum value.	

Configuring Overflow Warnings

In case of either wrap or saturate, you may want to know when overflow occurs. You can define how Simulink handles each of these overflow conditions in the model by clicking on the **Model Configuration Parameters** command ( ) on the tool bar menu, or typing `Ctrl-E`. In the Configuration Parameters dialog box, under the **Diagnostics → Data Validity** tab, you can specify values for the **Wrap on Overflow** and **Saturate on Overflow** fields. Each of these fields can have one of the following settings:

- **none**: Simulink takes no special action to report or handle the overflow.
- **warning**: A message will be displayed in the diagnostic viewer. The next warning for the same block will be ignored, and simulation will continue.

- **error**: An error message will be displayed in the diagnostic viewer and the simulation will be terminated.

TIP: *Help for this dialog box can be found under Model Configuration Parameters: Data Validity Diagnostics.*

Creating a Top-Level Subsystem Module

In order to generate output from the Model Composer model the top-level of the Model Composer model must contain the Model Composer Hub block, as described in Adding the Model Composer Hub, as well as a subsystem that encapsulates the application design. To generate output from the subsystem that is instantiated at the top-level of the design, only Xilinx® Model Composer blocks and a limited set of specific Simulink® blocks can appear in the subsystem. The Xilinx Model Composer blocks define the functions to be compiled for the packaged IP or compiled C++ code. The top-level design can contain other blocks and subsystem modules that serve different purposes, such as simulation, but the primary application must be completely contained within the specified subsystem.

TIP: *The specific Simulink blocks that are supported in the Model Composer subsystem also appear in the Xilinx Model Composer block library. Refer to Supported Simulink Blocks for a complete list.*

To create a subsystem from within a model, add one or more blocks to the model canvas, select the blocks, and turn the selected blocks into a subsystem:

1. Drag and drop blocks onto the model canvas in the Simulink Editor, as explained in Adding Blocks to a Model.
2. Select one or more blocks, and right-click to use the **Create Subsystem from Selection** command.
3. Name the subsystem, giving it the same name you want to assign to the generated output application or IP.
4. Double-click the subsystem to open it in the Simulink Editor, and continue the design.

The Explorer bar and Model Browser in Simulink help you navigate your model:

- The Explorer bar lets you move up and down the hierarchy, or back and forth between different views in the Simulink Editor.
- The Model Browser provides a view of the **Model Hierarchy**, and lets you select and open different levels to quickly move through the hierarchy of the design.

Chapter 3

Importing C/C++ Code as Custom Blocks

Introduction

Model Composer lets you import C or C++ code to create new blocks that can be added to a library for use in models along side other Xilinx® Model Composer blocks. This feature lets you build custom block libraries for use in Model Composer.

TIP: *The import function example in the product showcases most of the capabilities of importing C/C++ code using a series of small examples. Type `xmcOpenExample('import_function')` in the MATLAB command window to open the example.*

Using the xmcImportFunction Command

Xilinx Model Composer provides the `xmcImportFunction` command, for use from the MATLAB command line, to let you specify functions defined in source and header files to import into Model Composer, and create Model Composer blocks, or block library. The `xmcImportFunction` command uses the following syntax:

```
xmcImportFunction('libName',{'funcNames'},'hdrFile',{'srcFiles'},
{'srchPaths'},'options')
```

Where:

- `libName`: A string that specifies the name of the Model Composer library that the new block is added to. The library can be new, and will be created, or can be an existing library.
- `funcNames`: Specifies a list (cell array) of one or more function names defined in the source or header files to import as a Model Composer block. An empty set, {}, imports all functions defined in the specified header file (`hdrFile`). For functions inside namespaces, full namespace prefix needs to be given. For example, to import the function `'sinf'` in `hls_math`, the full function name `'hls::sinf'` needs to be specified.

- `hdrFile`: A string that specifies a header file (`.h`) which contains the function declarations, or definitions. This should be the full path to the header file if it is not residing inside the current working directory. For example, to import a function from `hls_math.h`, you need to specify the full path `'$XILINX_VIVADO/include/hls_math.h'`.

IMPORTANT! *The function signature must be defined in the header file, and any Model Composer (XMC) pragmas must be specified as part of the function signature in the header file.*

- `srcFiles`: Specifies a list of one or more source files to search for the function definitions. When used in a model, the header and source files, together with the main header file of the model, `headerFile`, will be compiled into the shared library for simulation, and copied into the Target Directory specified for output generation in the Model Composer Hub block, as described in Adding the Model Composer Hub.
- `srchPaths`: Specifies a list of one or more search paths for header and source files. An empty set, {}, indicates no search path, in which case the code is looked for in the MATLAB current folder. Use `'$XILINX_VIVADO/include'` to include the HLS header files.

In addition, the `xmcImportFunction` command has the following `options`, which can follow the required arguments in any order:

- `'unlock'`: Unlock an existing library if it is locked. The `xmcImportFunction` command can add blocks to an existing library, but it must be unlocked in order to do so.
- `'override'`: Overwrite an existing block with the same name in the specified library.

TIP: *The help for `xmcImportFunction` can be accessed from the MATLAB command line, using `help xmcImportFunction`, and provides the preceding information.*

As an example, the following `simple.h` header file defines the `simple_add` function, with two double-precision floating point inputs and a pointer output. The function simply adds the two inputs and returns the sum as the output.

```
void simple_add(const double in1, const double in2, double *out) {
    *out = in1 + in2;
}
```

To import the `simple_add` function as a block in a Model Composer library you can enter the following command at the MATLAB command prompt:

```
xmcImportFunction('SimpleLib',{'simple_add'},'simple.h',{},{})
```

Where:

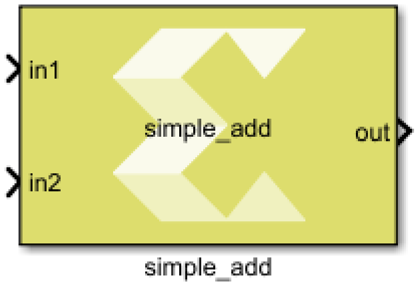
- `SimpleLib` is the name of the Model Composer library to add the block to.
- `simple_add` is the function name to import.
- `simple.h` is the header file to look in.

- No C source files or search paths are specified. In this case, the function definition must be found in the specified header file, and only the MATLAB current folder will be searched for the specified files.

TIP: *Model composer will give you a warning if you attempt to import a block with the same function name as a block that is already in the specified library.*

When `xmcImportFunction` completes, the `SimpleLib` library model will open with the `simple_add` block created as shown below.

Figure 11: **simple_add Block**



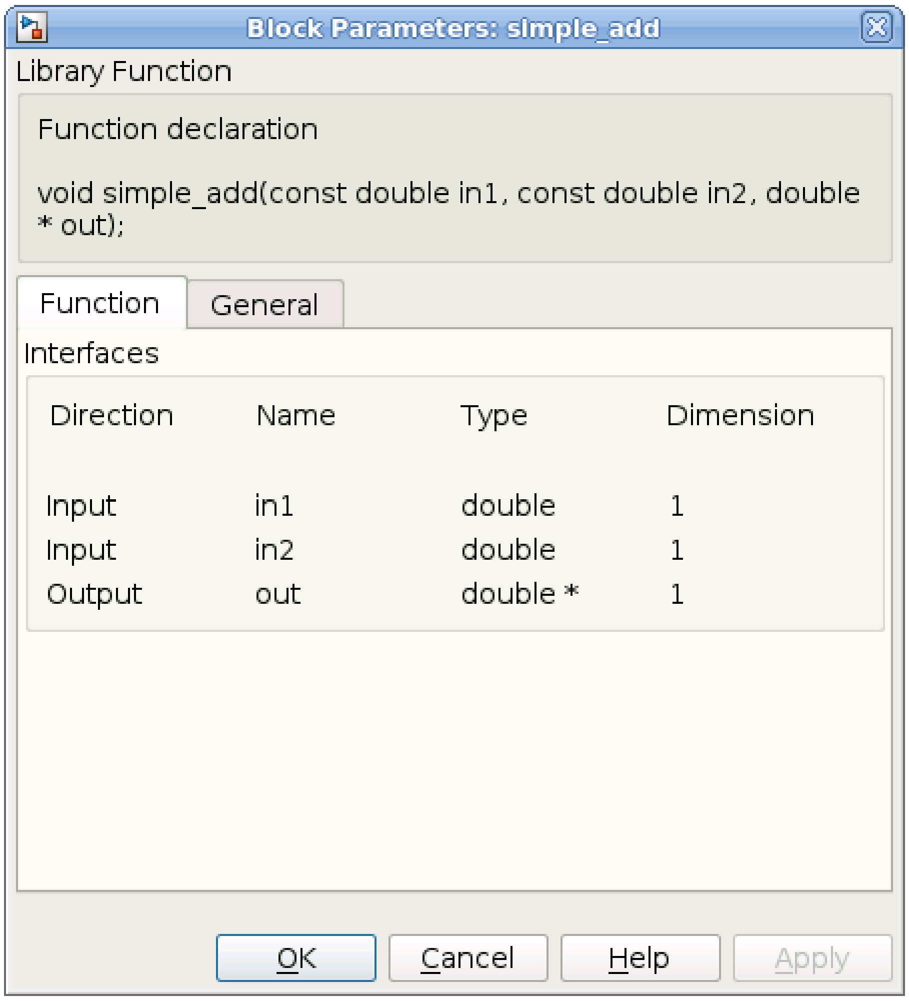
During simulation, the C/C++ code for the Library blocks will get compiled and a library file will get created and loaded. The library file will be cached to speed up initializing simulations on subsequent runs. You can change the source code underlying the library block without the need to re-import the block, or re-create the library, although the cached library file will be updated if you change the C/C++ source code.

However, if you change the function signature, or the parameters to the function, then you will need to rerun the `xmcImportFunction` command to recreate the block. In this case, you will also need to use the `override` option to overwrite the existing block definition in your library.

IMPORTANT! *You must rerun the `xmcImportFunction` command any time that you change the interface to an imported function, or the associated XMC pragmas. In this case, you should delete the block from your design and replace it with the newly generated block from the library. However, you can change the function content without the need to run `xmcImportFunction` again.*

After the block symbol has been created, you can double-click on the symbol to see the parameters of the imported block. You can quickly review the parameters as shown in the following figure to ensure the function ports have been properly defined.

Figure 12: **simple_add Block Parameters**



Function declaration

void simple_add(const double in1, const double in2, double * out);

Interfaces

Direction	Name	Type	Dimension
Input	in1	double	1
Input	in2	double	1
Output	out	double *	1

Importing C/C++ into Model Composer

While Model Composer lets you import C or C++ functions to create a library of blocks, it does have specific requirements for the code to be properly recognized and processed. The function source can be defined in either a header file (`.h`), or in a C or C++ source file (`.c`, `.cpp`), but the header file must include the function signature.

You can import functions with function arguments that are real or complex types of scalar, vectors, or matrices, as well as using all the data types supported by Model Composer, including fixed-point data types. Model Composer also lets you define functions as templates, with template variables defined by input signals, or as customization parameters to be specified when the block is added into the model or prior to simulating the model. Using function templates in your code lets you create a Model Composer block that supports different applications, and can increase the re-usability of your block library. Refer to Defining Blocks Using Function Templates for more information.

IMPORTANT! *The function signature must be defined in the header file, and any Model Composer (XMC) pragmas must be specified as part of the function signature in the header file.*

The `xmcImportFunction` command supports C/C++ function using `std::complex<T>` or `hls::x_complex<T>` types. For more details, see the explanation in Using Complex Types.

If the input of an imported function is a 1-D array, the tool can perform some automatic mappings between the input signal and the function argument. For example, if the function argument is `a[10]`, then the connected signal in Model Composer can be either a vector of size 10, or a row or column matrix of size 1x10 or 10x1.

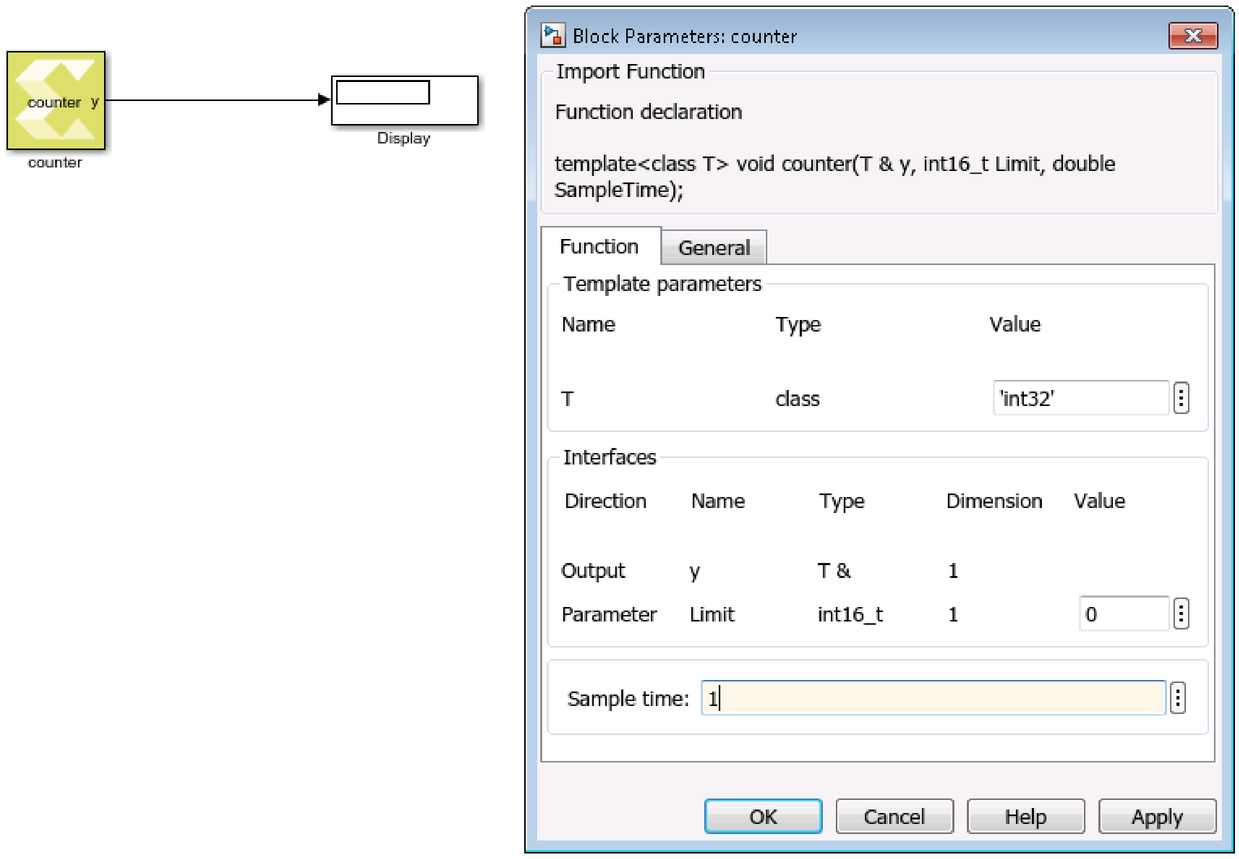
However, if all of the inputs and outputs of an imported function are scalar arguments, you can connect a vector signal, or a matrix signal to the input. In this case, the imported function processes each value of the vector, or matrix on the input signal as a separate value, and will combine those values into the vector, or matrix on the output signal. For example, a vector of size 10 connected to a scalar input, will have each element of the vector processed, and then returned to a vector of size 10 on the output signal.

You can import functions that do not have any inputs, and instead only generate outputs. This is known as a source block, and can have an output type of scalar, vector, complex, or matrix. You can also import source blocks with multiple outputs. The following example function has no input port, and `y` is the output:

```
#include <stdint.h>
#include <ap_fixed.h>

#pragma XMC OUTPORT y
#pragma XMC PARAMETER Limit
template <typename T>
void counter(T &y, int16_t Limit)
{
    static T count = 0;

    count++;

    if (count > Limit)
            count =0;
    y = count;
}
```

TIP: *Because source blocks have no inputs, the* `SampleTime` *parameter is automatically added when the block is created with* `xmcImportFunction` *command, as shown in the Function declaration in the following image. The default value is -1 whcih means the sample time is inherited from the model. You can also explicitly specify the sample time by customizing the block when it is added to a model, as shown below.*

Figure 13: **Setting Sample Time for a Source Block**



Function declaration

template<class T> void counter(T & y, int16_t Limit, double SampleTime);

Template parameters

Name	Type	Value
T	class	'int32'

Interfaces

Direction	Name	Type	Dimension	Value
Output	y	T &	1	
Parameter	Limit	int16_t	1	0

Sample time: 1

The direction of ports for the function arguments can be determined automatically by the `xmcImportFunction` command, or manually specified by pragma with the function signature in the header file.

- Automatically determining input and output ports:
  - The `return` value of the function is always defined as an output, unless the return value is `void`.
  - A formal function argument declared with the `const` qualifier is defined as an input.
  - An argument declared with a reference, a pointer type, or an array type without a `const` qualifier is defined as an output.
  - Other arguments are defined as inputs by default (e.g., scalar read-by-value).

- Manually defining input and output ports:
  - You can specify which function arguments are defined as inputs and outputs by adding the INPORT and OUTPORT pragmas into the header file immediately before the function declaration.
  - `#pragma XMC INPORT <parameter_name> [, <parameter_name>...]`
  - `#pragma XMC OUTPORT <parameter_name> [, <parameter_name>...]`

In the following example `in` is automatically defined as an input due to the presence of the `const` qualifier, and `out` is defined as an output. The imported block will also have a second output due to the integer `return` value of the function.

```
int func(const int in, int &out);
```

In the following function `in` is automatically defined as an input, and `out` as an output, however, there is no `return` value.

```
void func(const in[512], int out[512]);
```

In the following example the ports are manually identified using pragmas that have been added to the source code right before the function declaration. This is the only modification to the original C++ code needed to import the function into Model Composer. In this example the pragmas specify which parameter is the input to the block and which parameter is the output of the block.

```
#pragma XMC INPORT din
#pragma XMC OUTPORT dout
void fir_sym (ap_fixed<17,3,AP_TRN,AP_WRAP> din[100],
              ap_fixed<17,3,AP_TRN,AP_WRAP> dout[100]);
```

TIP: *`ap_fixed` specifies a fixed-point number compatible with Vivado HLS.*

Manually adding pragmas to the function signature in the header file to define the input and output parameters of the function is useful when your code does not use the `const` qualifier, and adding the `const` qualifier can require extensive editing of the source code when there is a hierarchy of functions. It also makes the designation of the inputs and outputs explicit in the code, which can make the relationship to the imported block more clear.

Some final things to consider when writing C or C++ code for importing into Model Composer:

- You should develop your source code to be portable between 32 bit and 64 bit architectures.
- Your source code can use Vivado HLS pragmas for resource and performance optimization, and Model Composer uses those pragmas but does not modify or add to them.

- If your code has static variables, the static variable will be shared across all instances of the blocks that import that function. If you do not want to share that variable across all instances you should copy and rename the function with the static variable and import a new library block using the `xmcImportFunction` command.
- If you use C (`.c`) source files to model the library function (as opposed to C++ (`.cpp`) source files), the `.h` header file must include an `extern "C"` declaration for the downstream tools (such as Vivado HLS) to work properly. An example of how to declare the `extern "C"` in the header files is as follows:

```
// c_function.h:
#ifdef __cplusplus
extern 'C' {
#endif
void c_function(int in, int &out);
#ifdef __cplusplus
}
#endif
```

Using Complex Types

C/C++ code for Vivado HLS can use `std::complex<T>` or `hls::x_complex<T>` to model complex signals for xmcImportFunction blocks. However, Vivado HLS does not support streaming `std::complex<T>` variables unless <T> is `ap_fixed<..>` or `ap_int<..>`. Large array variables that are non-streaming can use significant hardware resoources.

The code generated by Model Composer uses `hls::x_complex` for representing complex signals. If your imported C/C++ block function is modeled with `std::complex`, when generating the output code Model Composer will automatically insert `hls::x_complex`-to-`std::complex` adapters to convert the complex types for the block ports.

Importing functions that use `std::complex` is still supported. However, if the imported function has non-scalar argument of data type `std::complex<T>` a warning message will be issued during code generation that indicates that rewriting the function using `hls::x_complex` will improve the quality of results and allows streaming of complex variables.

The C/C++ code must also include the required header file for the complex type declaration. For the `hls::x_complex` type, the `xmcImportFunction` command must also include `‘$XILINX_VIVADO/include’` search path for the `hls_x_complex.h` header file. For example, the following imports the `complex_mult` function, and specifies the needed include path:

```
xmcImportFunction('my_lib',{'complex_mult'}, 'complex_mult.h', {},
{'$XILINX_VIVADO/include'});
```

Example Functions Using Complex Types

```
#include "hls_x_complex.h"
hls::x_complex<double>
complex_mult(hls::x_complex<double> in1, hls::x_complex<double> in2)
{ return in1 * in2; }

#include <complex>
std::complex<double>
complex_mult2(std::complex<double> in1, std::complex<double> in2)
{ return in1 * in2; }

#include <complex>
void
complex_mult3(std::complex<double> in1, std::complex<double> in2,
std::complex<double> &out1)
{ out1.real(in1.real() * in2.real() - in1.imag() * in2.imag());
   out1.imag(in1.real() * in2.imag() + in1.imag() * in2.real()); }
```

Defining Blocks Using Function Templates

IMPORTANT! *To use template syntax, your function signature and definition should both be specified in a header file when running `xmcImportFunction`.*

While it is common to write functions that accept only a predetermined data type, such as `int32`, in some cases you may want to create a block that accepts inputs of different sizes, or supports different data types, or create a block that accepts signals with different fixed-point lengths and fractional lengths. To do this you can use a function template that lets you create a block that accepts a variable signal size, data type, or data dimensions.

You can define blocks using function templates, as shown in the following example:

```
#include <stdint.h>
template <int ROWS, int COLS>
void simple_matrix_add(const int16_t in1[ROWS][COLS],
                       const int16_t in2[ROWS][COLS],
                       int16_t out[ROWS][COLS]) {
   for (int i = 0; i<ROWS; i++) {
      for (int j = 0; j<COLS; j++) {
         out[i][j] = in1[i][j] + in2[i][j];
      }
   }
}
```

The example uses the template parameters `ROWS` and `COLS`. The actual dimensions of the input and output arrays, `in1[ROWS][COLS]` for instance, are determined at simulation time by the dimensions of the input signals to the block. `ROWS` and `COLS` are template parameters used to define the dimensions of the function arguments, and also used in the body of the function, `i<ROWS` for example.

Use the command below to import the function into Model Composer:

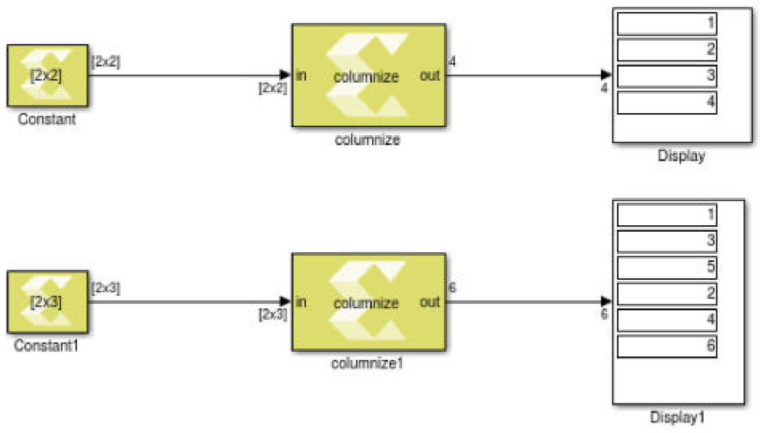
```
xmcImportFunction('SimpleLib',{'simple_matrix_add'},...
'template_example.h',{},{},'unlock')
```

TIP: *In the example above the ellipsis (...) is used to indicate a continuation of the command on the next line. Refer to Continue Long Statements on Multiple Lines in the MATLAB documentation for more information.*

You can perform simple arithmetic operations using template parameters. For example, the following code multiplies the ROWS and COLS of the input matrix to define the output, as shown in the figure below.

```
#include <stdint.h>
#pragma XMC INPORT in
#pragma XMC OUTPORT out
template<int ROWS,int COLS>
void columnize(const int16_t in[ROWS][COLS], int16_t out[ROWS*COLS]) {
   for (int i = 0; i<ROWS; i++) {
      for (int j = 0; j<COLS; j++) {
         out[i*COLS+j] = in[i][j];
      }
   }
}
```

Figure 14: **Columnize Function**



Other simple supported operations include +, -, *, /, %, <<, and >>, using both the template parameters and integer constants. For example:

```
template<int M, int N>
void func(const int in[M][N], int out[M*2][M*N]);

template<int ROWS, int COLS>
void func(array[2 * (ROWS + 1) + COLS + 3]);
```

You can also define a function template that uses a fixed-point data type of variable word length and integer length using function templates, as shown in the following example:

```
#include <stdint.h>
#include <ap_fixed.h>
#pragma XMC OUTPORT out
template <int WordLen, int IntLen>
void fixed_add(const ap_fixed<WordLen,IntLen> in1,
                      const ap_fixed<WordLen,IntLen> in2,
                      ap_fixed<WordLen+1,IntLen> &out) {
   out = in1+in2;
}
```

The example above uses the fixed point notations from Vivado HLS, which specifies the word length and the integer length. In Model Composer, as described in Working with Data Types, you specify the word length and the fractional length. This requires you to use some care in connecting fixed point data types in Model Composer to the imported `fixed_add` block. For example, in the function above if `WordLen` is 16 and `IntLen` is 11, in Model Composer fixed point data type the word length is 16, and the fractional length is 5. For more information on fixed-point notation in Vivado HLS, refer to the *Vivado Design Suite User Guide: High-Level Synthesis* (UG902).

TIP: *As shown in the example above, simple arithmetic operations are also supported in the fixed point template parameter.*

To import the `fixed_add` function and create a block in Model Composer, use the following command:

```
xmcImportFunction('SimpleLib',{'fixed_add'},fixed_example.h',{},...
{'$XILINX_VIVADO/include'})
```

Function Templates for Data Types

Function templates for data types are functions that can operate with generic data types. This lets you create library functions that can be adapted to support multiple data types without needing to replicate the code or block in the Model Composer block library to support each type. The `xmcImportFunction` command in Model Composer will create generic library blocks which the user of the block can connect to signals of any data types supported by the block.

The data type (`typename`) template parameters are resolved at simulation run time, when the code and simulation wrapper are generated. The parameters are replaced during simulation by the actual data types that are specified by the signals connecting to the library block. The resolved data types can only be the types that Model Composer supports as discussed in Working with Data Types.

To import a block that accepts multiple data types, you use a function template. For example:

```
template <typename T>
T max(T x, T y) {
   return (x > y) ? x : y;
}
```

Or, in the case of a function with complex function arguments, the example would appear as as follows:

```
#include <complex>
template <typename T>
void mult_by_two(std::complex< T > x, std::complex< T > *y)
{
   *Out = In1 * 2;
}
```

The determination of the type is made by Model Composer during simulation. The `typename` (or `class`) parameters are propagated from input signals on the block, or are customization parameters that must be defined by the user at simulation run time.

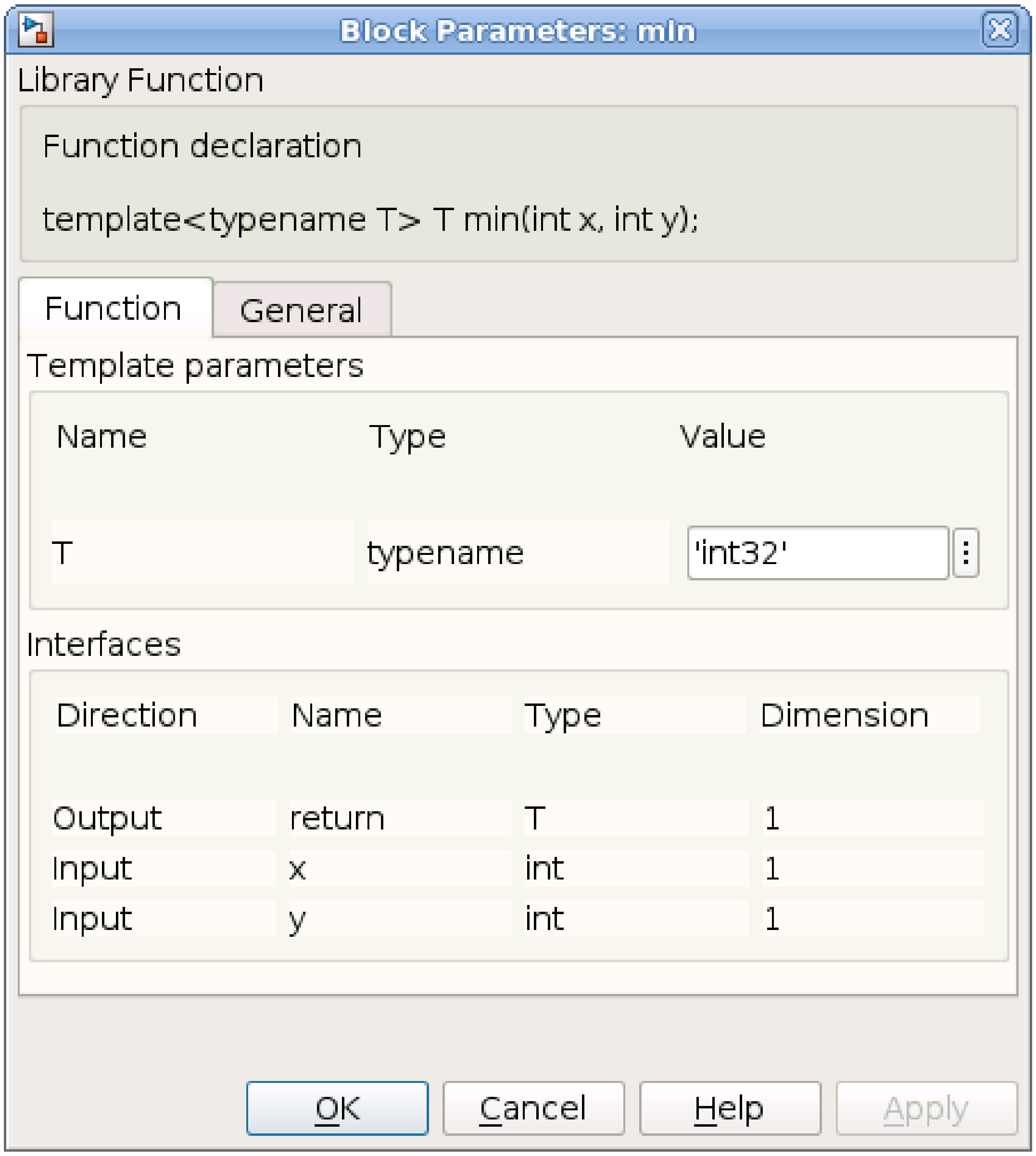
IMPORTANT! *The data type for a function or class cannot be propagated from an output.*

For example, the following function template specifies the parameter 'T' as a customization parameter. Because it is not associated with either input argument, 'x' or 'y', it must be specified by the user when the block is added to the model:

```
template <typename T>
T min(int x, int y) {
   return (x < y) ? x : y;
}
```

The Block Parameters dialog box for the generated Library Function block has an edit field to enter the template argument as shown in the following figure.

Figure 15: **Library Function Block Parameters**



Function declaration

template<typename T> T min(int x, int y);

Template parameters

Name	Type	Value
T	typename	'int32'

Interfaces

Direction	Name	Type	Dimension
Output	return	T	1
Input	x	int	1
Input	y	int	1

In the template syntax, the data type template parameters for function or class can be specified with other template parameters. The order of specification is not important. For example:

```
template <typename T1, int ROWS, int COLS, int W, int I>
T1 func(T1 x[ROW][COLS], ap_fixed<W, I> &y) {
...
}
```

IMPORTANT! *In the example above, notice that the 'T1' template parameter is used to specify both the function return and the data type of the input 'x'. In this case, because it is a single template parameter, both arguments will resolve to the same data type that is propagated from the input signal to the block.*

SUPPORTED_TYPES/UNSUPPORTED_TYPES Pragma

When defining a data type (`typename`) template parameter (or `class`), you can also define the accepted data types for the variable by using either the `SUPPORTED_TYPES` or `UNSUPPORTED_TYPES` pragma as part of the function signature. This is shown in the following code example.

```
#pragma XMC INPORT x
#pragma XMC INPORT y
#pragma XMC SUPPORTED_TYPES T: int8, int16, int32, double, single, half
template <class T>
T max(T x, T y) {
   return (x > y) ? x : y;
}

#pragma XMC UNSUPPORTED_TYPES T: boolean
#pragma XMC INPORT x, y
template <typename T>
T min(T x, T y) {
   return (x < y) ? x : y;
}
```

Model Composer supports an extensive list of data types as discussed in Working with Data Types. To specify which of these data types the template parameter supports, you can either include the list of supported types, or the unsupported types. The `SUPPORTED_TYPES` and `UNSUPPORTED_TYPES` pragmas are simply two opposite views of the same thing:

- `SUPPORTED_TYPES`: Specifies a template parameter name (`param`), and the list of data types that are accepted by that parameter. This implies the exclusion of all types not listed.

  ```
  #pragma XMC SUPPORTED_TYPES param: type1, type2, ...
  ```

- `UNSUPPORTED_TYPES`: Specifies a template parameter name (`param`), and the list of data types that are not accepted by that parameter. This implies the inclusion of all types not listed.

  ```
  #pragma XMC UNSUPPORTED_TYPES param: type1, type2, ...
  ```

With the `SUPPORTED_TYPES` or `UNSUPPORTED_TYPES` pragma in place, Model Composer will check the type of input signal connected to the block to ensure that the data type is supported. Without the use of one of these pragmas, the data type template parameter will accept any of the data types supported by Model Composer.

Function Template Specialization and Overloading

Specialization is supported for function templates in the `xmcImportFunction` command. Model Composer will create the library block for the generic function template, supporting multiple data types, but the block will also include any specialized functions to be used when connected to input signals with matching data types. Both the generic function, and any specialization functions are compiled into the block DLL. For example:

```
template <typename T>
T min(T x, T y) {
   return (x < y) ? x : y;
}

template <>
bool min<bool>(bool x, bool y) {
...
}
```

In this case, Model Composer will call the specialized boolean form of the `min` function when the block is connected to boolean signals.

Overloading of a function with the same number of input/output arguments is also supported by Model Composer. For example, the following defines two forms of the function:

```
int func(int x);
float func(float x);
```

You can also overload a function template as shown below:

```
template <typename T>
int func(int x, T y);

template <typename T>
float func(float x, T y);
```

TIP: *Overloading functions with different numbers of input/output arguments or different argument dimensions is not supported, and must be defined as separate functions.*

Defining Customization Parameters

Template parameters can be used to define the port array sizing and data type, and the parameters are defined by the input signals on the block. Additional customization parameters can also be defined, which are not defined by input signals, and thus must be defined by the user using the Block Parameter dialog box sometime before simulation run time.

There are two methods to define customization parameters for a library function block:

- Using C/C++ function templates.

- Assigning the Model Composer (XMC) PARAMETER pragma to a function argument, defining it as not connecting to an input or output of the block, but rather as a customization parameter.

There are pros and cons to both methods, which are discussed below.

Function Templates

The first method, defines a function template that uses template parameters for customization. A template parameter defines a customization parameter in Model Composer if its value is not determined by an input signal, or derived by the function as an output. You can use customization parameters to define the values, data types, or data dimensions of output ports, or for parameters in use in the function body.

IMPORTANT! *The template function signature and function definition must be defined in the C/C++ header file.*

The template parameter for the function argument is defined using standard function template syntax, but the template parameter is not assigned to an input argument in the function signature. When the block is instantiated into a model, Model Composer identifies template parameters whose values are not determined by input signals, and lets the user define the values for those customization parameters. Values can be defined for customization parameters in the model any time prior to simulation.

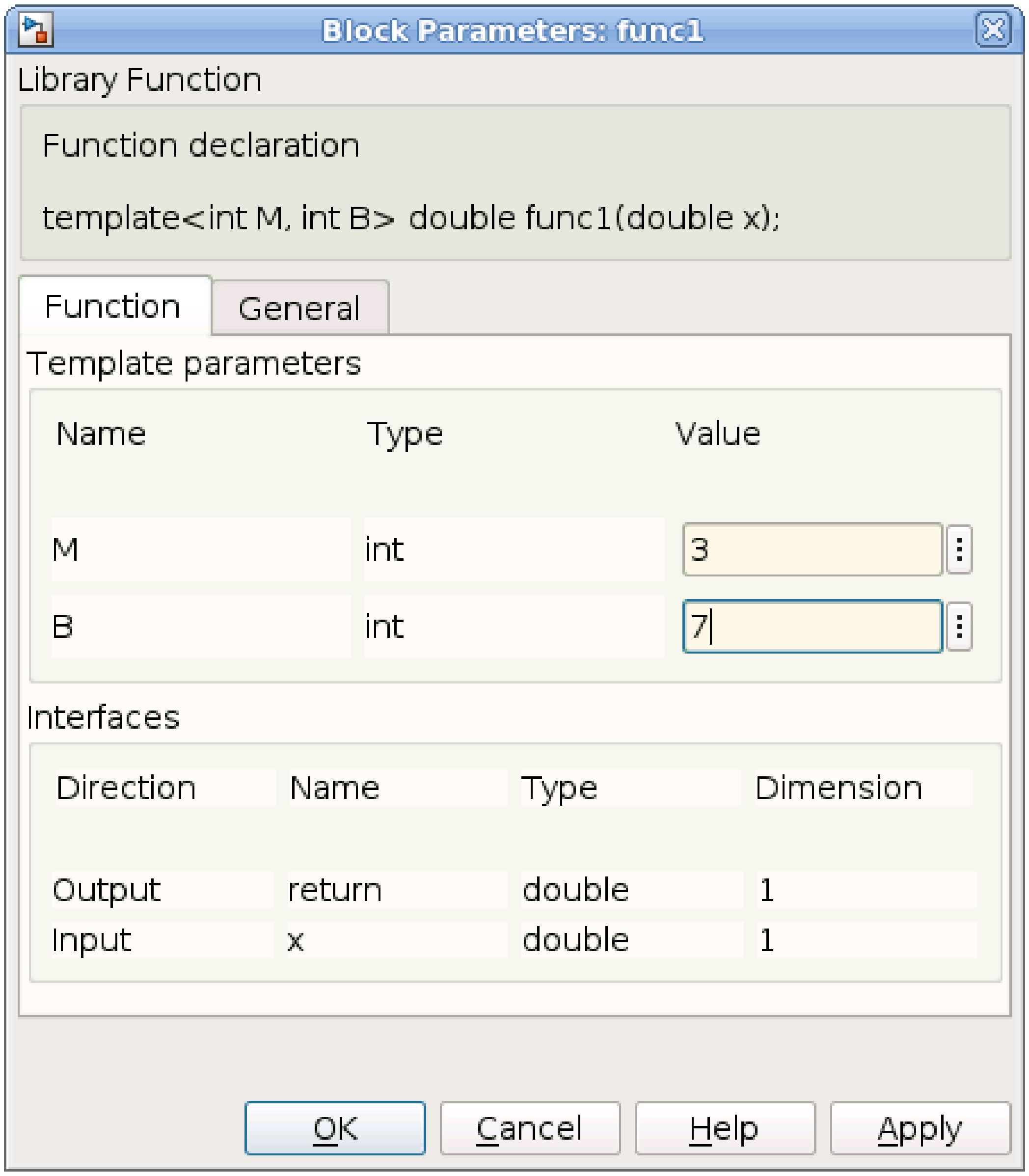
For function templates, the customization parameters can only be integer values to define the size or dimensions of a data type, or can only be scalar variables with definable data types. Model Composer defines a default value of 0 for integer parameters, and 'int32' for data type, or typename parameters.

In the function template example below, the template parameters ‘M’ and 'B' define customization parameters because the parameter values are not inherited from the input signal to the block. In this case, the parameters need to be customized by the user when the block is added to the model, or any time before simulation.

```
template <int M, int B>
double func1(double x) {
   return x * M + B;
}
```

Customization parameters are displayed in the Block Parameters dialog box for the imported block as shown for the `func1` function below. Double click on a block in the model to open the **Block Parameters** dialog box, then enter the value for any editable parameters, such as 'M' and 'B' below.

Figure 16: **Entering Parameter Values**



Block Parameters: func1

Library Function

Function declaration

template<int M, int B> double func1(double x);

Function | General

Template parameters

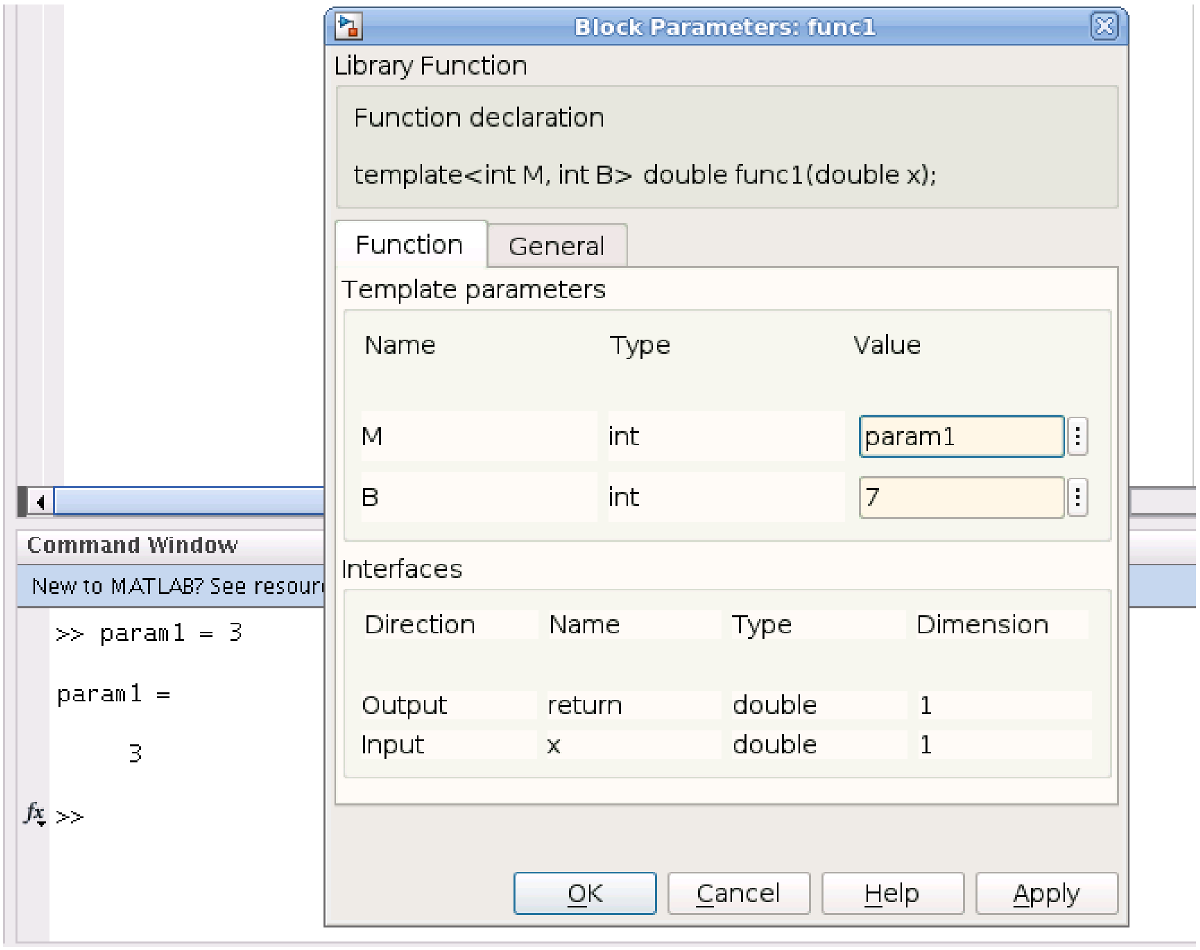
Name	Type	Value
M	int	3
B	int	7

Interfaces

Direction	Name	Type	Dimension
Output	return	double	1
Input	x	double	1

Optionally, the user can also specify the name of a MATLAB workspace variable in the text field for the customization parameter, and have the value determined by Model Composer through the MATLAB variable. For example, the variable `param1` is defined in the MATLAB workspace, and used to define the value for 'M'.

Figure 17: **Defining Parameters using Workspace Variables**



Block Parameters: func1

Library Function

Function declaration

template<int M, int B> double func1(double x);

Function | General

Template parameters

Name	Type	Value
M	int	param1
B	int	7

Interfaces

Direction	Name	Type	Dimension
Output	return	double	1
Input	x	double	1

OK Cancel Help Apply

Command Window

New to MATLAB? See resour

```
>> param1 = 3

param1 =

     3

>>
```

PARAMETER Pragma

The second method defines function arguments as customization parameters through the use of the Model Composer `PARAMETER` pragma.

To declare that a function argument is a customization parameter, you must add the `PARAMETER` pragma with the parameter name, or list of names, before the function signature in the header file. You can specify multiple parameters with one pragma, or have separate pragmas for each, as shown below.

```
#pragma XMC PARAMETER <name1>, <name2>
#pragma XMC PARAMETER <name3>
function declaration(<name1>, <name2>, <name3>)
```

When a function argument is declared a customization parameter by pragma, the `xmcImportFunction` command will not create an input or output port on the block for that argument. It will be defined for use inside the function body only. When the block is added to a model, a customization field is added to the Block Parameter dialog box, and the user of the block can define values for the customization parameters.

Using the `PARAMETER` pragma on a function argument that is already driven by the input signal will be flagged as an error or a warning. In this case, the signal input propagation through the function will have higher precedence than the customization parameter.

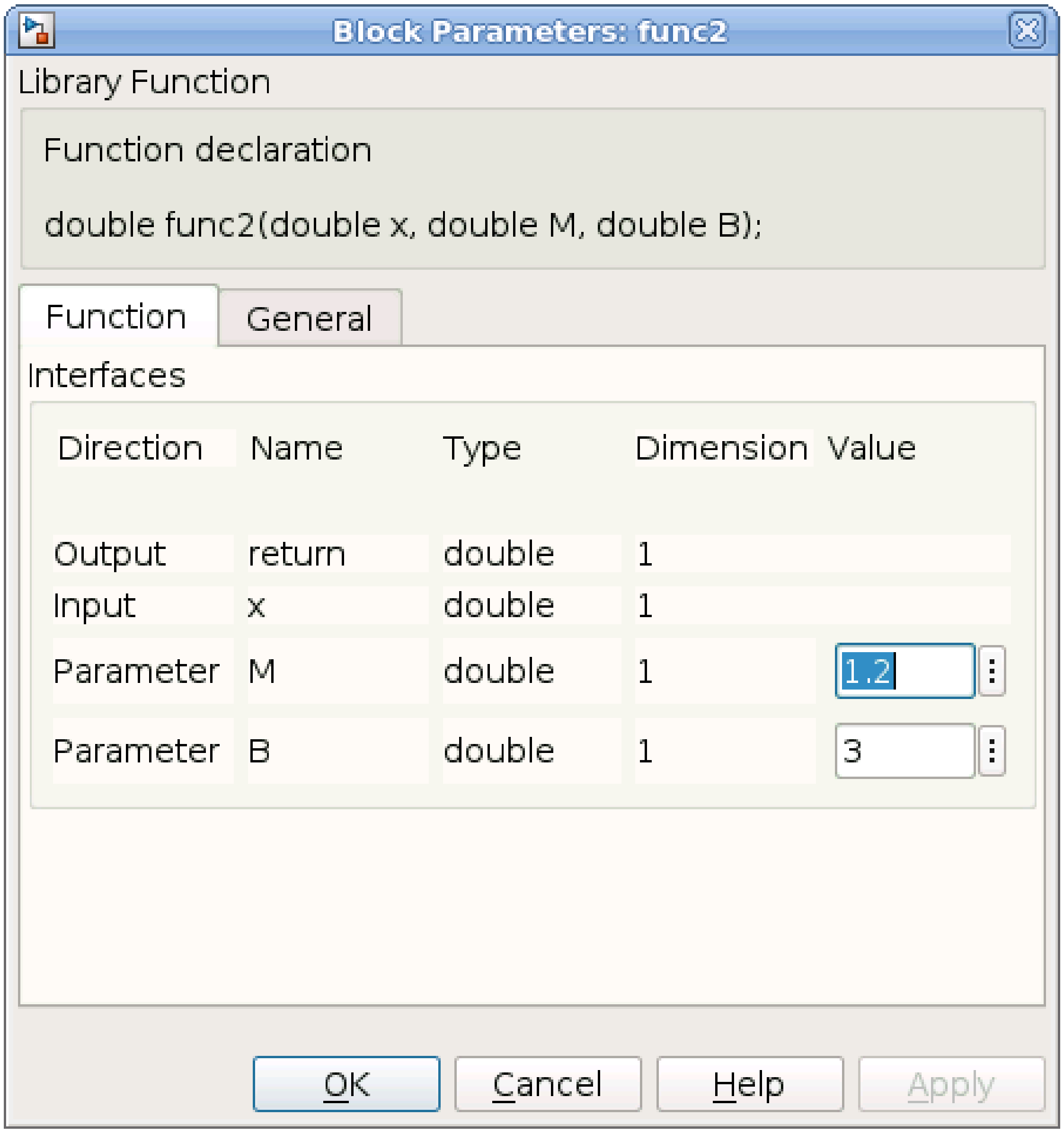
While the function templates method only supports scalar and integer type customization parameters, the `PARAMETER` pragma supports integer, floating point or fixed point data type for the parameters. The customization parameters also can be scalar, vector or a two-dimensional matrix. In addition, while the function template defines default values of 0 for integer types, and `int32` for the data type, the `PARAMETER` pragma lets you define default value for the parameters. Model Composers defines default values of 0 for all parameters that do not have user-defined defaults.

The example below uses the Model Composer `PARAMETER` pragma to define the customization parameters 'M' and 'B'.

```
#pragma XMC PARAMETER M, B
double func2(double x, double M = 1.2, double B = 3) {
return x * M + B;
}
```

The 'M' and 'B' customization parameters also have default values assigned: `M=1.2`, `B=3`. The default values for the customization parameters are assigned to the arguments in the function signature, and are displayed in the **Block Parameters** dialog box when opened for edit, as shown below.

Figure 18: **Customization Parameters with Defaults**



Function declaration

double func2(double x, double M, double B);

Direction	Name	Type	Dimension	Value
Output	return	double	1	
Input	x	double	1	
Parameter	M	double	1	1.2
Parameter	B	double	1	3

IMPORTANT! *If you define default values for the customization parameters of any argument, the C/C++ language requires that all arguments following that one must also have default values assigned, because the function can be called without arguments having default values. Therefore, you should add all customization parameters with default values at the end of the function argument list.*

Vector and Matrix Customization Parameters

The PARAMETER pragma method can also be used to specify customization parameters with vector and matrix dimensions, or values. In the following example the `coef` vector is defined by the pragma as a customization parameter:

```
#pragma XMC PARAMETER coef
#pragma XMC INPORT din
#pragma XMC OUTPORT dout
#pragma XMC SUPPORTS_STREAMING
void FIR(ap_fixed<17, 3> din[100], ap_fixed<17, 3> dout[100],
ap_fixed<16, 2> coef[52]);
```

The constant array values of the customization parameter are entered in MATLAB expression format. Note that commas are optional:

- Vector parameter: `[val1, val2, val3, ...]`
- Matrix parameter (row-major order): `[val11, val12, val13, ...; val21, val22, val23, ...; ...]`

Interface Output Types and Sizes

Customization parameters can also be used to directly set the data types and dimension size for output ports whose values are not determined by inputs to the function. In the function below, the template variables define the word length and fractional length of the `ap_fixed` data type and the array size.

```
template <typename T1, int N1, int W2, int I2, int N2>
void func(const T1 in[N1], ap_fixed<W2, I2> out[N2]) {
...
}
```

The template variables 'W2, 'I2'' and 'N2' define customization parameters because the values must be set by the user rather than determined from the input arguments. However, Model Composer recognizes that the template variables 'T1' and 'N1' are specified on the input port, and so the data type (`typename`) and the size of the input vector are not customization parameters, but rather get defined by the input signal on the block.

To set the data type for output ports, or arguments used in the body of the function, the `typename` specified must be one of the Model Composer supported data types, including the signed or unsigned fixed data types.

Table 4: **Model Composer Supported Data Types**

Supported Typenames
'int8'
'uint8'
'int16'

Table 4: **Model Composer Supported Data Types** *(cont'd)*

Supported Typenames
'uint16'
'int32'
'uint32'
'double'
'single'
'x_half'
'boolean'
'x_sfix<n1>_En<n2>'
'x_ufix<n1>_En<n2>'

In the example function below, while the `typename` for 'T1' is determined by the input signal, you can set the `typename` for 'T2' in the **Block Parameters** dialog box on the mask, when the block is added to a model, or before simulation run time:

```
template <typename T1, int N1, typename T2, int N2>
void func(const T1 in[N1], T2 out[N2]) {
...
}
```

Pragmas for xmcImportFunction

XMC SUPPORTS_STREAMING

The Model Composer `SUPPORTS_STREAMING` pragma indicates that the array parameters of a function support streaming data. This means that each element of the array is accessed once in a strict sequential order and indicates to Model Composer to optimize the design for streaming data. There can be no random access to the non-scalar arguments of a function to which the `SUPPORTS_STREAMING` pragma is applied.

The following example illustrates the difference between random access and sequential access. The `transform_matrix` output array of the `create_transform_matrix` function is addressed in a random order. It accesses the last row of the `transform_matrix` first, followed by the first and second row. This prevents the block from supporting streaming data.

```
void create_transform_matrix(const float angle, const float center_x,
                             const float center_y, float transform_matrix[3]
[3]) {
    float a = hls::cosf(angle);
    float b = hls::sinf(angle);

    transform_matrix[2][0] = 0;
    transform_matrix[2][1] = 0;
    transform_matrix[2][2] = 0;

    transform_matrix[0][0] = a;
    transform_matrix[0][1] = b;
    transform_matrix[0][2] = (1-a)*center_x-b*center_y;

    transform_matrix[1][0] = -b;
    transform_matrix[1][1] = a;
    transform_matrix[1][2] = b*center_x +(1-a)*center_y;
}
```

To change this function to support streaming data, it can be modified as depicted below to address the `transform_matrix` output array in a sequential manner:

```
#pragma XMC SUPPORTS_STREAMING
void create_transform_matrix(const float angle, const float center_x,
                             const float center_y, float transform_matrix[3]
[3]) {
    float a = hls::cosf(angle);
    float b = hls::sinf(angle);

    transform_matrix[0][0] = a;
    transform_matrix[0][1] = b;
    transform_matrix[0][2] = (1-a)*center_x-b*center_y;

    transform_matrix[1][0] = -b;
    transform_matrix[1][1] = a;
    transform_matrix[1][2] = b*center_x +(1-a)*center_y;

    transform_matrix[2][0] = 0;
    transform_matrix[2][1] = 0;
    transform_matrix[2][2] = 0;
}
```

As shown in the preceding example, to specify that an imported function supports streaming, simply add the `SUPPORTS_STREAMING` pragma in the C or C++ header file before the function declaration:

```
#pragma XMC SUPPORTS_STREAMING
```

If the function has array arguments that are accessed sequentially, but `SUPPORTS_STREAMING` is not specified, then a subsystem using that block will not be implemented in a streaming architecture. This means the performance of the function will not be optimized.

IMPORTANT! *If your function accesses the array arguments in random order, you must not specify the `SUPPORTS_STREAMING` pragma or an error will be returned when generating output or verifying your design.*

The following is an example of a function that accesses the array arguments in a strictly sequential order, supporting the streaming of data. This function flips the rows of an input image horizontally. The function accesses the input image in a sequential order and buffers two rows of the input image in a circular buffer. Once two full rows are buffered, the function writes the buffer content to the function argument in a sequential order. As such, this function supports streaming, and uses the `SUPPORTS_STREAMING` pragma to specify it.

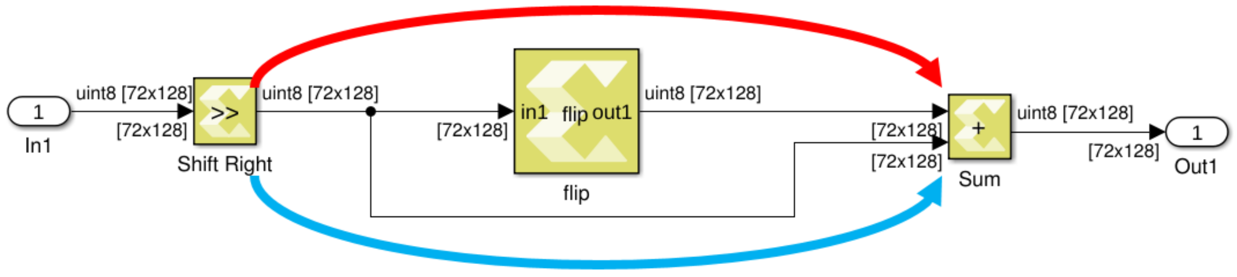
```
#ifndef _MY_FUNCS
#define _MY_FUNCS

#include <stdint.h>

#pragma XMC INPORT in1
#pragma XMC OUTPORT out1
#pragma XMC SUPPORTS_STREAMING
#pragma XMC BUFFER_DEPTH 4+2*WIDTH
// This function reverses each of the rows of the input image.
template<int WIDTH, int HEIGHT>
void
flip(uint8_t in1[HEIGHT][WIDTH],
     uint8_t out1[HEIGHT][WIDTH])
{
#pragma HLS dataflow

   uint8_t buf[2][WIDTH];

   int readBuf = 0;
   int writeBuf = 0;
   for (int row = 0; row < HEIGHT + 2; ++row) {
      for (int col = 0; col < WIDTH; ++col) {
#pragma HLS DEPENDENCE array inter false
#pragma HLS PIPELINE II=1
         if (row < HEIGHT) {
            buf[writeBuf][col] = in1[row][col];
            if (col == WIDTH-1) {
               ++writeBuf;
               writeBuf = (3 == writeBuf) ? 0 : writeBuf;
            }
         }
         if (row > 1) {
            out1[row - 2][col] = buf[readBuf][WIDTH- 1 - col];
            if (col == WIDTH-1) {
               ++readBuf;
               readBuf = (3 == readBuf) ? 0 : readBuf;
            }
         }
      }
   }
}
#endif
```

XMC BUFFER_DEPTH

The Model Composer `BUFFER_DEPTH` pragma provides information for properly sizing the buffers that connect the blocks in an implementation. These buffers are implemented as FIFOs in hardware. By default, Model Composer sets the depths of these buffers to 1. However, if your design has re-convergent paths (two paths converging into the same node) and the processing of data from the blocks of one path are not in lockstep with the processing of data from the other path, then a deadlock can occur. To avoid the deadlock the depth of one or more of the buffers on the paths can be increased to store the data. The following example illustrates this concept.

In the following diagram the `Sum` block consumes both the output signal of the `flip` block (red path), and the output of the `Shift Right` block (blue path). The `flip` block has been created with the `xmcImportFunction` command, and its source code is shown in the `flip` function previously described.

Figure 19: **Buffer Depth**



From the code for the `flip` block, you can see that the block needs to read 2 full rows before producing the first output. If the `BUFFER_DEPTH` pragma is not specified for the block, Model Composer sets the buffer sizes to 1 for the signals in the diagram. This results in deadlock, because the `flip` block reads 257 pixels from the input FIFO before producing the first output. However, by default, the parallel blue path feeding the second input of `Sum` has only enough storage for 1 pixel.

TIP: *Vivado HLS provides some capability to detect deadlocks during C/RTL co-simulation. In case a deadlock is detected, the tool prints out messages showing which FIFOs are involved in the deadlock, to help identify FIFOs that may require a BUFFER_DEPTH of more than 1.*

To change the default BUFFER_DEPTH, as shown in the `flip` function, place the pragma in the header file before the function declaration:

```
#pragma XMC BUFFER_DEPTH <depth>
```

Where `<depth>` specifies the buffer depth, and can be specified as a value or an expression.

By specifying `#pragma XMC BUFFER_DEPTH 4+2*WIDTH` in the `flip` function, Model Composer can determine that there is an imbalance in processing among the re-convergent paths, and address this imbalance by setting the buffer depth for the second input to the `Sum` block (blue path) to match the buffer depth of the `flip` block.

TIP: *Determining the minimal buffer depth may require a bit of trial and error because it also depends on the timing of the reads and writes into the FIFOs in the RTL code. In the `flip` function example, 256 (or `2*WIDTH`) was not sufficient `BUFFER_DEPTH`, but 260 (or `4+2*WIDTH`) prevented the deadlock.*

XMC THROUGHPUT_FACTOR

The Model Composer `THROUGHPUT_FACTOR` pragma provides some control over the throughput of an `xmcImportFunction` block. You can add the `THROUGHPUT_FACTOR` pragma to your function header file, along with the `SUPPORTS_STREAMING` pragma as shown in the following example:

```
#pragma XMC THROUGHPUT_FACTOR TF_param: 1,2,4
#pragma XMC SUPPORTS_STREAMING
template<int ROWS, int COLS, int TF_param>
void DilationWrap(const uint8_t src[ROWS][COLS], uint8_t dst[ROWS][COLS])
```

The syntax of the pragma as shown in the prior example is:

```
#pragma XMC THROUGHPUT_FACTOR TF_param: 1,2,4
```

Where:

- The *TF_param* must be an `int` type template parameter, as is in the example above.
- It is optional, though recommended, to specify any specific throughput factors that are supported by the function. In the example above, *1,2,4* specifies the supported throughput factors in the pragma, expressed as positive integers, and must include the value 1. If you do not explicitly specify the throughput factors, the *TF_param* is assumed to be valid for any positive throughput factor up to the upper limit of 16 that is supported by Model Composer.

As discussed in Controlling the Throughput of the Implementation, you specify the throughput factor for the model in the Model Composer Hub block. You can specify a throughput factor for the Hub block that divides evenly into one of the `THROUGHPUT_FACTOR` values on the `xmcImportFunction` block. For example, a Computer Vision function may support throughput factors of 1 and 8, but you can specify a throughput factor of 4 in the Hub block. Since 4 is a factor of 8, the throughput of 4 is accepted for the Hub block, but code generation inserts additional logic to convert the data flow between throughput of 4 on the Hub block and throughput of 8 on the Computer Vision function block. This additional logic increases resource utilization in the generated output.

IMPORTANT! *If the throughput factor of the Hub block does not match, or does not divide evenly into the* `THROUGHPUT_FACTOR` *specified by the* `xmcImportFunction` *block, then the throughput is reduced to 1 for the block function.*

Please note the following requirements:

- `THROUGHPUT_FACTOR` pragma must be used on Template functions.
- `THROUGHPUT_FACTOR` pragma must be used with `SUPPORTS_STREAMING` pragma.
- Only one `THROUGHPUT_FACTOR` pragma can be specified for an `xmcImportFunction` block.
- The block function will be called with actual arguments that have cyclic `ARRAY_RESHAPE` directives with factor=TF (see example below). For more information on the `ARRAY_RESHAPE` pragma, refer to HLS Pragmas in the *Vitis Unified Software Platform Documentation* (UG1416).
- The read accesses from a non-scalar input argument of the function should be compliant with the requirements for streaming, and Vivado HLS should be able to combine groups of TF reads into 1 read of the reshaped array.
- The write accesses into a non-scalar output argument of the function should be compliant with the requirements for streaming, and Vivado HLS should be able to combine groups of TF writes into 1 write of the reshaped array.

The following is an example function specifying both `SUPPORTS_STREAMING` and `THROUGHPUT_FACTOR` pragmas:

```
#include <stdint.h>

#pragma XMC THROUGHPUT_FACTOR TF: 1, 2, 4, 8, 16
#pragma XMC SUPPORTS_STREAMING
template<int TF>
void mac(const int32_t In1[240], const int32_t In2[240], const int32_t
In3[240],
         int32_t Out1 [240])
{
    #pragma HLS ARRAY_RESHAPE variable=In1 cyclic factor=TF
    #pragma HLS ARRAY_RESHAPE variable=In2 cyclic factor=TF
    #pragma HLS ARRAY_RESHAPE variable=In3 cyclic factor=TF
    #pragma HLS ARRAY_RESHAPE variable=Out1 cyclic factor=TF

    for (uint32_t k0 = 0; k0 < 240 / TF; ++k0) {
        #pragma HLS pipeline II=1
        int32_t Product_in2m[TF];
        int32_t Sum_in2m[TF];
        int32_t Product_in1m[TF];
        int32_t Sum_outm[TF];
        for (uint32_t k1 = 0; k1 < TF; ++k1) {
            Product_in2m[k1] = In2[(k0 * TF + k1)];
        }
        for (uint32_t k1 = 0; k1 < TF; ++k1) {
            Sum_in2m[k1] = In3[(k0 * TF + k1)];
        }
        for (uint32_t k1 = 0; k1 < TF; ++k1) {
            Product_in1m[k1] = In1[(k0 * TF + k1)];
        }
        for (uint32_t k1 = 0; k1 < TF; ++k1) {
            int32_t Product_in2s;
            int32_t Sum_in2s;
            int32_t Product_in1s;
            int32_t Product_outs;
            int32_t Sum_outs;
            Product_in2s = Product_in2m[k1];
            Sum_in2s = Sum_in2m[k1];
            Product_in1s = Product_in1m[k1];
            Product_outs = Product_in1s * Product_in2s;
            Sum_outs = Product_outs + Sum_in2s;
            Sum_outm[k1] = Sum_outs;
        }
        for (uint32_t k1 = 0; k1 < TF; ++k1) {
            Out1[(k0 * TF + k1)] = Sum_outm[k1];
        }
    }
}
```

Adding Your Library to Library Browser

To use the imported blocks in your library, you can simply open the Simulink model of your library, and copy blocks into a new Model Composer model. However, if you want to see your library listed in the Library Browser, and be able to drag and drop blocks from the library into new models, after running the `xmcImportFunction` command you must prepare your library using the following process.

1. Enable Library Browser parameter.
2. Save the library.
3. Create `slblocks.m` script for the library.
4. Add path to MATLAB, or add library to MATLAB path.
5. Refresh Library Browser.

TIP: *Setting up the library using this process is only required for newly created libraries, rather than existing libraries which have already been setup.*

To enable the library to be available in the Library Browser, you must turn on the `EnableLBRepository` parameter for the library. After importing a block into a new library using `xmcImportFunction`, with the library open, you must use the following command from the MATLAB command line prior to saving your library:

```
set_param(gcs,'EnableLBRepository','on');
```

This parameter identifies the library as belonging to the Library Browser. However, that is just the first step. Save the library by using the **File→Save** command from the main menu, or by clicking on the button in the toolbar.

Libraries in the Library Browser also require the presence of a script in the same directory as the library model, called `slblocks.m`, that defines the metadata associated with the library. You can create this script by copying an existing script from another library, or copying it from the MATLAB installation, or by editing the following text and saving it as `slblocks.m`:

```
function blkStruct = slblocks
  % This function adds the library to the Library Browser
  % and caches it in the browser repository

  % Specify the name of the library
  Browser.Library = 'newlib';
  % Specify a name to display in the library Browser
  Browser.Name = 'New Library';

  blkStruct.Browser = Browser;
```

Notice the `Browser.Library` specifies the name of the library model minus the `.slx` file extension, and `Browser.Name` specifies the display name that will appear in Library Browser.

Note: Each library should appear in a separate directory, with the `<library>.slx` file and the `slblocks.m` script for that library.

After creating the library and the `slblocks.m` script, you need to either add the library location to the MATLAB path so that MATLAB can find it, or copy the library to a folder that is already on the MATLAB path. You can type `path` at the MATLAB command prompt to see the current path that MATLAB uses. You can also add the library directory to the MATLAB path using the following commands:

```
addpath ('library_folder')
savepath
```

Where:

- `'library_folder'` is a string that defines the path to the new library,
- `savepath`

TIP: *To remove a folder from the MATLAB path you can use the following command sequence:*

```
rmpath ('library_folder')
savepath
```

Finally, to view the new library in the Library Browser, from the left side of the Library Browser window right-click and select the **Refresh Library Browser** command. This will load the library into the tool. You should now be able to view the imported blocks and drag and drop them into your models.

Debugging Imported Blocks

Model Composer provides the ability to debug C/C++ code that has been imported as a block using the `xmcImportFunction` command, while simulating the entire design in Simulink®. In this case, the `xmcImportFunctionSettings` command can be used to set the debugging tool.

This feature lets you build the C/C++ function with debug information, and load it into Simulink for simulation. Once loaded, you can step into the C/C++ code of a specific imported block, and debug the function. The debugging environment lets you set debug break points in the C/C++ code, step through, and observe the intermediate results to verify the function in the context of the simulation. Debugging C/C++ code during Simulink simulation provides a natural flow. You can set desired input stimulus in Simulink, and observe the effect stepping through the code.

The debug flow in Model Composer uses the following steps:

1. Specify the debug tool using the `xmcImportFunctionSettings` command.
2. Launch the debugging tool.
3. Add a breakpoint in the imported function.
4. Attach to the MATLAB® process.
5. Start Simulink simulation.
6. Debug the imported function during simulation.

Enable Debug Mode

Debugging the C/C++ function requires the simulation model to be built in the debug mode, instead of being built for release. You will use the `xmcImportFunctionSettings` command to configure the build mode prior to launching simulation. To enable the debug build, use the following command:

```
xmcImportFunctionSettings('build', 'debug')
```

Refer to xmcImportFunctionSettings Command Syntax for more information on the command options. Enabling debug mode in Windows operating system causes Model Composer to return the following messages in the MATLAB® Command Window:

```
Imported C/C++ code will be built with MinGW compiler. You can use gdb to
debug your C/C++ code.
MATLAB process ID is 4656.
You can also get the process ID by typing "feature getpid" in the MATLAB
command window.
```

The information returned above can be used to launch the default debug tool, and connect to the MATLAB process, as described in the following sections.

TIP: *You can restore the release build environment, using the 'release' value of the 'build' option:*

```
xmcImportFunctionSettings('build','release')
```

Launch the Debug Tool

After enabling the debug build mode, the `xmcImportFunctionSettings` command returns a link to the suggested debugging tool.

A third-party debugger is required for debugging with Model Composer. The default debugger is GDB for both the Linux and Windows operating systems.

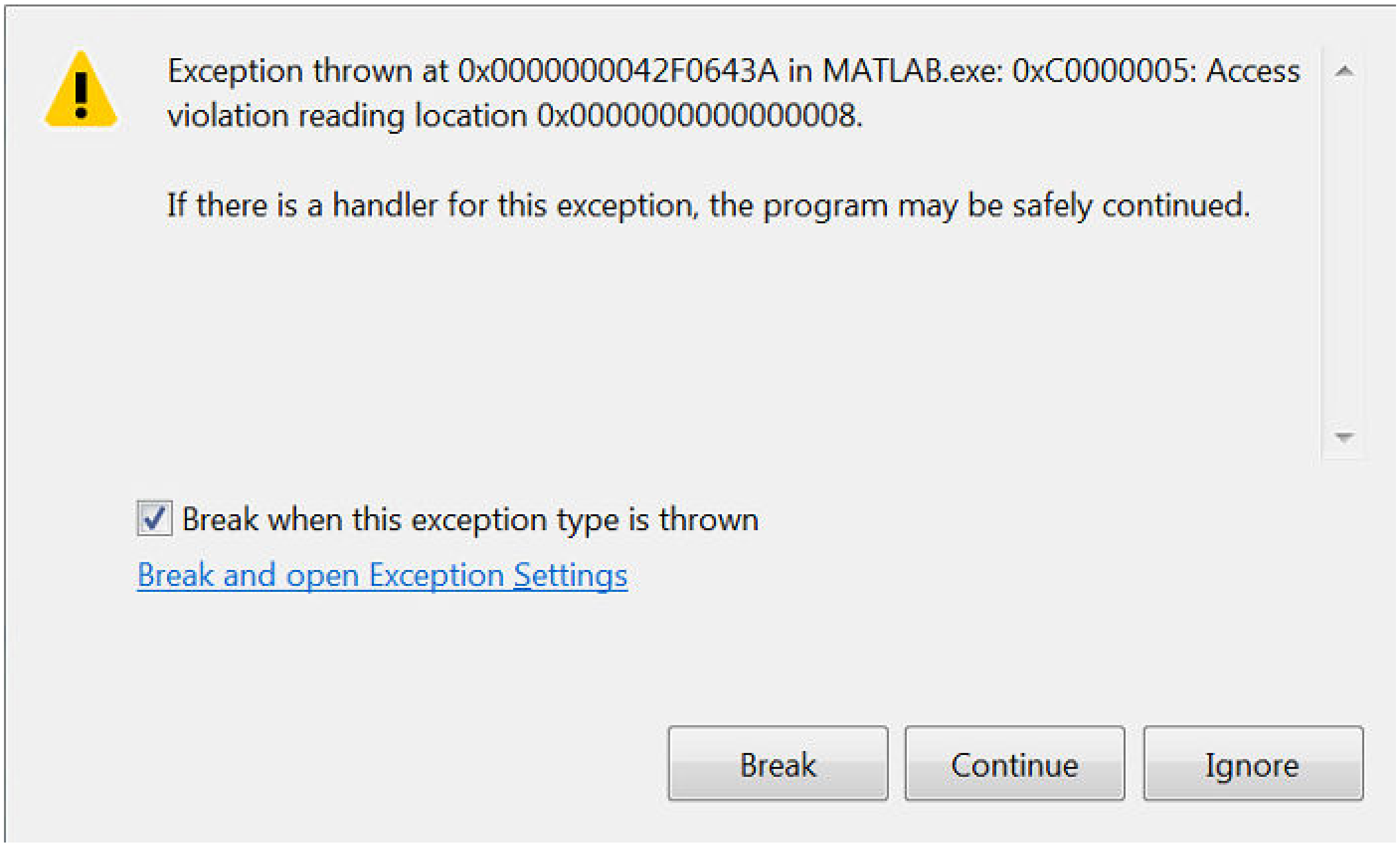
In the Windows environment, you can also specify the use of Microsoft Visual Studio as the compilation, and debugging tool. This requires the use of the `xmcImportFunctionSettings` command using the 'compiler' option, and specifying the 'Visual Studio' value:

```
xmcImportFunctionSettings('compiler','Visual Studio')
```

The Model Composer tool will look for Visual Studio in some default installation locations, but you can also specify the program location if needed.

TIP: *When debugging with Visual Studio, MATLAB may throw exceptions that can be safely ignored. When this happens disable the* ***Break when this exception type is thrown*** *checkbox, and click* ***Ignore****.*

Figure 20: **Visual Studio Exception**



Exception thrown at 0x0000000042F0643A in MATLAB.exe: 0xC0000005: Access violation reading location 0x0000000000000008.

If there is a handler for this exception, the program may be safely continued.

Break when this exception type is thrown
Break and open Exception Settings
Break
Continue
Ignore

Setting a Breakpoint for the Imported Function

When the debugger is launched, you can set a breakpoint for the C/C++ imported function in the current model. This will break, or pause the Simulink simulation at the point it enters the imported function. This lets you perform further debugging actions, such as stepping through the function, printing variable values, or listing lines of code. Refer to the documentation for your debugging tool for more information on specific commands, and debugging techniques.

TIP: *The following commands are provided for GDB, as it is the default debugger for Model Composer.*

Setting a breakpoint uses the function name of the imported function:

```
(gdb) break <function_name>
```

Because simulation has not yet started, GDB will respond that no symbol table is loaded, and indicate that you can use the "file" command to specify break points. This simply means that you can also specify breakpoints based on the source file for the imported C/C++ function, and line number, specified as follows:

```
(gdb) break <file name>:<line num>
```

For example: `break func3_d.h:10`

IMPORTANT! *Blocks created from function templates, as described in Defining Blocks Using Function Templates, require the file name and line number to set breakpoints.*

Connecting Debug to the MATLAB Process

With the breakpoint established, you can link GDB to the MATLAB® process by using the GDB attach command, and specifying the process ID (PID) returned by the `xmcImportFunctionSettings` command when you set up the debug build mode, as previously discussed. Use the following command:

```
(gdb) attach <PID>
```

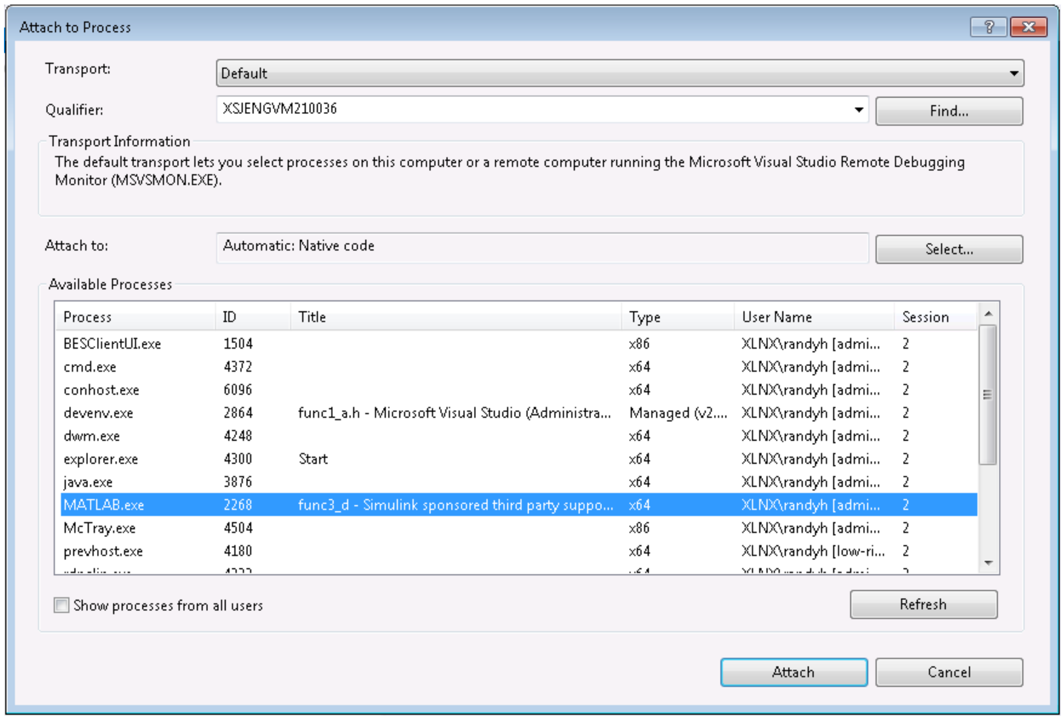
IMPORTANT! *After it is attached to GDB, the MATLAB process will be suspended. You must use the `continue` command to have the process resume running after the `gdb` prompt is returned:*

```
(gdb) continue
```

At this point you are ready to start the Simulink® simulation, and begin debugging your design.

When using Visual Studio, you can launch the application, and then select **Debug → Attach to Process** command to attach to the MATLAB process. This command opens an Attach to Process dialog box that lets you select the MATLAB process from the list of running processes, to attach to the Visual Studio debugging session.

Figure 21: **Visual Studio - Attach to Process**



Attach to Process

Transport: Default

Qualifier: XSJENGVM210036

Transport Information
The default transport lets you select processes on this computer or a remote computer running the Microsoft Visual Studio Remote Debugging Monitor (MSVSMON.EXE).

Attach to: Automatic: Native code

Available Processes

Process	ID	Title	Type	User Name	Session
BESClientUI.exe	1504		x86	XLNX\randyh [admi...	2
cmd.exe	4372		x64	XLNX\randyh [admi...	2
conhost.exe	6096		x64	XLNX\randyh [admi...	2
devenv.exe	2864	func1_a.h - Microsoft Visual Studio (Administra...	Managed (v2....	XLNX\randyh [admi...	2
dwm.exe	4248		x64	XLNX\randyh [admi...	2
explorer.exe	4300	Start	x64	XLNX\randyh [admi...	2
java.exe	3876		x64	XLNX\randyh [admi...	2
MATLAB.exe	2268	func3_d - Simulink sponsored third party suppo...	x64	XLNX\randyh [admi...	2
McTray.exe	4504		x86	XLNX\randyh [admi...	2
prevhost.exe	4180		x64	XLNX\randyh [low-ri...	2

Show processes from all users

IMPORTANT! *When attaching Visual Studio to the MATLAB process, you must also click **Select** to set the **Attach to** field, and explicitly choose **Native Code** from the drop down menu. The **Automatic:Native** mode option may not work for some versions of Visual Studio.*

xmcImportFunctionSettings Command Syntax

The `xmcImportFunctionSettings` command sets options for the import function feature in Model Composer. Options are specified in the form of name/value pairs. The current options include:

- `build`: Specifies the build environment for Model Composer. Supported values include:
  - `release`: the default build mode. Generates the specified output, and lets you perform simulation.
  - `debug`: Provides integration into a third-party compiler to let you step through and observe the imported C/C++ function using standard debug features.
- `compiler`: Specifies the third-party compiler to use for debugging purposes. The supported values are:
  - `default`: Compile with GCC or MinGW compiler for debugging using GDB debugger. Note that GCC or MinGW, and GDB are included with the standard installation of Model Composer.
  - `Visual Studio`: Compile and debug C/C++ code with Visual Studio.

- `blocks`: Specifies the specific block or blocks that you want to observe during the debugging process. If the option is not specified, all imported function blocks in the current design are included for debugging. You can specify one or more blocks using the following command form:

  ```
  xmcImportFunctionSettings('blocks', {'block1','block2', ...})
  ```

  You can also unset the selection of the blocks using the following command:

  ```
  xmcImportFunctionSettings('blocks', {})
  ```

  You can use the MATLAB `gcb` command to get the block path for a specific block. The blocks must be specified as a list of blocks, even if only one block is specified. For example:

  ```
  xmcImportFunctionSettings('blocks', {'xmc_optical_flow/Lucas-Kanade'})
  ```

- `Visual Studio Location`: Specify the path to the Visual Studio tool. By default, Model Composer will automatically look for Visual Studio in some standard locations. You can use this option to specify the location of the tool in cases where your installation may not be standard.
- `vcvars`: Specifies the location of the `vcvarsall.bat` file used to configure the Visual Studio environment for command-line processing. This option can be used to point to the `vcvarsall.bat` file when the `Visual Studio Location` option specifies a non-standard location for the tool.

TIP: *The settings specified by `xmcImportFunctionSettings` remain set until you exit Model Composer, or change the options using `xmcImportFunctionSettings`.*

Generating Outputs

Introduction

IMPORTANT! *To generate output from the Model Composer model, only Xilinx Model Composer blocks and a limited set of Simulink blocks can be used in the subsystem that is instantiated at the top-level of the design. The Simulink blocks compatible with output generation in Model Composer can be found in the Xilinx Model Composer block library. Refer to Supported Simulink Blocks for a complete list.*

Model Composer automatically compiles designs into low-level representations. However, a Model Composer model requires the addition of the Model Composer Hub block to configure compilation and generate outputs. Model Composer can create three different types of output from the model, as defined by the **Export type** setting of the Model Composer Hub block:

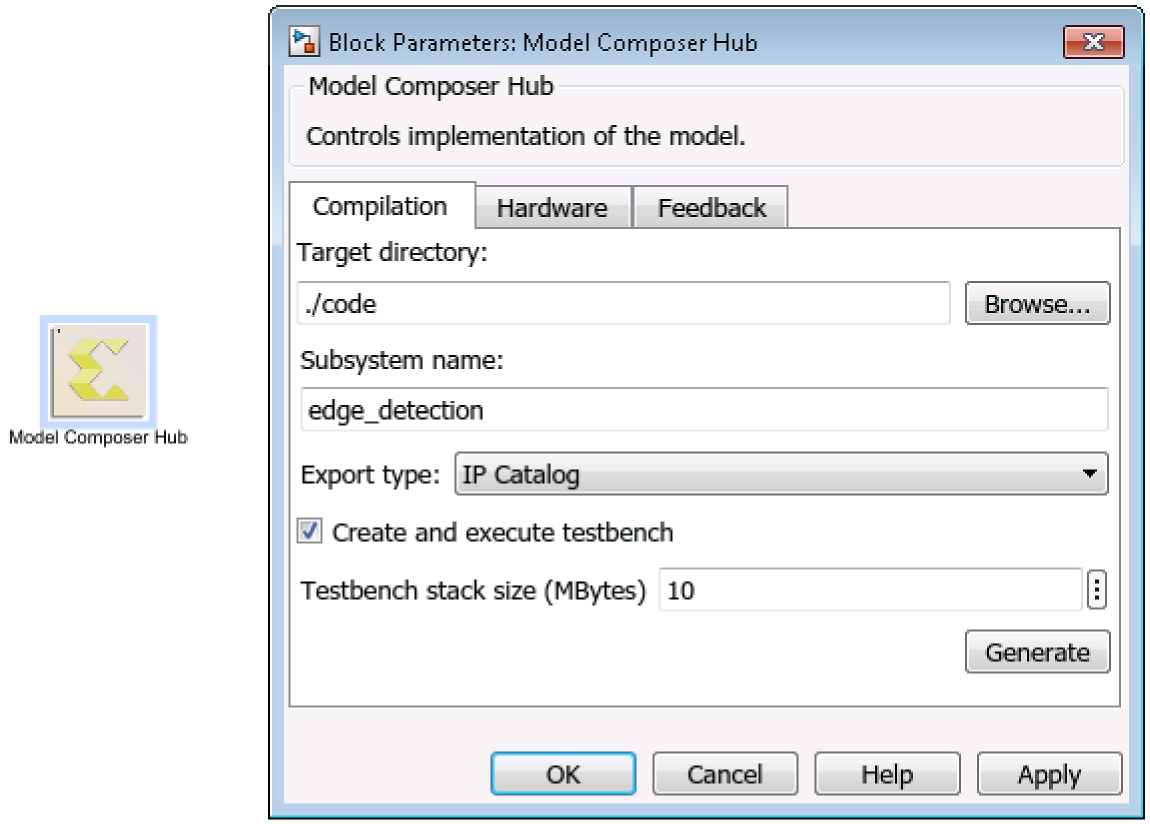
- IP Catalog
- System Generator
- C++ Code

Adding the Model Composer Hub

The Model Composer Hub block is a member of the **Tools** blockset within the Xilinx® Model Composer library. You can add it to your model just like any other block, by dragging it from the Library Browser onto the canvas of the Simulink® Editor. The Model Composer Hub block is a virtual block in Simulink® terms. It does not provide a physical purpose in the design, but rather provides directives for the compilation and output of the design.

The Model Composer Hub block and the Block Parameters dialog box are shown below.

Figure 22: **Model Composer Hub Block**



Block Parameters: Model Composer Hub

Model Composer Hub

Controls implementation of the model.

Target directory: ./code

Subsystem name: edge_detection

Export type: IP Catalog

☑ Create and execute testbench

Testbench stack size (MBytes) 10

You can see the Model Composer Hub block dialog block has three tabs, with the following features:

- **Compilation** tab
  - **Target Directory**: Defines the output folder where the compiled results will be written. The output generated by Model Composer can include a number of folders and files, which will all be written into the specified directory. The folder can be specified as an absolute path (e.g., C:/Data/Code), or a path relative to the current Model Composer model (e.g., ./code).

IMPORTANT! *On the Windows operating system, you cannot specify a **Target Directory** name with a space in it.*

  - **Subsystem name**: Specifies the name of the Model Composer subsystem located in the top-level of the model that is required to generate output. Refer to Creating a Top-Level Subsystem Module for more information. The subsystem should also have an Interface Specification block as discussed in Defining the Interface Specification. The subsystem name determines the name of the generated output files (e.g., `<subsystem_name>.cpp`,`<subsystem_name>.h`).

TIP: *The Interface Spec block is not required to generate output, but it is recommended in order to have full control over the interface specification of the design.*

- ◦ **Export type**: Select one of the available output products from Model Composer. These include packaged IP to add to the Vivado Design Suite IP catalog, System Generator block for use in RTL level block design, and C++ code for use in Vivado HLS.
- ◦ **Create and execute testbench**: When enabled this checkbox causes a simulation testbench to be created and launched for the output code. The simulation compares the "golden" results from Simulink with the results obtained from the newly compiled design. Refer to Chapter 5: Simulating and Verifying Your Design for more information.
  - **Testbench stack size**: When **Create and execute testbench** is enabled, this option specifies the size, in Megabytes, of the testbench stack frame used during C simulation (CSIM).

TIP: *Occasionally, the default stack frame size of 10 MB allocated for execution of the testbench may be insufficient to run the test, due to large signals arrays allocated on the stack and deep nesting of sub-systems. Typically when this happens, the test would fail with a segmentation fault and an associated error message. In such cases you may increase the size of the stack frame and re-run the test.*

- ◦ **Generate**: Compiles and writes the output from the Model Composer model.

- **Hardware** tab
  - ◦ **Project device**: Defines the current target part or board platform for the Model Composer model.
  - ◦ Browse button (**...**): Displays the Device Chooser dialog box. Refer to Device Chooser Dialog Box for more information.
  - ◦ **FPGA Clock Frequency** (MHz): Specifies the clock frequency in MHz for the Xilinx device. This frequency is passed to the downstream tool flow.
  - ◦ **Throughput Factor**: Specifies the data throughput requirement for the application, effecting the amount of data resources used in implementing the function in hardware.
- **Feedback** tab
  - ◦ **Connect**: You can provide feedback on the Xilinx Model Composer tool, highlight problems or difficulties, suggest enhancements to the tool, or recommend new blocks for the Model Composer library.

When you have specified the target directory, the subsystem module, and the export type, you are ready to click the **Generate** button to create the specified output. In addition to producing the C++ code, Model Composer generates the files needed to verify the code, and the necessary directives to synthesize the high-level code into RTL.

TIP: *If you make any changes to the settings of the Model Composer Hub block, you will need to* ***Apply*** *those changes prior to clicking Generate.*

Controlling the Throughput of the Implementation

Introduction

Throughput of a system is one of its most important design criteria. For example, if you are designing a system that processes High Definition video frames (1920x1080) at 30 frames per second, the required throughput of your application would be 62,208,000 pixels per second (1920x1080x30). If you process one pixel per clock (in hardware terminology this is called an initiation interval of one, or II=1), your device needs to be clocked at over 62.2 MHz. If your requirements change, and you need to process a 4K video frame at 60 frames per second, the required throughput of the application would be 497,664,000 pixels per second (3840x2160x60), and your device needs to be clocked at over 497 MHz.

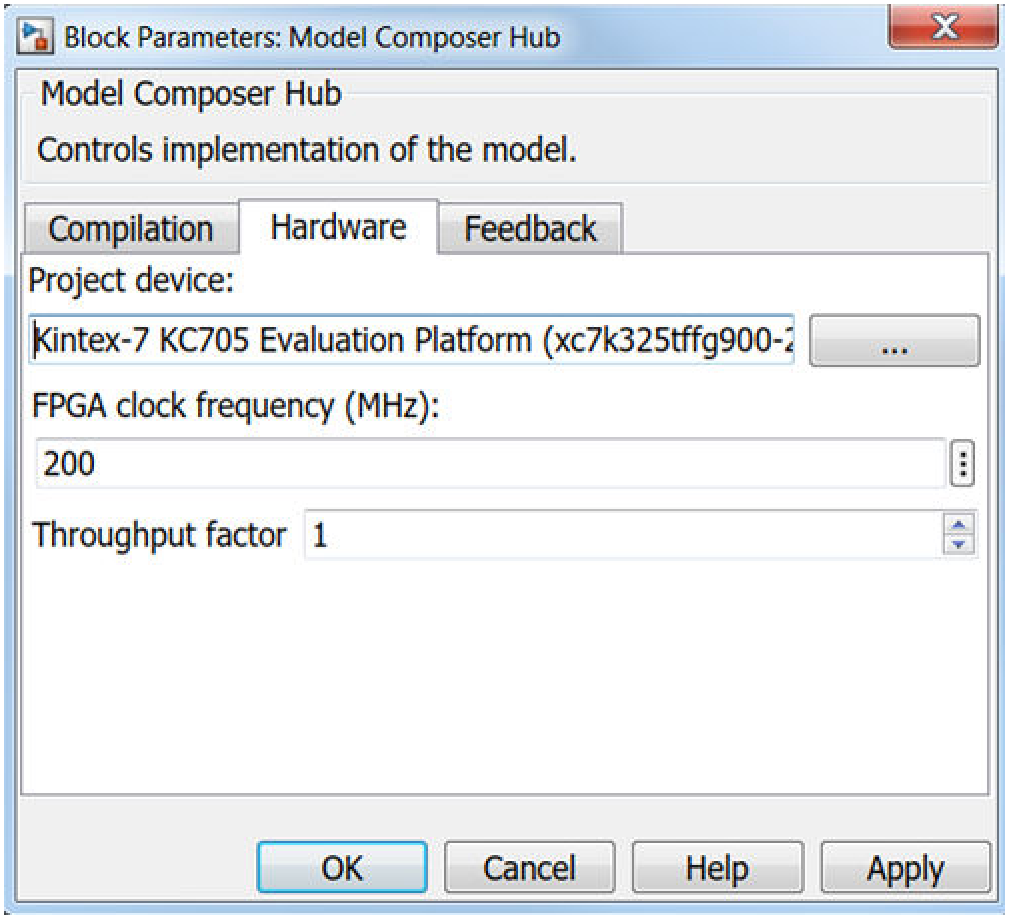
However, in practice you may not be able to achieve an initiation interval of one (II=1), therefore to achieve the desired throughput, you need to clock the device at even higher rates. In other applications, such as wireless communications, the clock frequencies needed to achieve a desired throughput could easily surpass the maximum clock frequency allowed for a device.

If you need to increase the throughput of your design, without increasing the clock frequency that your device is operating at (to operate at a clock frequency below the maximum allowed for a device, or to curtail power consumption), you can take advantage of the programmable logic nature of Xilinx FPGAs, and use parallelization techniques to process more samples per clock. Throughput control in Model Composer allows the user to do exactly that without making structural changes to their design in Simulink.

Setting Throughput Factor from the Hub block

The Model Composer Hub block provides Throughput Factor to control the throughput of the generated code, or hardware design. The Throughput Factor specifies how many sample elements of the inputs are to be processed per clock cycle. By default, the Throughput Factor is set to 1. You can specify this factor by clicking on the **Hardware** tab of the Model Composer Hub block, as shown below:

Figure 23: **Model Composer Hub - Throughput Factor**



Model Composer Hub

Controls implementation of the model.

Project device:

Kintex-7 KC705 Evaluation Platform (xc7k325tffg900-2

FPGA clock frequency (MHz):

200

Throughput factor 1

The Throughput Factor must be between 1 and 16. Specifying a value greater than 1 will create parallel logic to process the transactions, using more resources from the device, and increasing the HLS and Vivado Synthesis run time.

Code generation for designs with Throughput Factor > 1 imposes additional restrictions on the design. In case these restrictions are not met, Model Composer will return an error trying to explain the violation.

Restrictions on Using Throughput Control

After you set the value of Throughput Factor, and click the **Generate** button on the Hub, it triggers the compilation of the subsystem. A Throughput Factor of more than 1 can be achieved only if the design complies with the following restrictions:

- The Throughput Factor must be between 1 and 16.
- The subsystem must have at least one non-scalar input port.

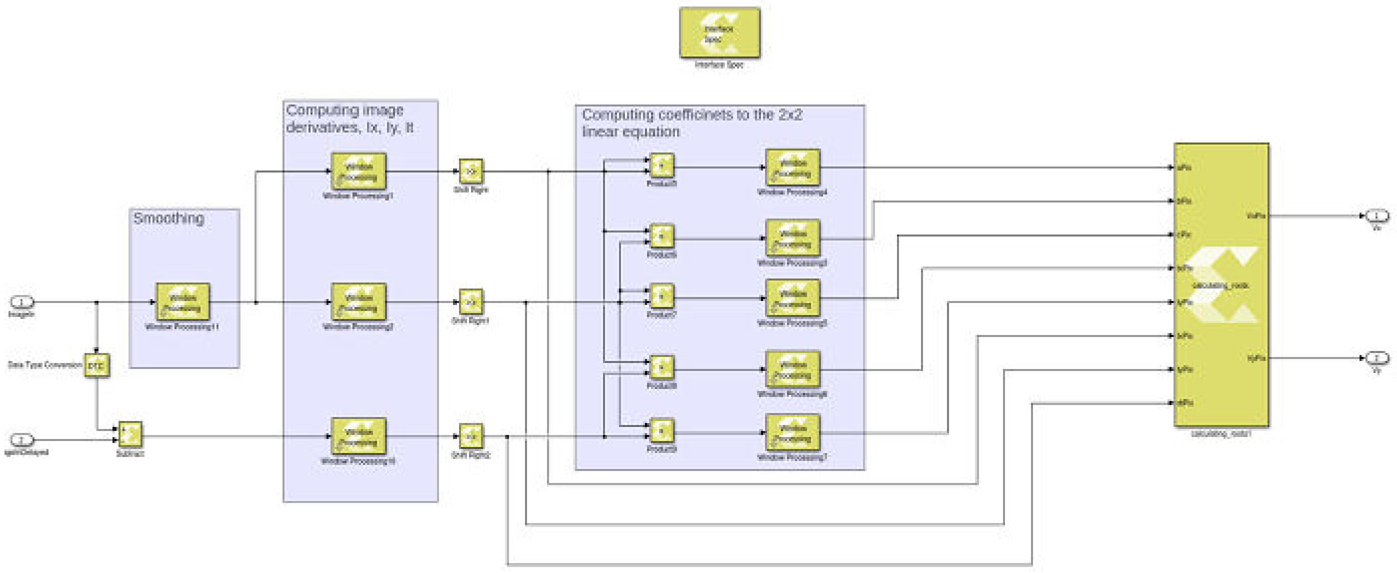
- All of the non-scalar ports of the subsystem must use either AXI4-Streaming or FIFO interfaces.
- For any vector signal within the subsystem, but not inside a Window Processing block kernel, the vector length must be a multiple of the Throughput Factor.
- For any matrix signal within the subsystem, but not inside a Window Processing block kernel, the number of columns must be a multiple of the Throughput Factor.
- Except for blocks inside a Window Processing block kernel, the subsystem must not include any of the following blocks:
  - look up
  - matrix multiply, QR inverse
  - transpose, hermitian
  - sum of elements and product of elements with floating point input
  - cumulative sum, reducing min, reducing max
  - if action subsystem
  - FFT, IFFT, FIR
- Many of the computer vision blocks only support Throughput Factor values up to 8, and some do not support Throughput Factor > 1. Refer to Throughput Control for Computer Vision Blocks for more information.
- For blocks created using `xmcImportFunction`, refer to XMC THROUGHPUT_FACTOR.

In summary, if the specified Throughput Factor is >1, and the design complies with all above mentioned restrictions, Model Composer can generate models that process samples concurrently. In cases where the design does not meet these restrictions, Model Composer will not generate an output model.

Understanding Throughput Control Through an Example

The following section demonstrates the benefits of using the Throughput Control feature with the Optical flow example design found in the list of examples for Xilinx Model Composer.

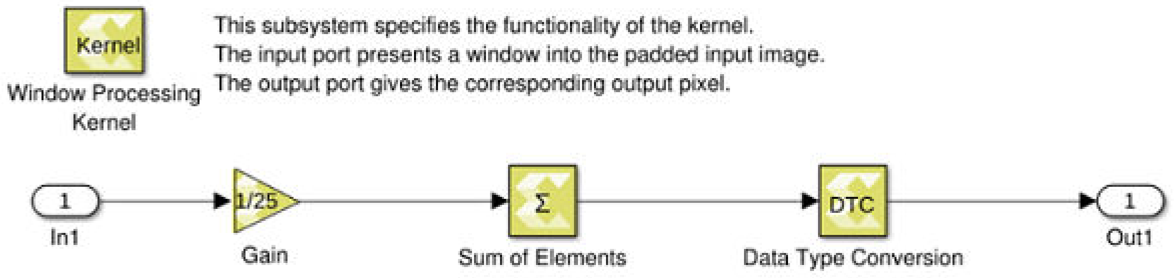
Figure 24: **Optical Flow Example**



This design uses the following blocks:

- Data Type Conversion, Subtract, Right Shift, Product
- Window processing blocks, with Gain, Sum of Elements, and Data Type Conversion.
- An Import Function block with the `calculating_roots` function.

Figure 25: **Window Processing Kernel**


This subsystem specifies the functionality of the kernel.
The input port presents a window into the padded input image.
The output port gives the corresponding output pixel.

All these blocks follow the element-wise application pattern, and comply with the restrictions previously discussed.

IMPORTANT! *Direct use of the Sum of Elements block in subsystems using Throughput Control is restricted. In this example, the 'Sum of Elements' block is used in the Window Processing block but not directly in the top-level subsystem.*

With the default Throughput Factor=1, Model Composer generates the code shown below:

```
void
Lucas_Kanade(hls::stream< uint8_t >& ImageIn, hls::stream< uint8_t >&
    ImageInDelayed, hls::stream< float >& Vx, hls::stream< float >& Vy)
{
    #pragma HLS INTERFACE axis port=ImageIn
    #pragma HLS INTERFACE axis port=ImageInDelayed
    #pragma HLS INTERFACE axis port=Vx
    #pragma HLS INTERFACE axis port=Vy
    #pragma HLS INTERFACE s_axilite port=return
    #pragma HLS dataflow
```

The IP reads its inputs, the image and delayed image, over AXI4-Stream. These streams will use a data width of 8 bits (1 pixel). Similarly pixels of the output image are streamed over an AXI4-Stream interface of data width 8 bits.

If we set TF=4, we get the code shown below.

```
void
Lucas_Kanade(hls::stream< xmc::MultiScalar< uint8_t, 4 > >& ImageIn,
    hls::stream< xmc::MultiScalar< uint8_t, 4 > >& ImageInDelayed,
    hls::stream< xmc::MultiScalar< float, 4 > >& Vx,
    hls::stream< xmc::MultiScalar< float, 4 > >& Vy)
{
    #pragma HLS INTERFACE axis port=ImageIn
    #pragma HLS data_pack variable=ImageIn
    #pragma HLS INTERFACE axis port=ImageInDelayed
    #pragma HLS data_pack variable=ImageInDelayed
    #pragma HLS INTERFACE axis port=Vx
    #pragma HLS data_pack variable=Vx
    #pragma HLS INTERFACE axis port=Vy
    #pragma HLS data_pack variable=Vy
    #pragma HLS INTERFACE s_axilite port=return
    #pragma HLS dataflow
```

This IP receives 4 pixels of the input image, and 4 pixels of the delayed input image, at the same time over AXI4-Stream that have data width of 32 bits. Inside the IP the logic has been duplicated so that 4 pixels are processed in parallel. The IP sends 4 pixels of the output image at a time over an AXI4-Stream, of data width 32 bits. Note that `xmc::MultiScalar<T,N>` is a template `struct` defined in `xmcMultiScalar.h`. It is a `struct` that contains an array of N elements of type T.

The following table represents the Vivado HLS timing and resource estimates for optical flow design.

Table 5: **Optical Flow Design Timing/Resource Utilization Estimates**

	Throughput factor = 1	Throughput factor = 4	Throughput factor = 8
Clock Freq	300 MHz	300 MHz	300 MHz
Latency/II	41848/41834	10483/10469	5358/5344
BRAM_18k (Utilization %)	5	2	4
DSP48E (Utilization %)	2	9	19

Table 5: **Optical Flow Design Timing/Resource Utilization Estimates** *(cont'd)*

	Throughput factor = 1	Throughput factor = 4	Throughput factor = 8
FF(Utilization %)	8	30	59
LUT(Utilization %)	14	36	88

The second line in the table shows the initiation interval (II). At clock frequency of 300 MHz and Throughput Factors 4 and 8, the initiation interval of the design is reduced by a factor of approximately 4 and approximately 8 respectively, when compared with the initiation interval for Throughput Factor=1. Note that this comes at the cost of increasing resource utilization when the Throughput Factor increases.

For Throughput factor of one, the II is 41,848. The input to this design is a 200x200 pixel image frame and the value of II here indicates the number of clocks to process the entire frame. As such it takes slightly more than the duration of one clock cycle to process one pixel. As the Throughput Factor increases, the II to process one frame decreases, and the application processes more than one pixel per clock cycle.

Throughput Control for Computer Vision Blocks

Model Composer does allow using most of the Computer Vision blocks in a design with Throughput Factors of up to 8. However, due to implantation reasons, when you use a Throughput Factor of more than one, and less than 8, these blocks will be processing 8 pixels per clock. While this will achieve the desired throughput, the block will use more hardware resources than is necessary.

There are further restrictions when we use Throughput Factors of more than one with Computer Vision blocks. First, the Throughput Factor must be supported by each of the Computer Vision blocks used as listed in the following table. Second, the number of columns of the image should be a multiple of the largest Throughput Factor supported by any of the Computer Vision blocks used, which is typically 8.

The following table lists Model Composer blocks that either do not support Throughput Factor, or have limited support for this feature. For more information on a specific block refer to Appendix B: Model Composer Block Library.

Table 6: **Computer Vision Blocks with Restrictions on Throughput Control**

Block Name	Throughput Control Limitations
2-D Convolution	Supports Throughput Factor 1, 2, 4, and 8.
Bilateral Filter	Does not support.
Dense Non-Pyramidal LK Optical Flow	Supports Throughput Factor 1 and 2.
Dilation	Supports Throughput Factor 1, 2, 4, and 8.
FAST Corner Detection	Supports Throughput Factor 1, 2, 4, and 8.

Table 6: **Computer Vision Blocks with Restrictions on Throughput Control** *(cont'd)*

Block Name	Throughput Control Limitations
Gaussian Blur	Supports Throughput Factor 1, 2, 4, and 8.
Gradient Magnitude	Supports Throughput Factor 1, 2, 4, and 8.
Gradient Phase	Supports Throughput Factor 1, 2, 4, and 8.
HOG Descriptor	Does not support.
Histogram	Supports Throughput Factor 1, 2, 4, and 8.
Histogram Equalization	Supports Throughput Factor 1, 2, 4, and 8.
Integral Image	Does not support.
Mean & Std Deviation	Supports Throughput Factor 1, 2, 4, and 8.
Median Blur Filter	Supports Throughput Factor 1, 2, 4, and 8.
MinMax Location	Supports Throughput Factor 1, 2, 4, and 8.
Otsu Thresholding	Supports Throughput Factor 1, 2, 4, and 8.
Pyramid Down	Does not support.
Pyramid Up	Does not support.
Remap	Does not support.
Resize	Does not support.
Sobel Filter	Supports Throughput Factor 1, 2, 4, and 8 for filter sizes 3x3 and 5x5. For filter size 7x7 the tool does not support Throughput Factor.
Stereo Block Matching	Does not support.
Warp Transform	Does not support.

Defining the Interface Specification

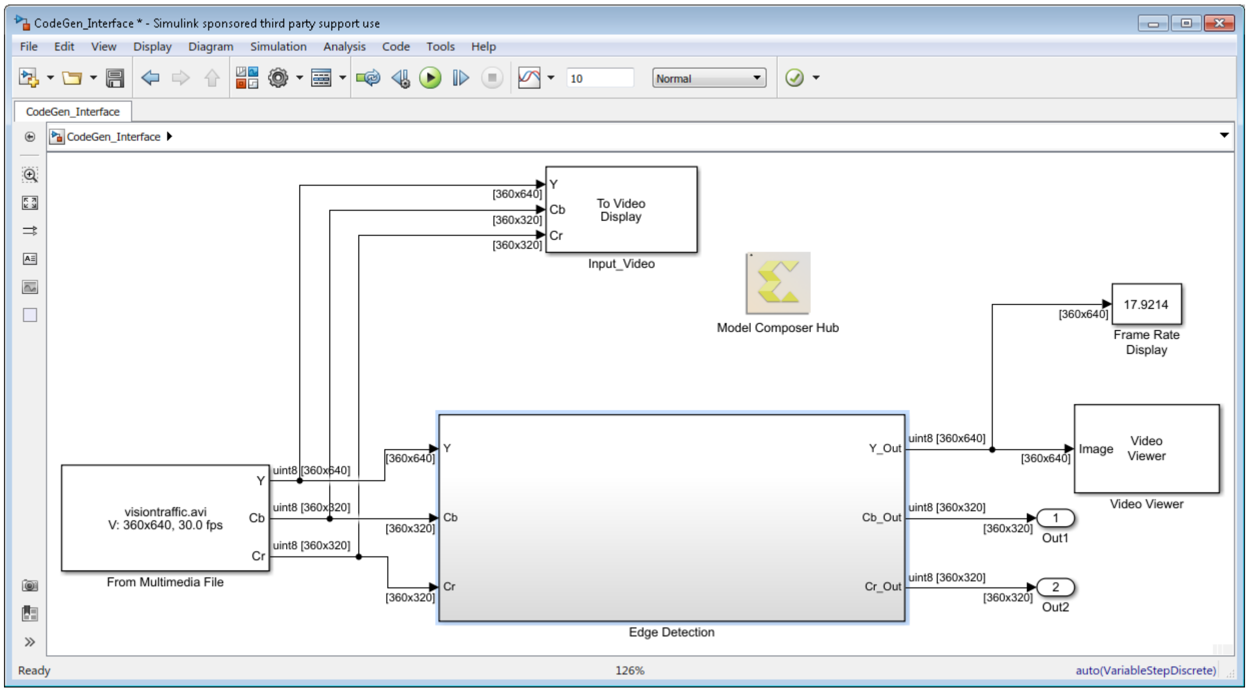
Within the Simulink® environment, the inputs and outputs in your design are defined using "Inport" and "Outport" blocks. However, while moving from the software algorithm to an RTL implementation in hardware, these same input and output ports must be mapped to ports in the design interface, using a specific input-output (I/O) protocol, which typically operates with some real world delay. Part of developing your design is to specify how your design will communicate with other designs or IP in the system. You do this by specifying the interface to your design and choosing among a few standard I/O protocols.

Model Composer requires the use of the Interface Specification (Interface Spec) block to define this I/O protocol.

Interface synthesis is supported only in the top-level subsystem module in the design, which Model Composer generates C++ code for. In the figure below, the Edge Detection module is the top-level subsystem module and the Interface Spec block must be instantiated inside that module.

TIP: *Any Interface Spec blocks instantiated in other subsystems modules, or nested subsystem modules are ignored.*

Figure 26: **Top-Level Subsystem Module**



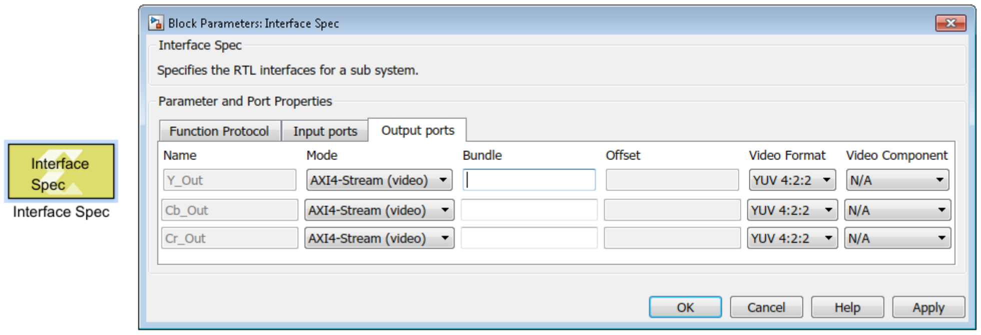
The Interface Spec block lets you control what interfaces should be used for the design. The Interface Spec affects only output code generation; it has no effect on Simulink simulation of your design. If you do not add an Interface Spec block to the subsystem module, Model Composer assigns default interfaces that may not be appropriate for the target platform or device. Therefore, it is recommended that you use the Interface Spec block to define the requirements of your subsystem module. The default function-level protocol is Handshake to specify control signals, and AXI4-Lite Slave for the function return. The default I/O protocol is AXI4-Lite Slave for scalar ports, and AXI4-Stream for non-scalar ports.

The Interface Spec block specifies how RTL ports are created from the function definition during interface synthesis. The ports in the RTL implementation are derived from the following.

- Any function-level protocol that defines control signals for the module.
- Function input and output arguments, and return values.
- Global variables accessed by the function but defined outside its scope.

  ***Note*:** If a global variable is accessed, but all read and write operations are local to the subsystem, the resource is created in the design, and does not require the definition of a port.

Figure 27: **Interface Spec Block**



Block Parameters: Interface Spec

Interface Spec

Specifies the RTL interfaces for a sub system.

Parameter and Port Properties

Function Protocol | Input ports | Output ports

Name	Mode	Bundle	Offset	Video Format	Video Component
Y_Out	AXI4-Stream (video)			YUV 4:2:2	N/A
Cb_Out	AXI4-Stream (video)			YUV 4:2:2	N/A
Cr_Out	AXI4-Stream (video)			YUV 4:2:2	N/A

The Interface Spec block consists of 3 tabs defining the following information:

- **Function Protocol**: This is the block-level interface protocol which adds signal ports to the subsystem telling the IP when to start processing data. It is also used by the IP to indicate whether it accepts new data, or whether it has completed an operation, or whether it is idle.
- **Input Ports**: This tab automatically detects the input ports in your subsystem and lets you specify the interface protocol on those ports.
- **Output Ports**: This tab automatically detects the output ports on the subsystem module, and lets you specify the interface protocol.

IMPORTANT! *The Interface Spec block has a current limitation of 8 input ports and 8 output ports on the subsystem module.*

The Interface Specification displays and lets you configure the following features or parameters of the function or I/O port protocol.

Table 7: **Function Protocol Tab**

Attribute	Description
Mode	Specifies a block-level protocol to add control signals to the subsystem module. The supported block-level protocols are: • **AXI4-Lite Slave**: Implements the return port as an AXI4-Lite Slave interface, and adds the block-level control ports defined by the Handshake protocol. This is the default function protocol. • **Handshake**: Defines a set of block-level control ports for the function to `start` processing input, and indicate when the design is `idle`, `done`, and `ready` for new input data. • **No block-level I/O protocol**: No control ports are added to the subsystem.
Bundle	Only valid with the AXI4-Lite Slave mode. Indicates that multiple ports should be grouped into the same interface. The bundle is specified by a `<name>` that cannot contain spaces or special characters.

Table 8: **Input/Output Port Tabs**

Attribute	Description
Name	Displays the port name, which cannot be changed from here.
Mode	Specifies the port-level I/O protocols. The supported port-level protocols are: • **Default**: Uses AXI4-Lite Slave interface for scalar ports, or AXI4-Stream interface for non-scalar ports. • **AXI4-Stream**: Implements ports as an AXI4-Stream interface for high-speed streaming data. • **AXI4-Stream (video)**: Implements ports as an AXI4-Stream interface, with the additional assignment of **Video Format** and **Video Component** attributes. • **AXI4-Lite Slave**: Implements the port as part of an AXI4-Lite Slave interface. All input or output ports with the same **Bundle** name are grouped into the same AXI4-Lite Slave interface. • **FIFO**: Implements the port with a standard FIFO interface, combining data input or output with associated active-low FIFO empty and full control signals. ***Note*:** The FIFO interface is the most hardware-efficient approach for access to a memory element that is always sequential, that is, no random access is required. To read from non-sequential address locations, use the Block RAM interface. • **Constant**: The data applied to the input port remains stable during the function operation, but is not a constant value that can be optimized. This allows internal optimizations to remove unnecessary registers. • **Valid Port**: Implements a data port with an associated `valid` port to indicate when the data is valid for reading or writing. • **No protocol**: No protocol. Neither the input or output data signals have associated control ports to indicate when data is read or written. • **Block RAM**: Implements array arguments as a standard RAM interface. If you use the generated IP in Vivado IP integrator, the memory interface appears as a single port.
Bundle	Used in conjunction with the AXI4-Stream (video) interfaces that have more than 1 color component. In this case there should be one port for each color component, and the ports should specify the same bundle `<name>` so they will be grouped into the same AXI4 Stream (video) interface. Also valid with the AXI4-Lite Slave mode. This parameter explicitly groups all interface ports with the same bundle `<name>` into the same AXI4-Lite Slave interface.
Offset	Only valid with the AXI4-Lite Slave mode. This parameter specifies an address offset associated with the port in the AXI4-Lite Slave address map. The offset is specified as a non-negative integer, with a default value of 0.

Table 8: **Input/Output Port Tabs** *(cont'd)*

Attribute	Description
Video Format	Only valid with the AXI4-Stream (video) mode. This parameter specifies the color format for a video stream. Valid formats include: • **YUV 4:2:2**: Video format based on brightness (luminance) and color (chrominance), with reduced color content. • **YUV 4:4:4**: Video format based on brightness (luminance) and color (chrominance), with full color content. • **RGB**: Video format based on separate Red, Blue, and Green color signals. • **Mono**: Specifies an audio format for your video.
Video Component	Only valid with the AXI4-Stream (video) mode. This parameter specifies the color component for a video format that uses more than one color component. Valid video components include: • **Y,U, V**: Specifies one component of the YUV video format. • **R, G, B**: Specifies one component of the RGB video format.

The choice of port-level interface protocol should take into account the following considerations:

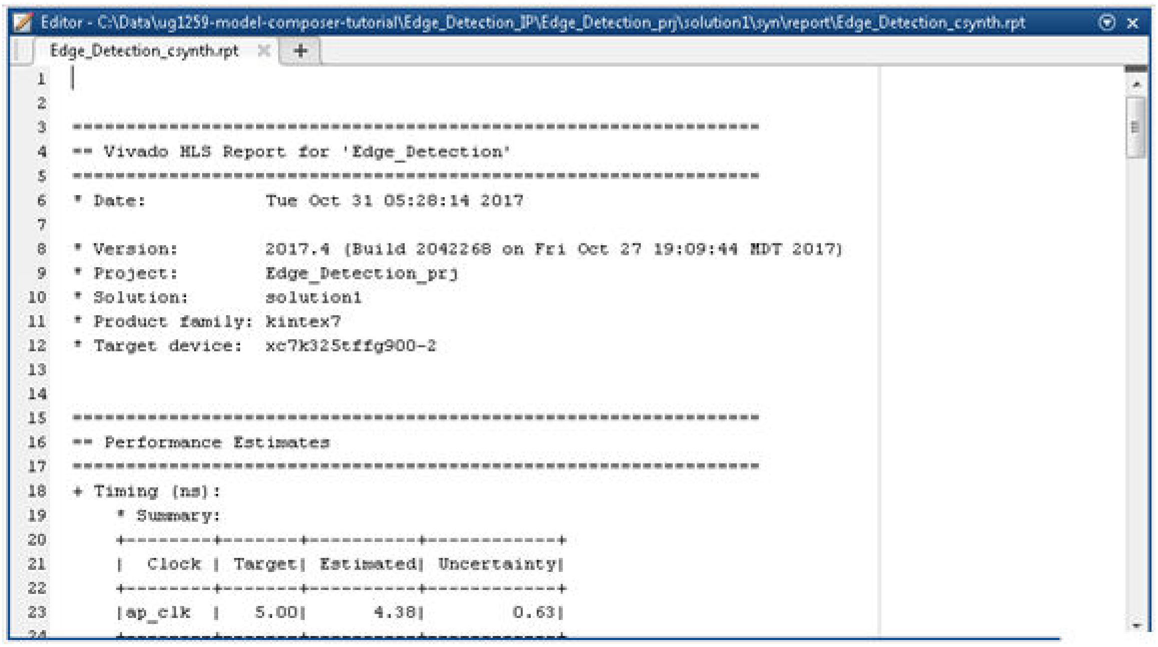
- Scalar ports can be implemented using any of the following protocols: Default, AXI4-Lite Slave, Constant, Valid Port, No protocol.
- Large array or matrix ports should use a streaming protocol such as AXI4-Stream, FIFO, or AXI4-Stream (video).
- Video signals can be transported over an AXI4-Stream (video) interface. In this case you also need to specify the video format YUV 4:2:2, YUV 4:4:4, RGB, or Mono. For video formats that have more than 1 color component, you also need to assign multiple ports to the same signal bundle, and you need to specify which port carries which color component. All of the ports that make up the video signal are implemented by a single AXI4-Stream interface that includes start-of frame and end-of-line sideband signals. For more information refer to *AXI4-Stream Video IP and System Design Guide* (UG934).

Generating Packaged IP for Vivado

Xilinx® Model Composer can automatically generate packaged IP for use in the Vivado® Design Suite IP catalog. When Model Composer generates output for the IP catalog, it first writes the C++ code as described in Generating C++ Code, and then it synthesizes RTL code from the C++ code. This process begins when you set the **Export Type** in the Model Composer Hub block to **IP catalog**, hit **Apply** to confirm any changes, and click **Generate**.

Model Composer displays a transcript window of the process. When the process has concluded, the MATLAB® window displays the Synthesis Report for your review, as shown in the figure below. The Synthesis Report includes details on the estimated performance and resource utilization of the RTL design synthesized by Model Composer. You can review this report to see the estimates and review your model.

Figure 28: **Synthesis Report**


Editor - C:\Data\ug1259-model-composer-tutorial\Edge_Detection_IP\Edge_Detection_prj\solution1\syn\report\Edge_Detection_csynth.rpt
Edge_Detection_csynth.rpt

```


================================================================
== Vivado HLS Report for 'Edge_Detection'
================================================================
* Date:           Tue Oct 31 05:28:14 2017

* Version:        2017.4 (Build 2042268 on Fri Oct 27 19:09:44 MDT 2017)
* Project:        Edge_Detection_prj
* Solution:       solution1
* Product family: kintex7
* Target device:  xc7k325tffg900-2


================================================================
== Performance Estimates
================================================================
+ Timing (ns):
    * Summary:
    +--------+-------+----------+------------+
    |  Clock | Target| Estimated| Uncertainty|
    +--------+-------+----------+------------+
    |ap_clk  |   5.00|      4.38|        0.63|
```

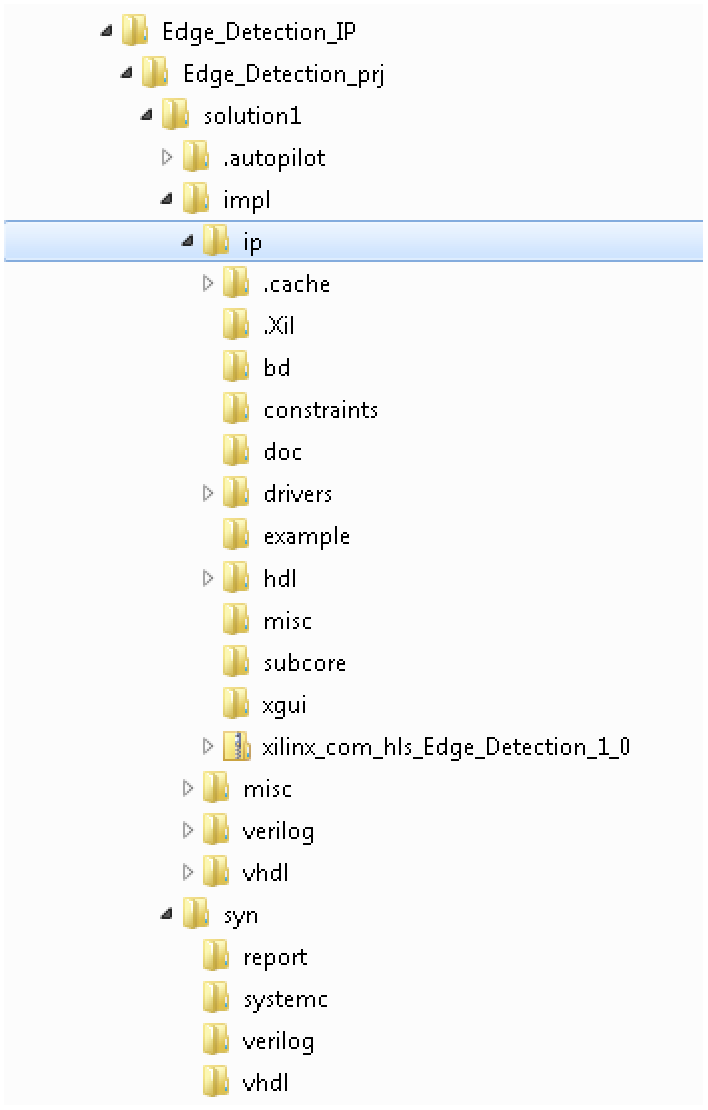
When Model Composer has completed synthesizing the RTL, it reports the message `Exporting RTL as a Vivado IP` to the transcript window, and launches the Vivado Design Suite to create and package the IP for the subsystem design.

Model Composer generates the following outputs from the algorithm:

- SystemC (IEEE 1666-2006, version 2.2)
- VHDL (IEEE 1076-2000)
- Verilog (IEEE 1364-2001)
- Report files created during synthesis, C/RTL co-simulation, and IP packaging.

When Model Composer has completed generating the packaged IP, it can be found in the project directory structure as shown in the following figure. The `Edge_Detection_IP` folder is the **Target Directory** specified by the Model Composer Hub. The `Edge_Detection_prj` folder is a project created by the `run_hls.tcl` script. The `solution1` folder is a Vivado HLS solution. For more information refer to the *Vivado Design Suite User Guide: High-Level Synthesis* (UG902). The `syn` and `impl` folders store the results of synthesis and implementation. The `ip` folder contains the packaged IP to add to the Vivado Design Suite IP catalog.

Figure 29: **Packaged IP Folder**



```
Edge_Detection_IP
  Edge_Detection_prj
    solution1
      .autopilot
      impl
        ip
          .cache
          .Xil
          bd
          constraints
          doc
          drivers
          example
          hdl
          misc
          subcore
          xgui
          xilinx_com_hls_Edge_Detection_1_0
        misc
        verilog
        vhdl
      syn
        report
        systemc
        verilog
        vhdl
```

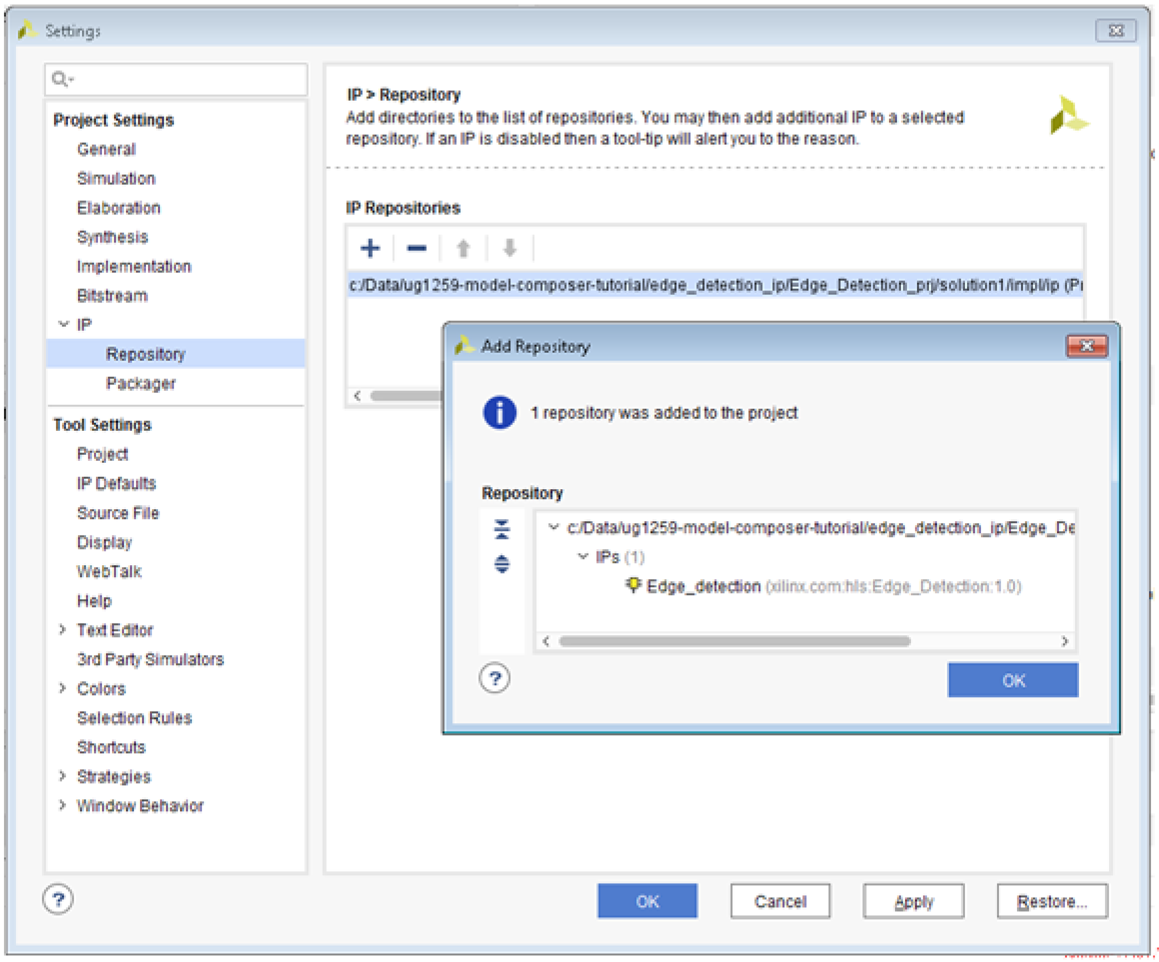
After Model Composer has generated the packaged IP, the `.zip` file archive in the `<project_name>/<solution_name>/impl/ip` folder can be imported into the Vivado IP catalog, and used in any Vivado Design Suite design, either as RTL IP, or in the IP integrator.

IMPORTANT! *If your model uses an RTL IP-based block, such as the FFT, IFFT, or FIR Filter, the output generated by Model Composer will be written to a different directory. Refer to RTL IP-Based Blocks for more information.*

For Model Composer models that specify AXI4-Lite Slave interfaces through the Interface Specification block, as discussed in Defining the Interface Specification, a set of software driver files is also created by Vivado HLS during the IP packaging process. These C driver files can be included in an SDK C project and used to access the AXI4-Lite Slave slave port. The software driver files are written to directory `<project_name>/<solution_name>/impl/ip/drivers` and are included in the packaged IP.

To add the IP into the Vivado IP catalog, from within the Vivado Design Suite GUI, select the **Tools→Settings** command to open the Settings dialog box. Select the **IP→Repository** command, and add the Vivado HLS packaged IP as shown below.

Figure 30: **Setting the IP Repository**



Add directories to the list of repositories. You may then add additional IP to a selected repository. If an IP is disabled then a tool-tip will alert you to the reason.

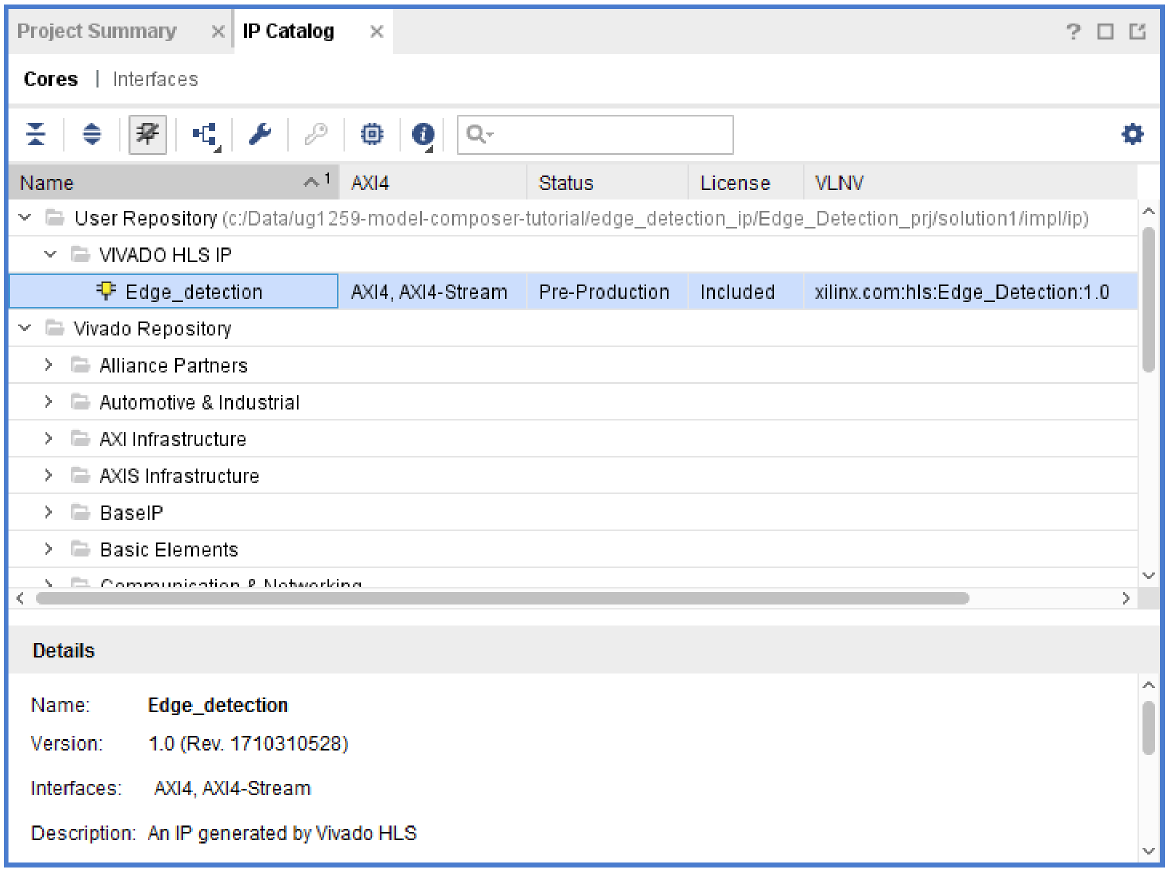
c:/Data/ug1259-model-composer-tutorial/edge_detection_ip/Edge_Detection_prj/solution1/impl/ip (P

1 repository was added to the project

c:/Data/ug1259-model-composer-tutorial/edge_detection_ip/Edge_De
IPs (1)
Edge_detection (xilinx.com:hls:Edge_Detection:1.0)

After adding the path to the repository, the IP is added to the IP catalog as shown in the following figure. You can now use the IP in standard RTL designs, or in Vivado IP integrator block designs. For more information on working with IP and adding to the IP repository refer to the *Vivado Design Suite User Guide: Designing with IP* (UG896).

Figure 31: **IP Catalog**



Name	AXI4	Status	License	VLNV
User Repository (c:/Data/ug1259-model-composer-tutorial/edge_detection_ip/Edge_Detection_prj/solution1/impl/ip)				
VIVADO HLS IP				
Edge_detection	AXI4, AXI4-Stream	Pre-Production	Included	xilinx.com:hls:Edge_Detection:1.0
Vivado Repository				
Alliance Partners				
Automotive & Industrial				
AXI Infrastructure				
AXIS Infrastructure				
BaseIP				
Basic Elements				
Communication & Networking				

Name: Edge_detection
Version: 1.0 (Rev. 1710310528)
Interfaces: AXI4, AXI4-Stream
Description: An IP generated by Vivado HLS

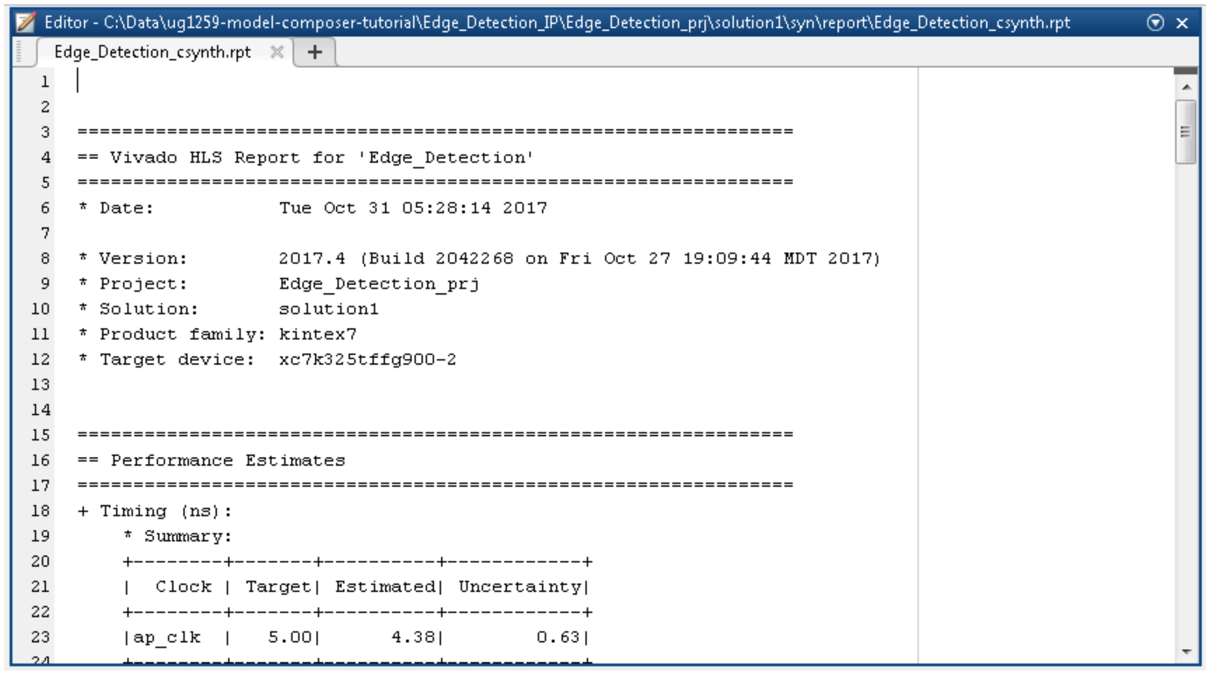
IMPORTANT! *If you see the repository added to the IP catalog, but do not see the Vivado HLS packaged IP, it is possible the target part for the current project is not compatible with the device used when generating the Model Composer output. You can fix this by changing the part in the current project to the device specified by the Model Composer model.*

Generating System Generator IP

Xilinx Model Composer can automatically generate a System Generator for DSP solution to import as a block into System Generator. When Model Composer generates output for System Generator, it first writes the C++ code as described in Generating C++ Code, and then it synthesizes RTL code from the C++ code. This process begins when you set the **Export Type** in the Model Composer Hub block to **System Generator**. This command creates an IP package for System Generator.

Model Composer displays a transcript window of the process. When the process has concluded, the MATLAB window displays the Synthesis Report for your review, as shown in the figure below. The Synthesis Report includes details on the estimated performance and resource utilization of the RTL design synthesized by Model Composer. You can review this report to see the estimates and review your model.

Figure 32: **Synthesis Report**


Editor - C:\Data\ug1259-model-composer-tutorial\Edge_Detection_IP\Edge_Detection_prj\solution1\syn\report\Edge_Detection_csynth.rpt
Edge_Detection_csynth.rpt

```


================================================================
== Vivado HLS Report for 'Edge_Detection'
================================================================
* Date:           Tue Oct 31 05:28:14 2017

* Version:        2017.4 (Build 2042268 on Fri Oct 27 19:09:44 MDT 2017)
* Project:        Edge_Detection_prj
* Solution:       solution1
* Product family: kintex7
* Target device:  xc7k325tffg900-2


================================================================
== Performance Estimates
================================================================
+ Timing (ns):
    * Summary:
    +--------+-------+----------+------------+
    |  Clock | Target| Estimated| Uncertainty|
    +--------+-------+----------+------------+
    |ap_clk  |   5.00|      4.38|        0.63|
    +--------+-------+----------+------------+
```

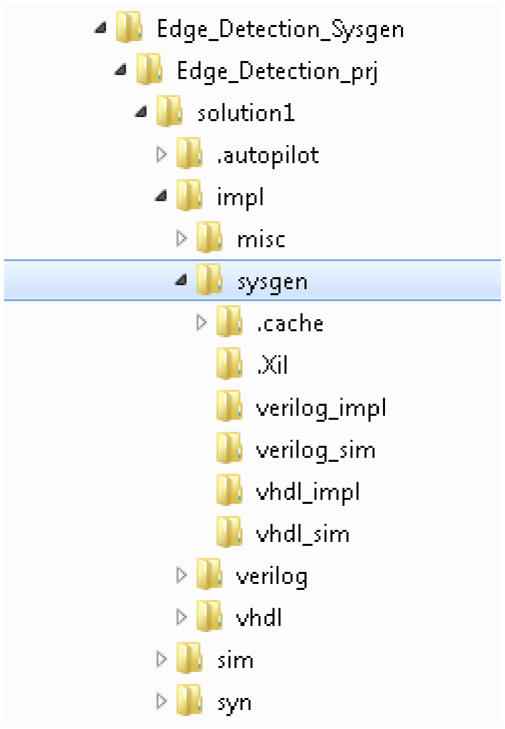
When Model Composer has completed synthesizing the RTL, it reports the message `Exporting RTL as an IP for System Generator for DSP` to the transcript window. This process is handled by a Tcl script, `run_hls.tcl`, that Model Composer writes to export the System Generator IP.

Model Composer generates the following outputs from the algorithm:

- SystemC (IEEE 1666-2006, version 2.2)
- VHDL (IEEE 1076-2000)
- Verilog (IEEE 1364-2001)
- Report files created during synthesis, C/RTL co-simulation, and IP packaging.

When Model Composer has generated the System Generator IP, you can find it in the project directory structure as shown in the following figure. The `Edge_Detection_Sysgen` folder is the **Target Directory** specified by the Model Composer Hub. The `Edge_Detection_prj` folder is a project created by the `run_hls.tcl` script. The `solution1` folder is a Vivado HLS solution. For more information refer to the *Vivado Design Suite User Guide: High-Level Synthesis* (UG902). The `syn`, `sim`, and `impl` folders store the results of synthesis, simulation, and implementation. The `sysgen` folder contains the System Generator files needed to use the subsystem IP.

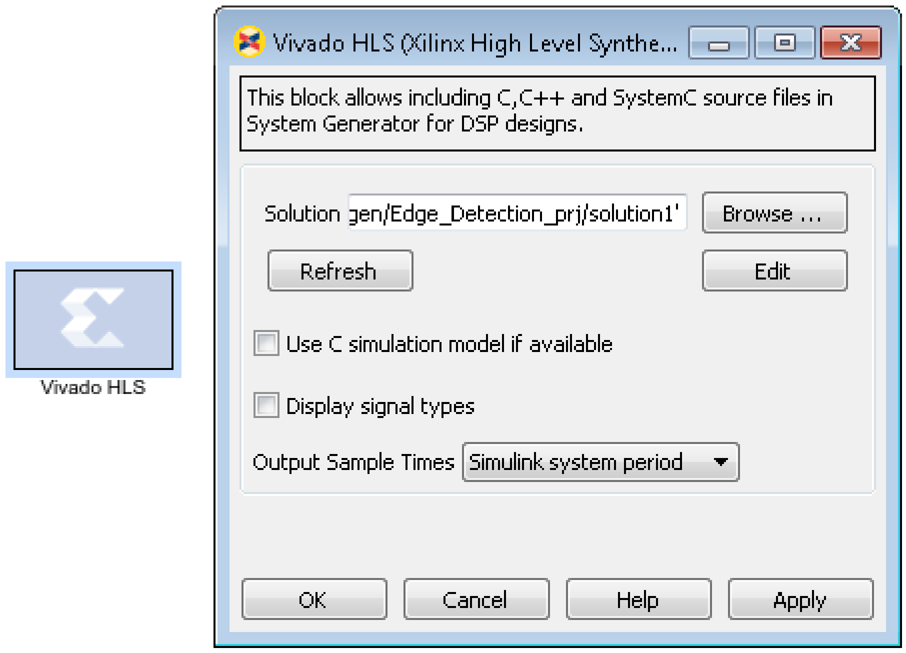
Figure 33: **System Generator Output**



You can import a Model Composer generated System Generator IP into a System Generator model using the following steps:

1. From within an open System Generator model, right-click in the canvas of the Simulink Editor and select the **Xilinx BlockAdd** command. This opens a menu of Xilinx System Generator blocks that can be added to your model.
2. Scroll down the list in dialog box, or type "HLS" in the **Add Block** search field to locate the Vivado HLS block and add it to your model.
3. Double-click on the newly added block to open the **Vivado HLS Block Parameter** dialog box as shown below.

Figure 34: **Vivado HLS Block**


Vivado HLS (Xilinx High Level Synthe...

This block allows including C,C++ and SystemC source files in System Generator for DSP designs.

Solution gen/Edge_Detection_prj/solution1'

☐ Use C simulation model if available

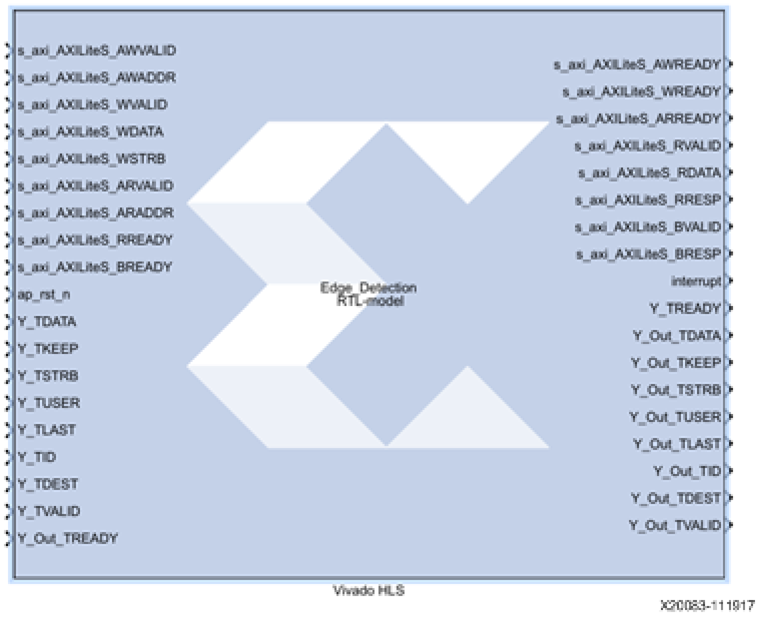
☐ Display signal types

Output Sample Times Simulink system period

4. **Browse** to the solution directory of the Vivado HLS project where the Model Composer output was generated. In the Edge Detection example above, browse to the `../Edge_Detection_prj/solution1` folder and select **OK**.

The Vivado HLS template block is converted to the Edge Detection IP in the System Generator model. You may need to drag the corners of the IP block to expand it as needed for your model. The block is initially sized to match the Vivado HLS template. The figure below shows the System Generator IP generated from the Model Composer model.

Figure 35: **Vivado HLS Block**



If any of the function arguments on the Model Composer subsystem module are transformed by Vivado HLS into a composite port, the `signal type` information for that port cannot be determined and included in the System Generator IP block. Any design that uses the reshape, mapping, or data packing optimization on ports must have the port type information manually specified in System Generator for these composite ports. You should know how the composite ports were originally created and then use `slice` and `reinterpretation` blocks in System Generator to connect the Vivado HLS block to other blocks in the system.

For example, if three 8-bit in-out ports R, G and B are packed into a 24-bit input port (`RGB_in`) and a 24-bit output port (`RGB_out`) ports. After the IP block has been included in System Generator:

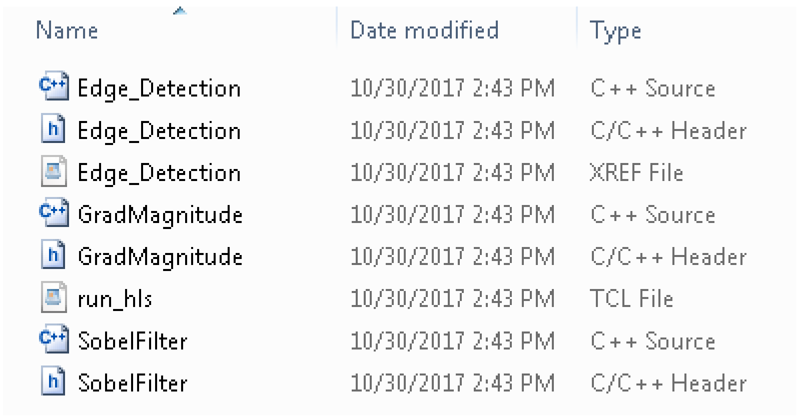
- The 24-bit input port (`RGB_in`) would need to be driven by a System Generator block that correctly groups three 8-bit input signals (`R_in`, `G_in` and `B_in`) into a 24-bit input bus.
- The 24-bit output bus (`RGB_out`) would need to be correctly split into three 8-bit signals (`R_out`, `G_out`, and `B_out`).

TIP: *See the System Generator documentation for details on using the slice and reinterpretation blocks to connect to composite type ports.*

Generating C++ Code

The following figure shows the C++ Code output by Model Composer from the **Generate** command. The C++ code is output either as an intermediate step when generating a packaged IP or System Generator output, or as a specified output to let you optimize the C++ code using directives or pragmas in Vivado® HLS.

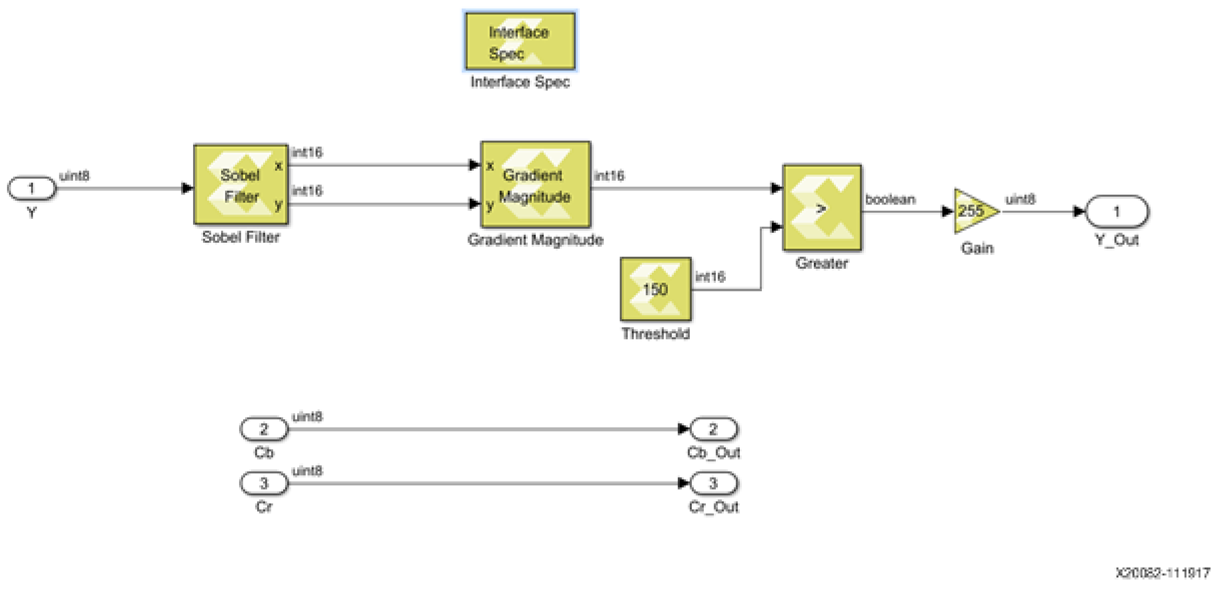
Figure 36: **C++ Output Files**



Name	Date modified	Type
Edge_Detection	10/30/2017 2:43 PM	C++ Source
Edge_Detection	10/30/2017 2:43 PM	C/C++ Header
Edge_Detection	10/30/2017 2:43 PM	XREF File
GradMagnitude	10/30/2017 2:43 PM	C++ Source
GradMagnitude	10/30/2017 2:43 PM	C/C++ Header
run_hls	10/30/2017 2:43 PM	TCL File
SobelFilter	10/30/2017 2:43 PM	C++ Source
SobelFilter	10/30/2017 2:43 PM	C/C++ Header

The files generated by Model Composer reflect the contents and hierarchy of the subsystem that was compiled. In this case, the subsystem is the Edge Detection function described in the *Model Composer Tutorial: Model-Based Design Using Model Composer* (UG1259) tutorial designs. The figure below shows the contents of the Edge Detection subsystem.

Figure 37: **Edge Detection Subsystem**



The `Edge_Detection.cpp` file specifies the following include files, which incorporate the code generated for the various Model Composer blocks used in the subsystem:

```
#include "Edge_Detection.h"
#include "GradMagnitude.h"
#include "SobelFilter.h"
```

The following shows the generated code for the Edge Detection subsystem. Notice the pragmas added to the function to specify the function protocol and the I/O port protocols for the function signature and return value. The pragmas help direct the solution synthesized by Vivado HLS, and result in higher performance in the implemented RTL.

```
Edge_Detection(hls::stream< ap_axiu<16, 1, 1, 1> >& Y,
    hls::stream< ap_axiu<16, 1, 1, 1> >& Y_Out)
{
    #pragma HLS INTERFACE s_axilite port=return
    #pragma HLS INTERFACE axis bundle=image_out port=Y_Out
    #pragma HLS INTERFACE axis bundle=input_vid port=Y
    #pragma HLS dataflow
    uint8_t core_Y[360][640];
    #pragma HLS stream variable=core_Y dim=2 depth=1
    uint8_t core_Cb[360][320];
    #pragma HLS stream variable=core_Cb dim=2 depth=1
    uint8_t core_Cr[360][320];
    #pragma HLS stream variable=core_Cr dim=2 depth=1
    uint8_t core_Y_Out[360][640];
    #pragma HLS stream variable=core_Y_Out dim=2 depth=1
    uint8_t core_Cb_Out[360][320];
    #pragma HLS stream variable=core_Cb_Out dim=2 depth=1
    uint8_t core_Cr_Out[360][320];
    #pragma HLS stream variable=core_Cr_Out dim=2 depth=1
    fourier::AxiVideoStreamAdapter< uint8_t >::readStreamVf0(Y,
        reinterpret_cast< uint8_t* >(core_Y), reinterpret_cast< uint8_t* >(
        core_Cb), reinterpret_cast< uint8_t* >(core_Cr), 360, 640);
    Edge_Detection_core(core_Y, core_Cb, core_Cr, core_Y_Out, core_Cb_Out,
        core_Cr_Out);
    fourier::AxiVideoStreamAdapter< uint8_t >::writeStreamVf0(Y_Out,
        reinterpret_cast< uint8_t* >(core_Y_Out), reinterpret_cast<
uint8_t* >(
        core_Cb_Out), reinterpret_cast< uint8_t* >(core_Cr_Out), 360, 640);
}
```

Finally, notice the `run_hls.tcl` file that is generated in the output folder. This is a Tcl script that can be used to run Vivado HLS on the generated output files to create a project and solution, synthesize the RTL from the C++ code, and export the design to System Generator. Each Vivado HLS project holds one set of C/C++ code and can contain multiple solutions. Each solution can have different constraints and optimization directives. For more information refer to the *Vivado Design Suite User Guide: High-Level Synthesis* (UG902).

You can run the `run_hls.tcl` script from a Vivado HLS command prompt as follows:

1. Open the Vivado HLS Command Prompt:

    - On Windows, use **Start → All Programs → Xilinx Design Tools → Vivado 2020.x → Vivado HLS → Vivado HLS 2020.x Command Prompt**.

- On Linux, open a new shell and source the `<install_dir>/Vivado/<version>/settings64.sh` script to configure the shell.

2. From the command prompt, change the directory to the parent folder of the **Target Directory** specified on the Model Composer Hub dialog box when you generated the output, as discussed at Adding the Model Composer Hub. For example:

   ```
   cd C:/Data
   ```

3. From the command prompt, launch the `run_hls.tcl` script that can be found in the **C**:

   ```
   vivado_hls -f ./code/run_hls.tcl
   ```

Vivado HLS launches to synthesize the RTL from the C++ code, generating a Vivado HLS project, and solution in the process. You can open the Vivado HLS project by going to the **Target directory** and entering the following name with the project name:

```
vivado_hls -p ./Edge_Detection_proj
```

This will open the Vivado HLS project in the GUI Mode.

Model Composer Log File

To help you diagnose issues related to code generation, Model Composer generates a log file, `model_composer.log`, that is written to the **Target Directory** specified on the Model Composer Hub block.

By default, Model Composer generates code for the model using a streaming micro-architecture in which blocks run concurrently via task pipelining, or dataflow. However, in some cases this streaming micro-architecture is not achievable, because the model includes an `xmcImportFunction` block that does not support streaming for example. In this case, Model Composer generates code in which the blocks operate sequentially. The Model Composer log file includes information to help you identify when this condition occurs, and what some possible causes might be.

The log file also contains information related to which blocks are used in the Model Composer model.

If **Export Type** on the Model Composer Hub block is set to C++ code, and **Create and execute testbench** is selected, the `model_composer.log` file will contain the output from running the C++ verification.

If **Export Type** is set to **IP Catalog** or **System Generator**, information related to running Vivado HLS is provided. In these cases, more detailed information can be found in the `vivado_hls.log` file which can also be found in the **Target Directory**.

Chapter 5

Simulating and Verifying Your Design

Introduction

Verification in Model Composer can be separated into two distinct processes:

- Verification of the algorithm in Simulink® to verify the functional correctness of the design.
- Verification of the Model Composer model, to confirm the equivalence of the simulation results in Simulink and the C++ and RTL outputs.

In high-level synthesis, running the compiled C program is referred to as C simulation. Executing the C++ algorithm generated by Model Composer simulates the function and verifies that the output from the code matches the output from the Simulink simulation to validate that the algorithm is functionally correct.

In C/RTL co-simulation, Vivado® HLS uses the C test bench to simulate the C function prior to synthesis and to verify the RTL output. The verification process consists of three phases:

1. The C simulation is executed and the inputs to the top-level subsystem are saved as "input vectors".
2. The input vectors are used in behavioral simulation of the RTL code for the top-level subsystem created by Vivado HLS. The outputs from the RTL are saved as "output vectors".
3. The output vectors are applied to the C test bench as output from the top-level subsystem, to verify the results of the C-simulation match.

Vivado HLS uses this return value for both C simulation and C/RTL co-simulation to determine if the results are correct.

Simulating in Simulink

Simulink can interactively simulate your model, and view the results on scopes and graphical displays. The Simulink model defines what data to input at the start of simulation, and defines what data to capture during simulation.

When defining the model, you define the input and output signals for the model. Input signals load data into the model for simulation, while output signals allow you to record simulation results. The kind of data you want to load impacts your choice of signal loading techniques. You can define input data as constant values, use source blocks, such as the Sine Wave block, import data from a spreadsheet, or use the output of a previous simulation. Refer to Prepare Model Inputs and Outputs in the Simulink documentation for more information on preparing for simulation in Simulink.

After the simulation is completed, you can analyze any logged data with MATLAB scripts and visualization tools like the Simulation Data Inspector within the MathWorks environment.

When you enable the **Create and execute testbench** checkbox, as discussed in Adding the Model Composer Hub, Model Composer performs two tasks:

1. Automatically logs the test data, or stimulus, at the input of your design, and the simulation results as test vectors for later use as "golden" data for comparison during C/C++ and RTL co-simulation. This file is named `signals.stim`, and is added to the specified **Target Directory** when generating output.
2. Executes the generated test bench and verifies equivalence of the code using C-simulation and C/RTL co-simulation. This process is compute intensive, and can take considerable time.

IMPORTANT! *If the model simulation time in Simulink is long, with significant amounts of data processed, the test bench execution will take an even longer time. Model Composer will generate an error if the simulation time becomes infinite.*

Managing the Model Composer Cache

Model Composer creates a cache entry when you simulate a model with an imported block created using `xmcImportFunction`. When you simulate the model again, unless you make any changes to the function, or supporting source files, Model Composer will use the cached entry for the block to initiate the simulation faster. You can manage the simulation cache in Model Composer using the following command from the MATLAB command prompt:

```
>> xmcSimCache
```

The usage for this command is as follows.

Table 9: **xmcSimCache Command Usage**

Command	Description
`xmcSimCache('enable')`	Enable the simulation cache. This is enabled by default.
`xmcSimCache('disable')`	Disable the simulation cache.
`xmcSimCache('isEnabled')`	Returns the state of the simulation cache.
`xmcSimCache('setLocation', <dir>)`	Specify a directory to use for the simulation cache. For example: `xmcSimCache('setLocation', 'C:/temp')`
`xmcSimCache('setDefaultLocation')`	Restore the simulation cache to its default location.
`xmcSimCache('getLocation')`	Return the current location of the simulation cache.
`xmcSimCache('clear')`	Clear the entire simulation cache.
`xmcSimCache('clear', 'release', <version>)`	Clear simulation cache entries for the specified release version. For example: `xmcSimCache('clear', 'release', '2017.4')`
`xmcSimCache('clear', 'days', <number>)`	Clear simulation cache entries older than or equal to the specified number of days. For example: `xmcSimCache('clear', 'days', 30)`
`xmcSimCache('clear', 'id', <vals>)`	Clear simulation cache entries by the cache ID. For example: `xmcSimCache('clear', 'id', {'12345678', 'abcdefgh'})`

Verifying the C++ Code

When generating the output using the Model Composer Hub block, you also have the ability to **Create and execute testbench**. When selecting this option, you are enabling the Model Composer verification flow. This causes Model Composer to generate a few more output files, including a makefile, the test bench, `tb.cpp`, and `signals.stim` as previously discussed. The purpose of the test bench is to apply input stimuli, generated during Simulink simulation, to the top-level function of the generated C++ or RTL code and compare that function's output against the output samples captured in the `signals.stim` file. Depending on the output generated, the verification flow runs simulation on the C++ or RTL outputs generated by Model Composer and looks for the same result as generated by Simulink.

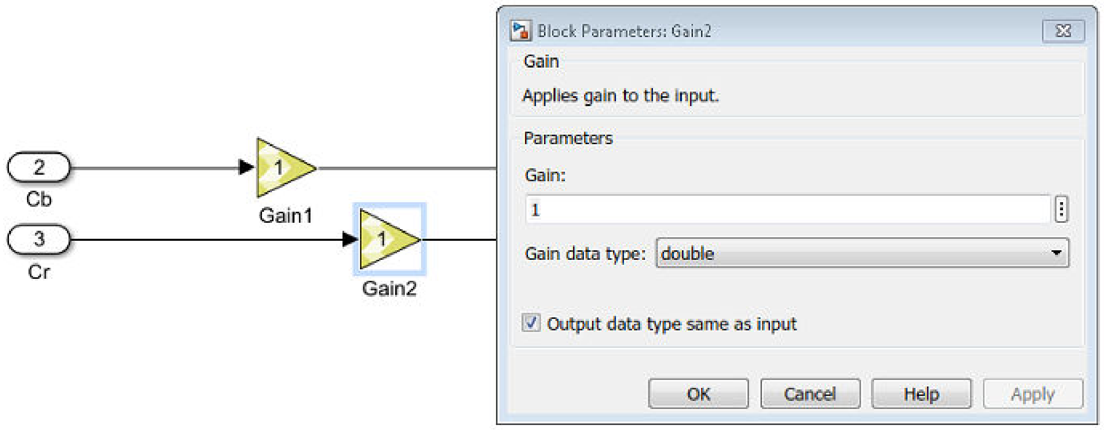
When the **Export type** on the Model Composer Hub block is C++ code, the verification flow is as follows:

- The model is simulated in Simulink and the input and outputs are logged into the `signals.stim` binary file.

- Model Composer generates the C++ code and a test bench,`tb.cpp`, which contains a `main()` function.
- Model Composer launches simulation.
- It verifies that the output from the generated C++ code matches the output logged from the Simulink simulation, `signals.stim`.
- In case of a mismatch, the mismatched output signal name is reported, as well as the actual and expected values.
- The result is a Pass/Fail returned by Model Composer.

IMPORTANT! *During simulation by Model Composer, you may receive the following error message:* `Failed to find a XilinxLibrary block connected to input port.` *This error simply means that there is an input in the subsystem that does not connect to a block from the Xilinx Model Composer block set. This may be due to the presence of signals that you are simply passing through the subsystem for signal grouping, or improved readability. The recommended fix for this conditions is to connect the input to a Gain block from the* ***Xilinx Model Composer → Math Functions → Math Operations*** *block library, with the default value of 1, and* ***Output data type same as input*** *checked, as shown in the following figure.*

Figure 38: **Adding Gain to Unconnected Inputs**



Gain

Applies gain to the input.

Parameters

Gain:

1

Gain data type: double

☑ Output data type same as input

Verifying the C/RTL Code

When the **Export Type** specified on the Model Composer Hub dialog box is either IP catalog or System Generator, and the verification flow is enabled, Model Composer uses C/RTL co-simulation to verify the RTL output. Again, the objective is to verify that the results of the RTL simulation match the results of the Simulink simulation. In this case, the verification flow is as follows:

- The Model Composer model is simulated in Simulink to capture the test vectors in `signals.stim`.

- Model Composer generates the C/C++ code and the C test bench, `tb.cpp`.
- Model Composer runs the C-synthesis and generates the RTL output.
- Model Composer runs the C/RTL co-simulation. This step ensures
    - That the C++ code generated by Model Composer is correct by comparing with the Simulink simulation, `signals.stim`.
    - That the RTL code generated by Model Composer is correct by comparing the output stimulus from RTL with C/C++ output.

TIP: *After verification, Model Composer exports the RTL as an IP for System Generator, or packages the IP for use in the Vivado Design Suite.*

- The result is a separate Pass/Fail returned by Model Composer for both the C-simulation and the C/RTL co-simulation. If the C-simulation fails, the process stops before the C/RTL simulation is run.

Appendix A

Select Target Device or Board

Device Chooser Dialog Box

Choose the default part or platform board to use for the current project. The device resources that the design is synthesized against, and placed onto are determined by selecting the target part or board.

- **Parts tab:** Lists the available target parts for the current project.
- **Boards tab:** Lists the available target boards for the current project.

IMPORTANT! *The devices and boards that are available are determined at the time the Model Composer tool is installed. You can also add uninstalled devices or boards. To learn more refer to Xilinx® Answer Record 60112 at https://www.xilinx.com/support/answers/60112.html.*

The selection of target part or boards can be limited or filtered by specifying search patterns in one or more of the available search fields at the top of the Device Chooser dialog box.

For parts:

- **Category:** Choose devices according to Military, Automobile, or Commercial grade products.
- **Family:** Filter devices according to the available device families (such as Virtex, Kintex, or UltraScale).
- **Package:** Specify the type of package the device will have.
- **Speed:** Filter the device by a specific speed grade.
- **Temperature:** Filter the device by a specific temperature.

For boards:

- **Vendor:** Choose devices according to the available vendors of platform boards.
- **Name:** Filter devices according to the available display names.
- **Board Rev:** Specify the revision level of the board.

The target parts or boards that match the specified filters and/or search string appear in a table of results in the lower portion of the dialog box.

Choose a target part or board for the design, and click **OK**.

Appendix B

Model Composer Block Library

Supported Simulink Blocks

You can mix blocks from the Model Composer block library and blocks from Simulink®, and other add-on toolboxes, during simulation in Simulink. However, you can only generate output from Model Composer, as described in Chapter 4: Generating Outputs, from the top-level subsystem that uses only a limited set of Simulink blocks that are supported for code generation.

The following Simulink blocks are fully supported by Model Composer for code generation, and can be found in the Xilinx Model Composer block library as well.

Table 10: **Supported Simulink Blocks**

Simulink Block	Description
Action Port	Place this block in a subsystem to link to a signal from an `If` block or a `Switch-Case` block.
Bus Creator	This block creates a bus signal from multiple inputs.
Bus Selector	Pulls signals from an input bus to pass to output.
Display	Provides numeric display of input values.
DocBlock	Create and edit text associated with a model, and save that text with the model.
From	Receive signals from the Goto block with the specified tag.
Goto	Send signals to From blocks that have the specified tag.
If	Provides an `IF`/`ELSEIF`/`ELSE` condition for branching inputs to alternate outputs.
Inport	Provide an input port for a subsystem or model.
Merge	Merge multiple input signals into a single output signal whose initial value is specified by the 'Initial output' parameter.
Outport	Provide an output port for a subsystem or model.
Scope	Displays time domain signals with respect to simulation time.
Stop Simulation	Stop simulation when input is non-zero.
Terminator	Used to "terminate" output signals to prevent warnings about unconnected output ports.
To File	Incrementally write data into a variable in the specified MAT-file.
To Workspace	Write input to specified timeseries, array, or structure in a workspace.

Refer to the Simulink documentation for a complete description of the block.

Model Composer Block Taxonomy

Table 11: **Computer Vision**

Block	Description
2-D Convolution	Perform a two dimensional convolution of an image with a user-defined kernel
Bilateral Filter	Apply a Bilateral Filter function to an unsigned 8-bit integer greyscale image
Dense Non-Pyramidal LK Optical Flow	Compute the optical flow between two adjacent frames using the Lucas-Kanade method. The outputs are horizontal (X) and vertical (Y) components of the flow vectors.
Dilation	Morphological function to replace each element of the input signal with the maximum intensity value in its 3x3 neighborhood. The function accepts and produces grayscale images (2-D matrices of real uint8 data type).
Erosion	Erode away high intensity pixels from the regions where high intensity and low intensity pixels form the boundaries in the input image.
FAST Corner Detection	Apply the FAST corner detection algorithm to the input image and produce a matrix of image coordinates of candidate points where the corners are detected.
Gaussian Blur	Apply a Gaussian blur function to an unsigned 8-bit integer grey scale image
Gradient Magnitude	Compute the element-wise magnitudes of an array of vectors given their orthonormal basis components (x,y), based on either Manhattan distance (L1-Norm) or Euclidean distance (L2-Norm)
Gradient Phase	Compute the element-wise phase angle of an array of vectors given their orthonormal basis components (x,y)
Histogram	Compute the histogram for 8-bit greyscale images
Histogram Equalization	Improve the contrast of an image supplied on the input ports by adjusting intensity values such that the resulting image has more uniform distribution of the intensity values.
Integral Image	Compute the integral image of the input
Mean & Std Deviation	Compute the mean and standard deviation of the input data
Median Blur Filter	Apply a 3x3 median filter to the input
MinMax Location	Determine the value and the location of the minimum and maximum of the input image
Otsu Thresholding	Find threshold intensity value for the input greyscale image (uint8 data type) such that pixel intensity levels on each side of the threshold have the minimum spread.
Pyramid Down	Downsample an unsigned 8-bit integer grayscale image to half its size
Pyramid Up	Upsamples an unsigned 8-bit integer greyscale image to twice its size
Remap	Take pixels from one place in the input image and relocate them to another position in the output image
Resize	Resize an image to specified dimensions
Sobel Filter	Compute the gradients of an input image in both x and y direction by convolving the input image with a 3x3, 5x5, or 7x7 filter kernel
Warp Transform	Perform either an affine or a perspective transform of an 8-bit greyscale image
Window Processing	Assemble an output matrix by applying the kernel subsystem to submatrices (windows) of the input matrix in row-major order

Table 12: **Logic and Bit Operations**

Block	Description
Bit Concat	Perform bitwise concatenation of input values into a single output value
Bit Slice	Extract a range of bits from a value
Bitwise AND	Perform element and bitwise Boolean AND operation on the inputs
Bitwise NOT	Perform element and bitwise Boolean NOT operation on the input
Bitwise OR	Perform element and bitwise Boolean OR operation on the inputs
Bitwise XOR	Perform element and bitwise Boolean XOR operation on the inputs
Logical AND	Performs element-wise logical AND operation on inputs
Logical NOT	Performs element-wise logical NOT operation on the input
Logical OR	Performs element-wise logical OR operation on inputs
Reduction AND	Compute bitwise AND of the elements of the input over all dimensions or over a specified dimension
Reduction OR	Compute bitwise OR of the elements of the input over all dimensions or over a specified dimension
Reduction XOR	Compute bitwise XOR of the elements of the input over all dimensions or over a specified dimension
Shift Left	Perform logical shift left of input over a constant number of bit positions specified by a non-negative integer mask parameter
Shift Right	Perform logical shift right of input over a constant number of bit positions specified by a non-negative integer mask parameter

Table 13: **Lookup Tables**

Block	Description
Lookup Table	Perform one-dimensional lookup operation with an input index

Table 14: **Math Functions / Math Operations**

Block	Description
Abs	Compute element-wise absolute value of input signal
atan	Compute element-wise inverse tangent of input signal
atan2	Compute element-wise four-quadrant inverse tangent of input signal
Complex to Polar	Element-wise conversion of complex input signals into magnitude and radiant phase angle
Complex to Real-Imag	Output real and imaginary parts of complex input signal
Conjugate	Apply element-wise complex conjugate operation to the input signal
Cosine	Element-wise computation of the cosine function for a given argument
cosh	Element-wise computation of the hyperbolic cosine for a given argument
Cumulative Sum	Compute the cumulative sum along the specified dimension of the input
Divide	Perform element-wise division
Exp	Perform an element-wise exponential value of the input
Gain	Multiply the input signal with a constant gain factor.
Log	Compute element-wise natural logarithm of input

Table 14: **Math Functions / Math Operations** *(cont'd)*

Block	Description
Log10	Compute element wise base 10 logarithm of input
Modulus	Perform element-wise modulus operation on the input signals
Negate	Perform element-wise unary minus operation on the input data
Polar to Complex	Element-wise conversion of real magnitude and angle representation signals into a complex signal
Pow	Compute the element-wise power function
Product	Compute element-wise product of the input signals
Product of Elements	Multiply the elements of the input signal
Real-Imag to Complex	Convert real and/or imaginary inputs to complex signal
Reciprocal	Perform element-wise computation of the reciprocal for a given argument
Reciprocal Sqrt	Perform element-wise computation of the reciprocal square root for a given argument
Remainder	Perform element-wise division on the input signal, and the output is the remainder after the division
Signum	Perform an element-wise signum function (sign extraction)
Sine	Element-wise computation of the sine function for the given input
sinh	Element-wise computation of the hyperbolic sine for a given argument
Sqrt	Element-wise computation of the square root for a given argument
Subtract	Perform an element-wise subtraction
Sum	Performs element-wise addition of two input signals
Sum of Elements	Perform element-wise addition on the input, column-wise, row-wise, or in all-dimensions
Tangent	Perform an element-wise computation of the tangent function for the given argument

Table 15: **Math Functions / Matrices and Linear Algebra**

Block	Description
Hermitian	Perform element-wise conjugate transpose operation on the input signal
Matrix Multiply	Compute matrix product of two input signals
QR Inverse	Compute the inverse of a matrix using QR factorization
Submatrix	Select a subset of elements (submatrix) from matrix input
Transpose	Perform an element-wise transpose operation on the input signal

Table 16: **Ports and Subsystems**

Block	Description
If	Model `if-else` control flow
In1	Create input port for subsystem or external input
Out1	Create output port for subsystem or external output

Table 17: **Ports and Subsystems / If Action Subsystem**

Block	Description
Action Port	Implement Action subsystems used in `if` and `switch` control flow statements
In1	Create input port for external input
Out1	Create output port for subsystem or external output

Table 18: **Ports and Subsystems / Subsystem**

Block	Description
In1	Create input port for subsystem
Out1	Provide an output port for a subsystem or model

Table 19: **Relational Operations**

Block	Description
Equals	Perform element-wise equal to relational operation on the inputs
Greater	Perform element-wise greater than relational operation on the inputs
Greater Equals	Perform element-wise greater than or equal relational operation on the inputs
Lesser	Perform element-wise less than relational operation on the inputs
Lesser Equals	Perform element-wise less than or equal relational operation on the inputs
Not Equals	Perform element-wise not equal to relational operation on the inputs

Table 20: **Signal Attributes**

Block	Description
Data Type Conversion	Convert the input to the data type of the output
Reinterpret	Element-wise reinterpretation of the input type into a compatible output type with the same bit width

Table 21: **Signal Operations**

Block	Description
Delay	Delay input signal by specified number of samples
Unit Delay	Provides a delay of one sample period

Table 22: **Signal Routing**

Block	Description
Bus Creator	Create Signal Bus
Bus Selector	Select signals from incoming bus
Conditional	Pass through input T when control input satisfies a selected criterion; otherwise, pass through input F
From	Accept input from Goto block

Table 22: **Signal Routing** *(cont'd)*

Block	Description
Goto	Pass block input to From blocks
Merge	Combine multiple signals into single signal

Table 23: **Sinks**

Block	Description
Display	Show value of input
Scope	Display signals generated during simulation
Stop Simulation	Stop simulation when input is nonzero
Terminator	Terminate unconnected output port
To File	Write data to file
To Workspace	Write data to workspace

Table 24: **Source**

Block	Description
Constant	Generates the constant specified by the **Constant Value** parameter

Table 25: **Tools**

Block	Description
DocBlock	Create text that documents model and save text with model
Interface Spec	Specify the RTL interfaces for a subsystem
Model Composer Hub	Control implementation of the model

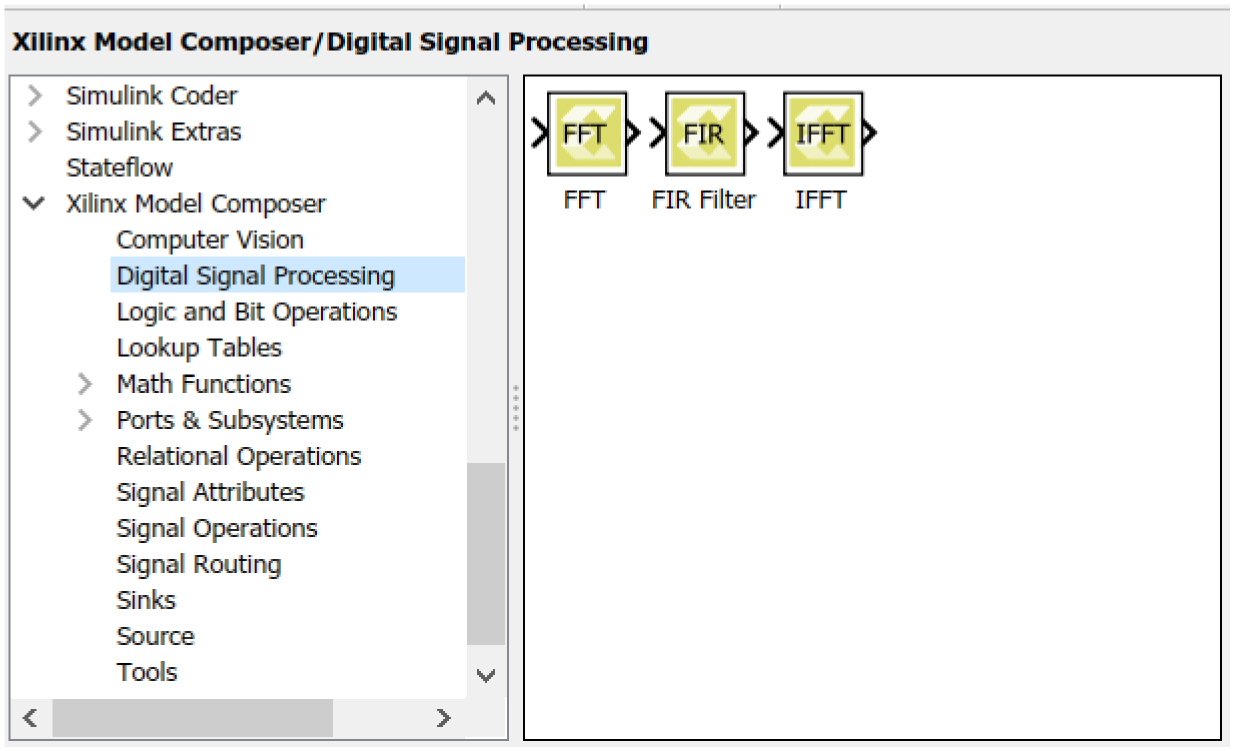
RTL IP-Based Blocks

Introduction

Model Composer provides blocks which are based on RTL IP from the Xilinx® IP Catalog. Model Composer currently supports three RTL IP blocks widely used in Digital Signal Processing applications: FIR Filter, FFT, and IFFT. These IP are optimized for particular target devices, and help to meet higher performance and area constraints while still maintaining a higher level of abstraction.

To add these blocks into your design, navigate to the Digital Signal Processing library in the Model Composer block library.

Figure 39: **DSP Library**



IP Generation

As discussed in Generating Packaged IP for Vivado, Model Composer can generate a packaged IP of the current model, which can be integrated into a platform or used in other designs. By default, when the model does not utilize one of the RTL IP-based blocks discussed above, the output is generated at the following location:

```
<Target directory>/<Subsystem name>_prj/solution1/impl/ip
```

However, when the Model Composer model includes one or more of the RTL IP-based blocks, the output IP is generated at:

```
<Target directory>/<Subsystem name>_ip/ip_repo
```

Verification

As discussed in Chapter 5: Simulating and Verifying Your Design, when an RTL IP-based block is not used in the model, RTL co-sim is done by Vivado HLS after generating the IP for the current model.

TIP: *To run RTL co-sim, select the* ***Create and execute testbench*** *option and set the 'Export type' to 'IP Catalog' or 'System Generator' as described in Adding the Model Composer Hub.*

However, when an RTL IP-based block is present in the model, Model Composer runs RTL co-sim on the final packaged IP of the current model using Vivado IDE tools with the project generated under `<Target directory>/<Subsystem name>_cosim` directory.

When verification is complete the following message is printed:

```
INFO: XMC COSIM: ***** Model Composer RTL Co-simulation has finished. *****

Info: /OSCI/SystemC: Simulation stopped by user.

Info: /OSCI/SystemC: Simulation stopped by user.
$stop called at time : 1632500 ps
Info: [Simtcl 6-16] Simulation closed
```

Limitations

- RTL IP-based blocks are not supported when nested in:
    - Window Processing kernels
    - If Action Subsystem blocks
- Latency, timing, and resource estimates will not be provided in the MATLAB Editor after generating the packaged IP of the Model Composer model.
- IP-based blocks do not support throughput control as described in Controlling the Throughput of the Implementation.
- Only real-type signals are accepted for AXI4-Lite port interfaces. Complex-type signal are not supported.

Model Composer Blocks

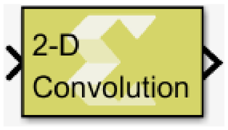
The Xilinx Model Composer block library includes the following blocks. The content shown here can also be accessed from the **Help** command for the block within the Model Composer block library in Simulink.

2-D Convolution

Perform a two dimensional convolution of an image with a user-defined kernel

Library

Computer Vision



Description

The 2-D Convolution block takes as input two signals: an input grayscale image, and a convolution kernel matrix.

The result is a single output image of the same dimensions as the input image.

Data Type Support

The input image data type is 8-bit unsigned integer of real numeric type.

Filter kernel coefficients are user-defined within certain constraints:

- Coefficients are converted into signed 16-bit integer or fixed-point values with a user specified number of fractional bits.
- The numeric type of coefficients is always real.
- The filter kernel must be two-dimensional and have an odd number of rows and columns, for example, 3x3, 7x5, 3x9.
- The kernel size is not limited, but in practice should be kept at or below 31x31 coefficients due to resource usage.

Parameters

Filter kernel

The **Filter kernel** parameter is provided via an edit box to either enter the convolution kernel directly, or in form of a Simulink workspace variable. Since Simulink does not natively support the x_fixed data type used by Model Composer, the block will internally convert the kernel coefficients from the native Simulink data type to the appropriate int16 or x_fixed data type, as specified via the **Coefficient fractional bits** parameter.

The size of the convolution kernel is determined by the coefficient matrix entered. The kernel size must have an odd number of rows and columns. Any kernel with an even number of rows or columns will be rejected.

The 2-D Convolution block accepts filter kernels only in integer and fixed point format. Convolution kernel coefficients in floating point format need to be converted into either fixed-point or integer types. This requires an additional parameter **Coefficient fractional bits** to fully specify the coefficient data type (a signed type with a word length of 16 bits is implied and cannot be changed).

Coefficient fractional bits

This parameter identifies the data type to use for the kernel coefficients. Since the word length for coefficients is fixed at 16 bits and coefficients are always signed, the number of fractional bits, F, completely specifies the data type.

For the special case of F=0, the resulting data type is int16, otherwise it is x_fixed_16_N, where N = 16 - F.

Settings for the **Coefficient fractional bits** parameter are:

Setting	Description
0	Coefficient data type will be int16
1 ... 15	Coefficient data type will be x_fixed_16_N with N = 16 - F.

Output Data Type

The output data type is application dependent and can be set by the user. The two choices are unsigned 8-bit integer, or signed 16-bit integer types. The filter uses saturating arithmetic, so for non-unity gain filter kernels the extended range of a 16-bit output data type may be preferable to avoid loss of fidelity.

Settings for the **Output data type** parameter are:

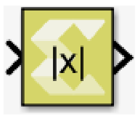
Setting	Description
uint8	Sets the output data type to unsigned 8-bit integer.
int16	Sets the output data type to signed 16-bit integer.

Abs

Compute element-wise absolute value of input signal

Library

Math Functions/Math Operations



Description

The Abs block computes element-wise absolute value on the input data.

Data Type Support

Data type support is:

- Dimension: Input can be scalar, vector, or matrix.
- Data Types: Input supports signals of integer type, floating point data type (double, single, and half), and signed and unsigned fixed point type.
- Complex Numbers Support: Yes.

Output has the same dimension and data type as the input. Output of the Abs block is always real.

Parameters

The Abs block has no parameters to set.

atan

Compute element-wise arctangent function of an argument.

Library

Math Functions / Math Operations

Description

The atan block returns the output of the atan (`x`) function for each element in array `x`.

Data Type Support

Data types accepted at the inputs of the block are:

- Dimension: Input can be scalar, vector, or matrix.
- Data Types: Input supports signals of integer type, floating point type (double, single and half) and fixed point type.
- Complex Number Support: No

Output has the same dimension and type as the input. However, If the data type of the input is a fixed point type, the data type of the output is fixed point type with integer width fixed as 2. The reason for this is that the output of the atan function is between -π/2 and π/2. Use the atan2 function if you need the output of the function to be between -π and π.

Parameters

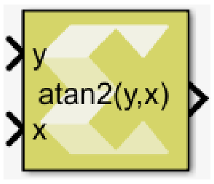
The Atan block has no parameters to set.

atan2

Compute element-wise four-quadrant inverse tangent of input signal.

Library

Math Functions / Math Operations



Description

The atan2 block returns the output of the function atan2(`y,x`).

Data Type Support

Data types accepted at the inputs of the block are:

- Dimension: Inputs can be scalar, vector, or matrix. If one of the inputs is scalar and the other is a vector or matrix then the scalar input is expanded to match the other input dimension, and operation will be performed element wise. If both inputs are non-scalar, then they must match in dimension.
- Data Types: Input supports signals of integer type, floating point type (double, single, and half) and signed and unsigned fixed point type. Both inputs must be of the same data type.
- Complex Number Support: No

Except for fixed-point input data type, output has the same data type as the input. For fixed point input, the output data type will be signed fixed-point with a 3-bit integer width, to be able to represent numbers between $-\pi$ and π.

Parameters

The atan2 block has no parameters to set.

Bilateral Filter

Apply a Bilateral Filter function to an unsigned 8-bit integer grayscale image

Library

Computer Vision

Description

The Bilateral Filter block applies a nonlinear, edge preserving and noise reducing smoothing filter to the input image. The filter kernel can be of sizes 3x3, 5x5, or 7x7. The filter has two parameters:

- Spatial kernel variance for smoothing differences in pixel location (**Spatial sigma** parameter)
- Color space (range) kernel variance influencing the smoothing of differences in pixel intensity levels (**Color space sigma** parameter)

The intensity of the output pixel $I_D(i, j)$ at row i and column j is computed as the normalized weighted sum of its neighboring pixels I(k,l) as follows.

$$I_D(i,j) = \frac{\sum_{k,l} I(k,l)w(i,j,k,l)}{\sum_{k,l} w(i,j,k,l)}$$

where w(i,j,k,l) is the weight given to an input pixel in the neighborhood of the output pixel, computed as follows.

$$w(i,j,k,l) = \exp\left(-\frac{(i-k)^2+(j-l)^2}{2\sigma_d^2} - \frac{\|I(i,j)-I(k,l)\|^2}{2\sigma_r^2}\right)$$

where σ_d is the spatial variance parameter, and σ_r is the color space (range) variance.

Data Type Support

The Bilateral Filter block accepts grayscale images (real-valued 2-D matrices) of type uint8 as input, and the output has the same type and dimensions as the input.

Parameters

Filter width

The width of the filter is a **3x3**, **5x5**, or **7x7** kernel applied to the neighborhood of the input pixel location.

Following are the settings for the **Filter width** parameter.

Table 26: **Filter Width Settings**

Setting	Description
3x3	Sets the filter kernel size to 3x3 coefficients.
5x5	Sets the filter kernel size to 5x5 coefficients.
7x7	Sets the filter kernel size to 7x7 coefficients.

Spatial sigma

The **Spatial sigma** parameter represents the variance of the spatial kernel.

The parameter is set to a positive scalar value of either single or double precision floating point type.

Color space sigma

The **Color space sigma** parameter represents the variance of the color space (range) kernel.

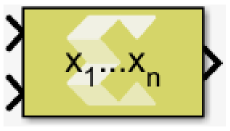
The parameter is set to a positive scalar value of either single or double precision floating point type.

Bit Concat

Perform bitwise concatenation of input values into a single output value

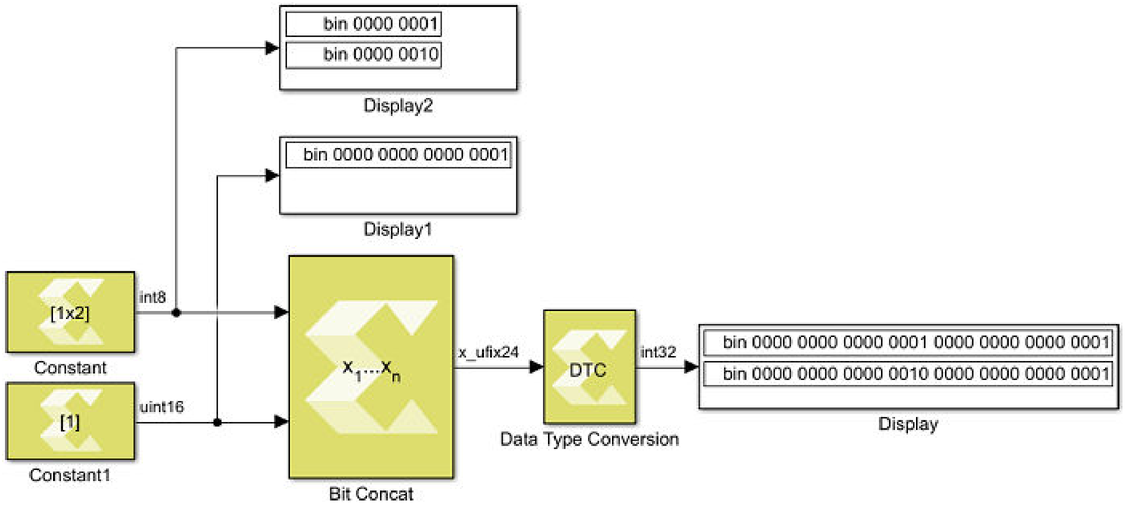
Library

Logic and Bit Operations



Description

Starting from the input with the highest order (the first input at the top in normal block orientation) the bit values of all input ports are concatenated into a single output bit vector. For multidimensional inputs the dimensions of all inputs must match and concatenation proceeds element-wise as it would in the scalar case. A scalar value on one input is automatically expanded to match the dimensions of the other input.



Data Type Support

Data type support for the Bit Concat block is:

- All integer types (including Boolean) and fixed-point types are supported. Floating point types are not supported.
- All inputs must be of real numeric type. Complex types are not supported.
- Scalars, Vectors and 2-D Matrices are supported. Unless an input is a scalar, the dimensions of all inputs must agree.
- The output type is always an unsigned fixed-point type without fractional bits.

Parameters

Number of inputs

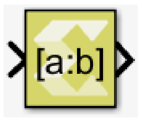
Sets the number of inputs to be concatenated. The minimum number of inputs is 2, the maximum is 128. The sum of all input bit widths shall not exceed 1024 bits.

Bit Slice

Extract a range of bits from a value

Library

Logic and Bit Operations



Description

The Bit Slice block allows the element-wise extraction of a contiguous set of bits from the input values. The extracted bits are returned as unsigned fixed point values of an all-integer range, and you specify the width of the specified extraction range. The block dialog box allows you to specify the range of bits using one of these methods:

- **Bottom bit + width** - You specify the bottom bit and the number of bits to be extracted (**Slice width**).
- **Top bit + width** - You specify the top bit and the number of bits to be extracted (**Slice width**).
- **Top and bottom bit** - You specify the top and bottom bits and the number of bits to be extracted is implied.

The top and bottom bit specifications have multiple ways of specifying the position in relation to either the Least Significant Bit (LSB), the Binary Point of a fixed-point value, or the Most Significant Bit (MSB). In case of integer inputs the Binary Point and Least Significant Bit options are equivalent. Offsets to specify the position relative to these anchors can be positive or negative. However, an error will occur during simulation and/or code generation if the extraction range lies outside of the input type bit range.

Data Type Support

The Bit Slice block accepts any real-valued integer or fixed-point type of any dimension N ≤ 2. Floating point values and complex numeric types are not supported.

The output data type is always a real-valued unsigned fixed-point type with integer-only range. The output data has the same width as the extraction range you specify.

The output dimensions are the same as the input dimensions.

Parameters

Specify range as

The **Specify range as** parameter specifies the extraction range.

- If you select **Bottom bit + width**, parameters in the **Bottom of bit range** section are enabled and parameters in the **Top of bit range** section are disabled.
- If you select **Top bit + width**, parameters in the **Top of bit range** section are enabled and parameters in the **Bottom of bit range** section are disabled.
- If you select **Top and Bottom bit**, parameters in both the **Top of bit range** section and the **Bottom of bit range** section are enabled, and the **Slice width** parameter is disabled.

Following are the settings for the **Specify range as** parameter.

Table 27: **Specify Range As Parameter**

Setting	Description
Bottom bit + width	The **width** specifies the number of bits to extract. The **Bottom bit** of the range specifies the offset at which the range begins (offset of the least significant bit to be extracted).
Top bit + width	The **width** specifies the number of bits to extract. The **Top bit** of the range specifies the offset at which the range begins (offset of the most significant bit to be extracted).
Top and bottom bit	The **Top** bit of the range specifies the offset of the most significant bit to be extracted. The **bottom** bit of the range gives the offset of the least significant bit to be extracted. The width of the extracted range is given implicitly.

Slice width

Specifies the width of the bit range to be extracted. **Slice width** is only enabled if the **Specify range as** parameter is set to **Top bit + width** or **Bottom bit + width**.

Enter a scalar positive integer value for **Slice width**.

Bit position relative to

Defines the basis for offset specifications in both **Top of bit range** and **Bottom of bit range** sections of the block dialog box.

Following are the settings for the **Bit position relative to** parameter.

Table 28: **Bit Position Relative To Parameter**

Setting	Description
Least Significant Bit	Defines the offset parameter as counting from the LSB of the input value, with offset 0 denoting the LSB, offset 1 denoting the bit to the left of the LSB, etc. If the **Least Significant Bit** setting is selected, the **With offset** parameter cannot specify a negative offset.
Binary point	Defines the offset parameter as counting from the binary point of a fixed point value, with offset 0 denoting the least significant integer bit. A negative offset denotes a range starting in the fractional bits with offset -1 being the most significant fractional bit. A positive offset denotes a range starting in the integer portion of the value.
Most Significant Bit	Defines the offset parameter as counting from the MSB of the input value, with offset 0 denoting the MSB, offset -1 denoting the bit to the right of the MSB, etc. If the **Most Significant Bit** stetting is selected, the **With offset** parameter cannot specify a positive offset.

With offset

Specifies the offset to be applied to the basis specified by the corresponding **Bit position relative to** parameter. The **With offset** parameter is available in both **Top of bit range** and **Bottom of bit range** sections of the block dialog box.

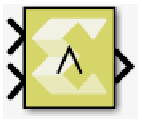
Negative offsets specify bit positions to the right of the anchor (zero offset basis). Positive offsets specify bit positions to the left of the anchor.

Bitwise AND

Perform element and bitwise Boolean AND operation on the inputs

Library

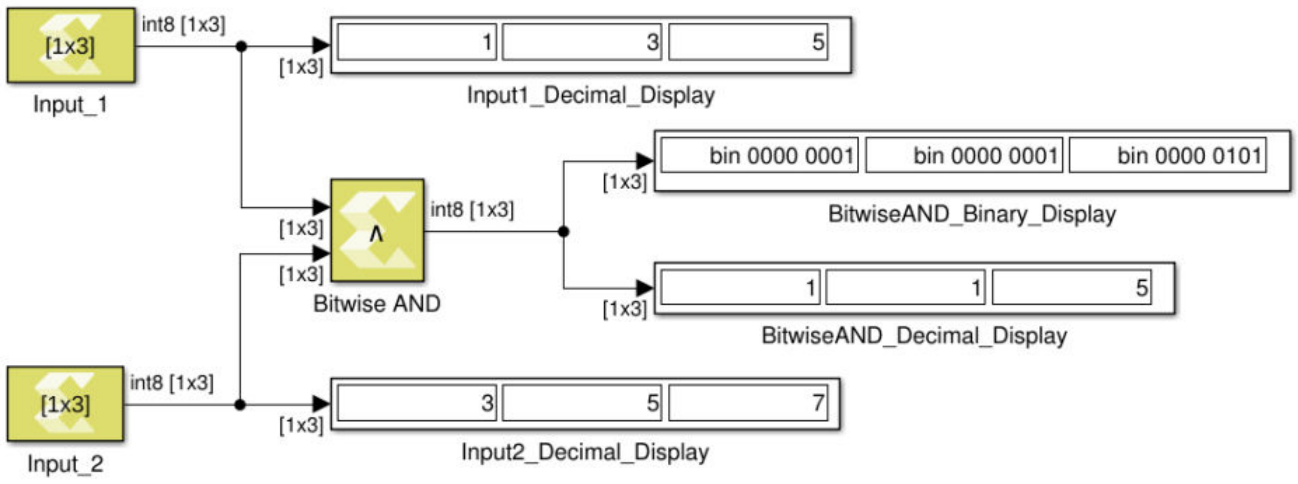
Logic and Bit Operations



Description

The Bitwise AND block has two input signals and one output signal. The block performs element and bit-wise Boolean AND operation on the inputs. The first input corresponds to the top input port and the second input to the bottom input port. Both input ports must have the same data type. The dimension of the output signal matches the dimensions of the input signals. Unless an input is a scalar, the dimensions of all inputs must agree. A scalar value on one input is automatically expanded to match the dimension of the other input.

Figure 40: **Bitwise AND**



Data Type Support

The block accepts integer, fixed-point, and Boolean types of real numeric type. It does not support floating point input types. Complex signals are not supported for this operation.

Parameters

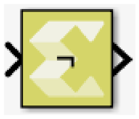
The Bitwise AND block has no parameters to set.

Bitwise NOT

Perform element and bit-wise Boolean NOT operation on the input

Library

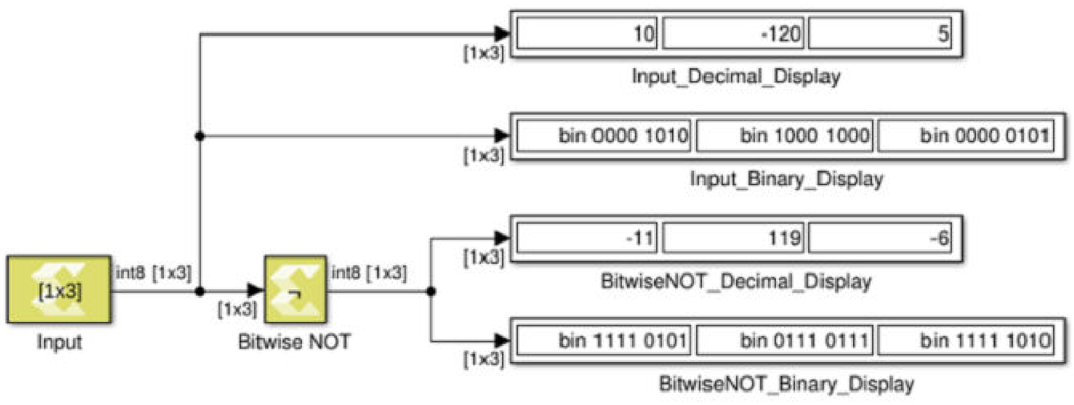
Logic and Bit Operations



Description

The Bitwise NOT block has one input signal and one output signal. It performs element and bit-wise Boolean NOT operation on the input. The dimension of the output signals matches the dimension of the input signals.

Figure 41: **Bitwise NOT**



Data Type Support

The Bitwise NOT block accepts integer, fixed-point, and Boolean types of real numeric type. The block does not support floating point input types. Complex signals are not supported.

Parameters

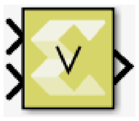
The Bitwise NOT block has no parameters to set.

Bitwise OR

Perform element and bitwise Boolean OR operation on the inputs

Library

Logic and Bit Operations



Description

The Bitwise OR block has two input signals and one output signal. The block performs element and bit-wise boolean OR operation on the inputs. The first input corresponds to the top input port and the second input to the bottom input port. The dimension of the output signal matches the dimensions of the input signals.

Data Type Support

The Bitwise OR block supports native and fixed-point types except the floating point types (double, single, and half). If one of the inputs is scalar, the output dimension is non-scalar. If both inputs are non-scalar, they must have the same dimension. The dimension can be scalar, vector, or matrix.

Parameters

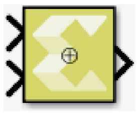
The Bitwise OR block has no parameters to set.

Bitwise XOR

Perform element and bit-wise Boolean XOR operation on the inputs

Library

Logic and Bit Operations



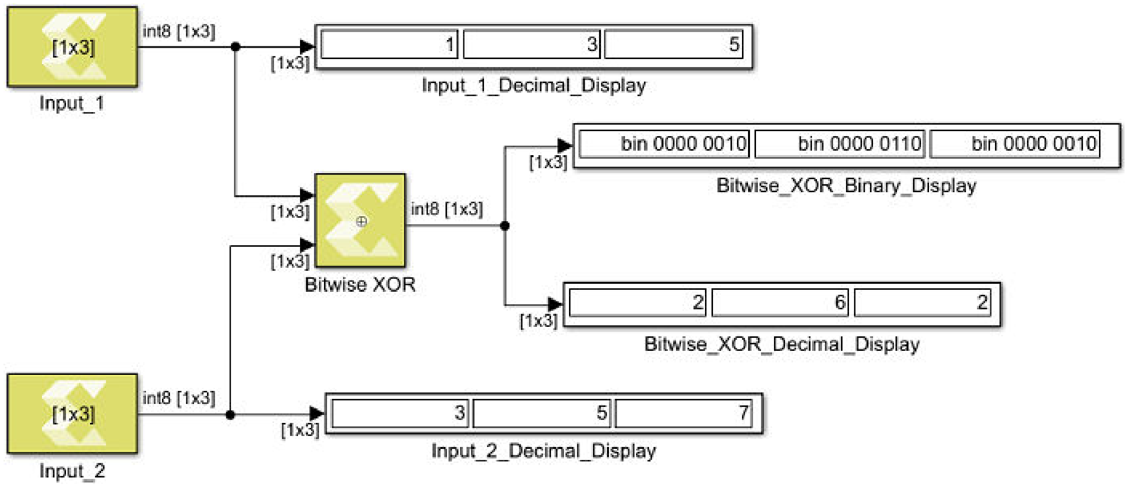
Description

The Bitwise XOR block has two input signals and one output signal. The block performs element and bit-wise Boolean XOR operation on the inputs. The first input corresponds to the upper input port and the second input to the lower input port. The dimension of the output signal matches the dimensions of the input signals.

Data Type Support

If one of the inputs is scalar, the output dimension is non-scalar. If both inputs are non-scalar, they must have the same dimension. The dimension can be scalar, vector, or matrix.

Figure 42: **Bitwise XOR**



Parameters

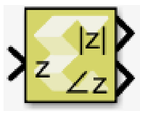
The Bitwise XOR block has no parameters to set.

Complex to Polar

Perform an element-wise conversion of complex input signals into magnitude and radiant phase angle

Library

Math Functions / Math Operations



Description

The Complex to Polar block accepts a complex-valued signal of floating point data types such as double, single and half. It outputs the magnitude and phase angle of the input signal. The outputs are real values of the same data type as the block input. The input can be a scalar, vector or matrix of complex signals, in which case the output signals are also scalar, vector or matrix. The magnitude signal array contains the magnitudes of the corresponding complex input elements. The angle output similarly contains the angles of the input elements in radian.

Data Type Support

Data type support for the input port is:

- Dimension: The input can be scalar, vector or matrix. If the input is not scalar then the outputs have the same dimension as the input.
- Data Types: Input supports complex signal of floating point data type such as double, single, and half.

The outputs are always real-valued signals of floating point data type such as double, single, and half.

Parameters

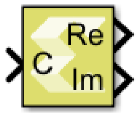
The Complex to Polar block has no parameters to set.

Complex to Real-Imag

Computes the real and imaginary components of the input

Library

Math Functions / Math Operations



Description

The Complex to Real-Imag block accepts a complex or real signal of any valid data type and outputs the real and/or imaginary components of the input signal. The outputs are real-valued signals of the same data type as the block input. The input can be a scalar, vector, or matrix. The outputs have the same dimensions as that of the input.

Data Type Support

Data type support for the input port is:

- Supports real or complex input of any valid data type. The outputs are real-valued signals of the same data type as the block input.
- The input can be a scalar, vector, or matrix. The outputs have the same dimensions as that of the input.

Parameters

- **Output:**

  This parameter specifies the kind of output the block produces.

  Settings for the **Output** parameter are:

Table 29: **Output Parameters**

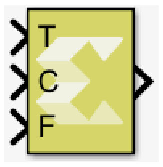
Setting	Description
Real and imag	Outputs real and imaginary parts of the input signal as Re and Im outputs of the block, respectively.
Real	Outputs the real part of the input signal as Re output of the block.
Imag	Outputs imaginary part of the input signal as Im output of the block. If the input is real, the Im output is zero valued.

Conditional

Pass through input T when control input C satisfies a selected criteria; otherwise, pass through input F. The first and third input ports are data ports, and the second input port is the control port.

Library

Signal Routing



Description

The Conditional block passes through input T when control input C satisfies the selected criteria; otherwise, it passes through input F. The first and third input ports are data ports, and the second (or middle) input port is the control port.

Data Type Support

Data type support for the Conditional block is:

- Inputs T and F should either be both complex or real, and can be scalar, vector, or matrix. For non fixed-point data types, inputs T and F do not necessarily have the same data type. But, for fixed point data types, both these inputs should be of fixed-point type. In that case the output data type will be fixed-point, and the number of integer and fractional bits will be set to accommodate both inputs without loss of precision. For example, if one of the inputs is x_sfix16_En10 (1 signed bit, 5 integer bits and 10 fractional bits), and the other is x_sfix16_En5 (1 signed bit, 10 integer bits, and 5 fractional bits), the output will be x_sfix21_En10 (1 signed bit, 10 integer bits, and 10 fractional bits).
- Input C can be a scalar, vector, or matrix if inputs T and F are scalars and Threshold parameter is scalar. In this case, the output dimension matches with the input C dimension. Input C must match the dimension of input T and F if they are non scalars. Input C should be real and can be of any data type..

Parameters

- **Criteria for passing first input:**

  This parameter is used to select the condition under which the block passes the first input (T). If the control input C meets the condition set in the **Criteria for passing first input** parameter, the block passes the first input (T). Otherwise, the block passes the third input (F).

  Settings for the **Criteria for passing first input** parameter are:

  Table 30: **Criteria for Passing First Input Settings**

Setting	Description
C >= Threshold	Select input T if control input C is greater than or equal to the **Threshold** parameter.
C > Threshold	Select input T if control input C is greater than the **Threshold** parameter.
C ~= 0	Select input T if control input C is not equal to 0. Selecting **C ~= 0** disables the **Threshold** parameter.

- **Threshold:**

  This parameter assigns the switch threshold that determines which input the block passes to the output. Threshold parameter is rounded to the same data type as that of the C input.

  - **Settings:**

Table 31: **Supported Threshold Settings**

Settings	Description
0	default value
`real number`, `vector`, or `matrix`	any real scalar, vector or matrix

TIP:

To specify a non-scalar threshold, use brackets. For example, the following entries are valid: [1 3 5].

For a non-scalar threshold, the inputs must be scalars. In that case, the output dimension is the same as the threshold dimension, and input 2 is compared to each element of the threshold, and depending on the criteria, either T or F is selected to populate the output signal.

- **Dependencies:** Setting Criteria for passing first input to C ~= 0 disables this parameter.

Conjugate

Apply element-wise complex conjugate operation to the input signal

Library

Math Functions / Math Operations

Description

The Conjugate block applies element-wise complex conjugate operation to the input signal.

The conjugate of a complex number is the number with equal real part, where the imaginary part is equal in magnitude but opposite in sign. The complex conjugate of a+bi is a-bi.

Data Type Support

Data type support for the input is:

- The input signals can be signed integer, fixed-point, or floating-point data type.
- Boolean and unsigned data types are not supported.
- The input signals can be a scalars, vectors, or matrices.
- The input signal supports complex type.

The data type and dimension of the output signal are the same as those of input signal. For complex type input, only the magnitude of the imaginary part changes.

Parameters

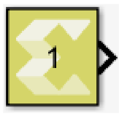
The Conjugate block has no parameters to set.

Constant

Provides constant value as a source.

Library

Source



Description

The Constant block generates a constant output of the value specified by the **Constant value** parameter. If you select **Interpret vector parameters as 1-D** parameter and specify the constant value as a scalar, row matrix, or column matrix, then the output is a 1-D array. Otherwise, the output is always two-dimensional. The Constant block supports real or complex constant values.

Data Type Support

By default, the Constant block outputs a signal with **double** data type and the same complexity as the **Constant value** parameter. However, you can specify the output to be any data type that Model Composer supports, including fixed-point and half data types by selecting the **Output data type** parameter.

Parameters

- **Constant value:**

  The Constant value parameter specifies the constant value output of the block.

  You can enter any expression that MATLAB® evaluates as a scalar or matrix.

  Table 32: **Settings**

Choices	Description
1.0	Constant Value

- **Interpret vector parameters as 1-D:**

  Specifies whether the constant value should be interpreted as a 1-D array.

Table 33: **Settings**

Choices	Description
On	If the specified constant value is a scalar, row matrix, or column matrix, then the output is a 1-D array. Otherwise, the output is a 2-D matrix.
Off	The output is a 2-D scalar or matrix.

- **Sample time:**

  Specifies block sample time as a numerical value. The sample time of a block indicates when, during simulation, the block generates outputs or updates its internal state.

  The block allows you to specify a block sample time directly as a numerical value. For example, to generate output at every two seconds, you can directly set the discrete sample time by specifying the numerical value of 2 as the **Sample time** parameter.

  Settings for **Sample time** are:

Table 34: **Sample Time Parameter**

Sample Time Type	Sample Time Supported	Description	Supported?
Discrete	T_s	Generates output at discrete samples T_s. Discrete sample time is supported with the initial time offset value fixed to 0. The initial offset value is not configurable.	Yes
Continuous	0	Generate output continuously by dividing the sample hits into major time steps and minor time steps. The Simulink ODE solver you choose integrates all continuous states from the simulation start time to a given major or minor time step.	No

Table 34: **Sample Time Parameter** *(cont'd)*

Sample Time Type	Sample Time Supported	Description	Supported?
Inherited	-1	The sample time value is inherited from other sources. It is determined by applying a set of heuristics and based on the context of the block within the model by Simulink. Allowing a design to inherit sample time maximizes its reuse potential. For example, a design might fix the data types and dimensions of all its input and output signals. But you could reuse the design with different sample times (for example, discrete at 0.1 or discrete at 0.2, triggered, and so on).	Yes
Constant	inf	Constant sample time. Same as inherited sample time for Model Composer blocks.	Yes
Variable	-2	Variable Sample time.	No
Triggered	-1 (implicit)	Execute the block upon some implicit condition when it is inside a subsystem like triggered, function call, or iterator subsystem.	Yes

For additional details for simulating sample time, see Types of Sample time in the Simulink documentation.

- **Output data type:**

  This parameter specifies the data type of the output signal.

  If the output data type is one of the integer types, then the Constant value is rounded off as explained below.

  A value with the fractional part less than 0.5 is rounded towards zero, the fractional part more than 0.5 is rounded away from zero.

  In case of a tie (fractional part is 0.5), the Constant value is rounded up, i.e. the negative Constant value, is rounded towards zero and the positive Constant value is rounded away from zero.

- **Settings:**

  The following data types are supported:

Table 35: **Output Data Type Parameter**

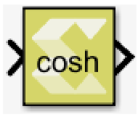
Setting	Description
double, single, and half	Floating point data types.
int8, uint8, int16, uint32, int32, uint32	Signed and unsigned integer data types.
Logical	Boolean data type.
Fixed point data type	• Fixed-point arithmetic data type with configurable output data type attributes like signedness, word length, fractional length. • Constant value conversion attributes rounding and overflow for reading constant value parameter.
Data type Expression	

cosh

Element-wise computation of the hyperbolic cosine for a given argument

Library

Math Functions / Math Operations



Description

The cosh block returns the output of the function cosh(x), which is the hyperbolic cosine, for each element in array x.

The hyperbolic cosine of x is:

$$\cosh(x) = \frac{e^{x} + e^{-x}}{2}$$

Data Type Support

Data type support is:

- Dimension: Input can be scalar, vector, or matrix.
- Data Types: Input supports signals of integer type, floating point data types (double, single, and half) and fixed point type.
- Complex Numbers: Complex numbers are not supported.

Output has the same dimension and data type as the input.

Parameters

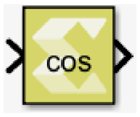
The cosh block has no parameters to set.

Cosine

Computes cosine value for the input.

Library

Math Functions / Math Operations



Description

The Cosine block returns the output of the function cos(x) for each element in array x.

Data Type Support

Data type support is:

- Dimension: Input can be scalar, vector, or matrix.
- Data Types: Input supports signal of floating-point data types (double, single, and half), and signed fixed point type. It does not support integer, Boolean, and unsigned fixed point data types.
- Complex Numbers: Complex numbers are not supported.

Output has the same dimension and data type as the input. However, if the data type of the input is a fixed point type, the data type of the output is fixed point type with integer width fixed as 2.

Parameters

The Cosine block has no parameters to set.

Cumulative Sum

Compute the cumulative sum along the specified dimension of the input

Library

Math Functions / Math Operations

Description

The Cumulative Sum block computes the cumulative sum of the input signal along the specified dimension or across time (running sum). The output signal has the same dimensions, data type and complexity as the input signal.

Summing is performed in this way:

- Summing along Rows: If the block is configured for cumulative sum along rows, each element in the output signal is the sum of the corresponding element in the input and all of the elements in the same row and to the left of that element. If the input is 1-dimensional, each element in the output signal is the sum of the corresponding element in the input and all of the preceding elements.
- Summing along Columns: If the block is configured for cumulative sum along columns, each element in the output signal is the sum of the corresponding element in the input and all of the elements in the same column and above of that element. If the input is 1-dimensional, each element in the output signal is the sum of the corresponding element in the input and all of the preceding elements.
- Running sum: If the block is configured for running sum, each element in the output signal is the sum of the corresponding element in the input signal over time. In this case you can specify an optional reset port and restart the running sum when the reset signal is asserted.

Data Type Support

Data type support is :

- Data input: The data In input is the data signal to be summed. It supports integer, fixed point and floating-point data types but not boolean. The signal can be complex or real. The signal can be a scalar, vector or matrix.
- Reset: The Reset input is applicable only for running sum with non-default reset type. The reset signal must be scalar and real, and its data type must be integer or floating-point. Fixed point data type is not supported.
- Output: The data type, dimension and complexity of the output signal are the same as those of input signal.

Parameters

Sum input along:

This parameter specifies the dimension along which sum elements are computed.

Settings for the **Sum input along:** parameter are:

Table 36: **Sum Input Along Parameter**

Setting	Description
Columns	The block computes the cumulative sum of each column of the input.
Rows	The block computes the cumulative sum of each row of the input.
Channels (running sum)	The block computes a running sum for each element of the input across time. When you select the **Channels (running sum)** option, you will also have to specify a **Reset port** parameter.

Reset port

This parameter applies only to running sum. The **Reset port** parameter appears if you select **Channels (running sum)** for the **Sum input along:** parameter.

Settings for the **Reset port** parameter are:

Table 37: **Reset Port Paremeter**

Setting	Description
None	Omits the Reset port.
Non-zero sample	Triggers a reset operation at each sample time that the Reset input is not zero.

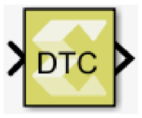
Data Type Conversion

Convert the input to the data type of the output.

The block warns or errors out when an integer or fixed-point output overflows during simulation. To configure, select **Configuration Parameters→Diagnostics→Data Validity**. In the Data Validity pane, set **Wrap** or **Saturate** to Overflow.

Library

Signal Attributes



Description

The Data Type Conversion block has one input and one output. It converts the value of the input signal to the data type of the output. This conversion tries to preserve the mathematical value of the input signal. The data type of the output is specified via the mask dialog. The conversion is governed by the following rules:

- Conversions where the output data type is fixed-point, first select the nearest number that can be represented, taking into account the overflow mode. In case of a tie, the rounding mode breaks the tie.
- Conversions where the output data type is integer are performed as in the C language. Overflow is handled via truncation.
  - As per IEEE Standard for Floating-point Arithmetic (IEEE Standard 754, Section-7.2), conversion from floating-point to integral is an invalid operation, when the floating-point value is outside the range of the destination integer data type. In this case, the output integer value depends upon the implementation of a C compiler. Hence, the results from the Model Composer Data Type Conversion block may differ from the results from Simulink® Data Type Conversion block.
  - When the floating-point input value is outside the range of the integer data type, the simulation results between Model Composer and RTL co-simulation in Vivado® HLS may also differ.
  - During simulation, to check whether the input floating-point value goes outside the range of the destination integer type, in Simulink select **Model Configuration Parameters → Diagnostics → Data Validity** . Then set **Saturate on overflow** to either **warning** or **error**.
- Conversions where the output data type is floating point follow the rules implemented in the C language.

Data Type Support

The input signal can be double, single, an integer, boolean, Xilinx supported half or Xilinx supported fixed-point data type.

The data type of the output is specified the mask parameters.

The input can be real or complex, and scalar, vector, or matrix. The output signal has the same complexity and dimensions as the input signal.

Parameters

- **Output data type:**

  This parameter specifies the data type of the output signal. If `fixed` is specified, more parameters are available.

  Settings for the **Output data type** parameter are as follows.

Table 38: **Output Data Type Parameter**

Setting	Description
double	double precision floating point
single	single precision floating point
int8	8-bit signed integer

Table 38: **Output Data Type Parameter** *(cont'd)*

Setting	Description
uint8	8-bit unsigned integer
int16	16-bit signed integer
uint16	16-bit unsigned integer
int32	32-bit signed integer
uint32	32-bit unsigned integer
logical	boolean
fixed	Xilinx supported fixed-point
half	Xilinx supported half precision floating point
data type expression	

- **Signedness:**

  If the **Output data type** is set to **fixed**, the **Signedness** parameter specifies whether the output is a signed fixed-point or unsigned fixed-point data type.

  Settings for the **Signedness** parameter are as follows.

  Table 39: **Signedness Parameter**

Setting	Description
Signed	The output type contains both positive and negative numbers.
Unsigned	The output type contains only non-negative numbers.

  This parameter is available only if **fixed** is selected as the setting for parameter **Output data type**.

- **Word length:**

  If the **Output data type** is set to **fixed**, the **Word length** parameter specifies the number of bits used to represent it.

  Table 40: **Word Length Parameter**

Choices	Description
16	
N	A positive integer

  This parameter is available only if **fixed** is selected as the setting for parameter **Output data type**.

- **Fractional length:**

If the **Output data type** is set to `fixed`, the **Fractional length** parameter specifies the number of bits to the right of the binary point.

Table 41: **Fractional Length Parameter**

Choices	Description
10	
N	An integer

This parameter is available only if **fixed** is selected as the setting for parameter **Output data type**.

- **Round:**

  If the **Output data type** is set to **fixed**, the **Round** parameter allows you to select among five rounding and two truncation options.

  If one of the five rounding options is selected, the block always rounds to the nearest *supported precision*. The five rounding choices are relevant only in case of a tie. For example, assume the output type is signed fixed-point, with a word length of 6 and a fractional length of 2, and the input to the block is 2.74. In this case, the output is rounded to the nearest supported precision, 2.75, regardless of which one of the five rounding modes is selected. However, if the input value is 2.625 (halfway between 2.5 and 2.75), then the output value depends on the chosen rounding mode. If **Round to plus infinity** is selected, the value will be 2.75, and if **Round to zero** is selected, the value will be 2.5. For more information on this, refer to the *Vivado Design Suite User Guide: High-Level Synthesis* (UG902).

  If one of the two truncation options is selected, the output will be truncated to the supported precision specified by the truncation selection.

  Truncation to minus infinity is the default setting for **Round** and requires the smallest hardware resources among all the options.

  The **Round** parameter is available only if **fixed** is selected as the setting for the **Output data type** parameter.

  Settings for the **Round** parameter are:

Table 42: **Round Parameter**

Setting	Description
Round to plus infinity	Rounding to plus infinity
Round to zero	Rounding to zero
Round to minus infinity	Rounding to minus infinity
Round to infinity	Rounding to infinity
Convergent rounding	Convergent rounding

Table 42: **Round Parameter** *(cont'd)*

Setting	Description
Truncation to minus infinity	Truncation to minus infinity
Truncation to zero	Truncation to zero

- **Overflow:**

  If the **Output data type** is set to `fixed`, the **Overflow** parameter specifies the overflow mode applied during conversion.

  This parameter is available only if **fixed** is selected as the setting for parameter **Output data type**.

  Settings for the **Overflow** parameter are:

  Table 43: **Overflow Parameter**

Setting	Description
Saturation	Saturation
Saturation to Zero	Saturation to zero
Symmetrical Saturation	Symmetrical saturation
Wrap around	Wrap around
Sign-Magnitude Wrap Around	Sign magnitude wrap around

- **TypeExp:**

  If the **Output data type** is set to **data type expression**, the **Type Expression** parameter specifies the output data type as a string.

  This parameter is available only if **data type expression** is selected as the setting for parameter **Output data type**.

- **Saturate on integer overflow:**

  This parameter specifies whether integer overflow is handled by wrapping (default) or by saturating. This parameter is relevant only if the output is integral (int8, int16, int32, uint8, uint16, uint32).

  When overflow is detected, the Diagnostic Viewer displays messages depending on the diagnostic action selected in the Configuration Parameters dialog box. To configure, in the **Configuration Parameters → Diagnostics → Data Validity** pane, set the Wrap or Saturate on overflow.

  Settings for the **Saturate on integer overflow** parameter are:

Table 44: **Saturate On Integer Overflow Parameter**

Setting	Description
Not selected	Integer overflow is handled by wrapping.
Selected	Integer overflow is handled by saturation.

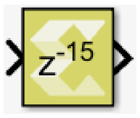
If the **Output data type** is set to **fixed** and overflow is detected, the Diagnostic Viewer displays messages that depend on the diagnostic action you specify in the Simulink Editor. To configure, select **Simulation → Model Configuration Parameters → Diagnostics → Data Validity** for your model in the Simulink Editor, then set the **Wrap on overflow** or **Saturate on overflow** parameter.

Delay

Delay input signal by specified number of samples

Library

Signal Operations



Description

The delay block produces an output signal by delaying the input signal by the number of samples specified in the block dialog box. If the latency of the block is N, then the N-1 first output samples are always 0, and the N-th output sample is the first input sample.

Data Type Support

All data types are supported.

Input can be a vector or a matrix. If input is a vector or a matrix and the latency value is a scalar, the scalar value will apply to all the elements of the input.

Output is complex if the input is complex.

Parameters

Latency

The **Latency** parameter specifies the number of samples by which the input signal is delayed.

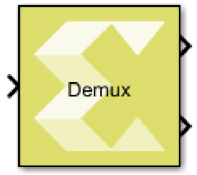
Latency should be a real, non-negative, scalar integer with minimum value as 1 and maximum value as 2^25.

Demux

Separates a vector input into a number of scalar and vector outputs.

Library

Signal Routing



Description

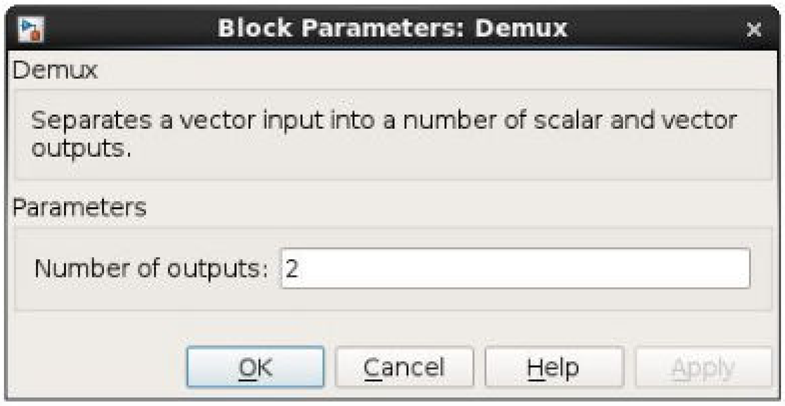
The Demux block input signal can be a scalar, a vector, a row matrix (1xN), or a column matrix (Nx1). The block splits elements in the input signal into scalar and vector type output signals according to the width of each output port starting from the first port at the top right.

The number of output ports and the port widths are configurable using the Number of outputs block parameter,

You can specify the width of each port or it can be dynamically computed by the block based on how the parameter value is provided. Refer to the Parameters section below for more details.

When the value of the Number of outputs parameter is changed, the output ports are either added or removed starting from the last port at the bottom right.

Figure 43: **Demux Parameter Dialog**



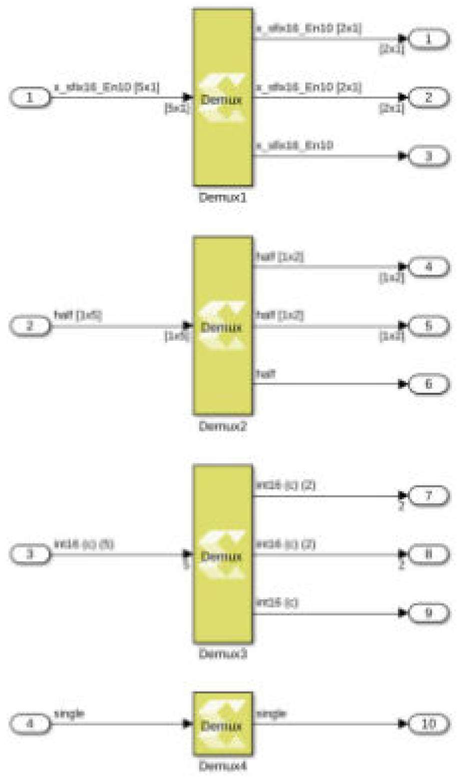
Demux

Separates a vector input into a number of scalar and vector outputs.

Parameters

Number of outputs: 2

Figure 44: **Demux Diagaram**



The output ports are added/removed starting from the last port at the bottom right.

Data Type Support

- **Inputs:**
  - The block has one input port.
  - The input signal can be a scalar, a vector (N), a row matrix (1xN), or column matrix (Nx1), where N is the width of input signal.
  - The Demux block supports all native data types (double, single, uint8, int8, uint16, int16, uint32, int32, and boolean), and Model Composer supported half and fixed-point data types.
  - The block supports input data in real or complex numeric type.
- **Outputs:**
  - The number of outputs for the block is specified using the Number of outputs block parameter.
  - The value of the block parameter can be a finite positive integer, P, or an array of integers. The numbers in the array are used to decide the number of outputs as well as the width of each output signal.

- An output signal can be configured as a scalar, a vector (M), a row matrix (1xM), or a column matrix (Mx1) where M is less than or equal to the width of the input signal.
- The sum of widths of all output signals is ensured to be the same as the input signal width.
- The data type and the numeric type (real or complex) of output signals are the same as those of the input signal.

Parameters

- **Number of outputs:**

  This parameter takes number of outputs in several ways. Depending upon the parameter value, the output ports are added/removed starting from the last port at the bottom right.

Table 45: **Number of Outputs**

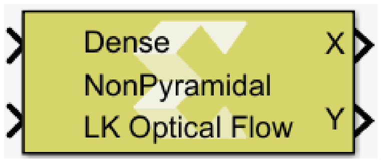
Option	Choices	Description
1	2	The block icon is initially created with two output ports. The input signal width is equally divided between the two outputs. If the input signal width is an odd number, then any remainder of the width is assigned to the first port at the top right.
2	P	A finite integer value representing the number of output ports. P must be greater than 0. The block icon is redrawn with the specified number of output ports. The widths of the output ports are dynamically computed by the block as follows: The width of the input is equally divided among the outputs. Any remainder of the width is assigned, one each, to the outputs starting from the first port at the top right. For example, if N is 3, and the width of the input is 14, then the first output is assigned with the first 5 input elements, the second output is assigned with the next 5 input elements, and the third output is assigned with the last 4 input elements.
3	[P]	A finite positive integer in square brackets is treated just like option 2 above. Here, the number of outputs will be P.
4	[-1 -1 -1]	The block icon is redrawn with 3 output ports. Here -1 means that the width of the particular output port needs to be computed in the same way as it is explained in the option 2 above.
5	[3 -1 -1]	The block icon is redrawn with 3 output ports. You specify the width of the first output, and Model Composer computes the widths of the second and the third outputs. For example, if width of the input is 8, and the first output width is 3, then the remaining width of 5 is divided between the second and the third outputs. This results in the widths of the second and the third outputs to be set to 3 and 2 respectively.
6	3 3 1	The block icon is redrawn with 3 output ports. The width of each output port is already specified by the user. The sum of the width of the outputs is 7. The width of the input must be 7, otherwise, an error message appears.

Dense Non-Pyramidal LK Optical Flow

Compute optical flow between two adjacent frames using the Lucas-Kanade method. The outputs are horizontal (X) and vertical (Y) components of the flow vectors.

Library

Computer Vision



Description

Optical flow is a pattern of motion of objects in an image between two consecutive frames. It is created due to the movement of objects and/or the camera in the time duration between the frames. To identify the movement of a pixel, a window around the pixel is considered with an assumption that all pixels in the small window have the same motion and their intensities do not change between two consecutive frames.

The Lucas-Kanade method is used to form the small window around each candidate pixel. The pixel movements are calculated in X and Y directions as separate flow vectors.

The Dense Non-Pyramidal LK Optical Flow block outputs are 2D matrices containing displacement vectors in horizontal (X) and vertical (Y) directions, respectively.

Data Type Support

- The inputs must be two consecutive frames of 2D grayscale image in uint8 data type.
- Each output is a 2D matrix of 32-bit floating point data type containing the displacement vectors in X or Y direction.

Parameters

Window width

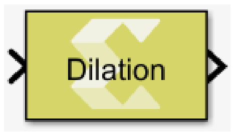
The **Window width** parameter specifies the width of the window that is formed around a pixel for the Lucas-Kanade method. This value must be a positive odd integer.

Dilation

Morphological function replaces each element of the input signal with the maximum intensity value in its 3x3 neighborhood. The function accepts and produces grayscale images (2-D matrices of real uint8 data type).

Library

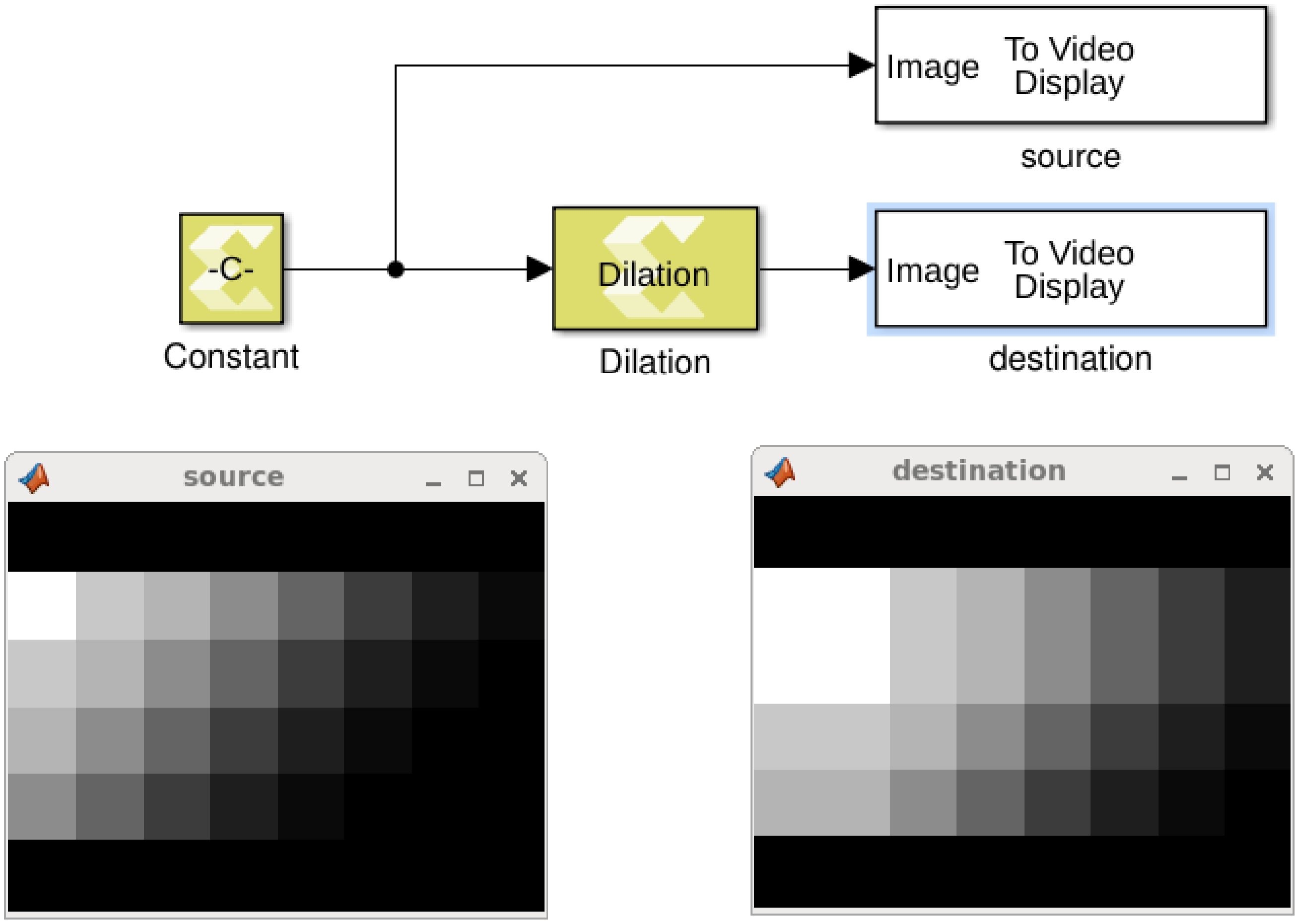
Computer Vision



Description

The Dilation block implements morphological dilation on a single channel (grayscale) 2-D image matrix, using a structuring element of size 3x3 pixels. For each pixel of the input image the block uses the structuring element centered around the source pixel to select neighboring pixels and choose the one with maximum intensity for the output.

Figure 45: **Dilation Block**



Data Type Support

The block only supports real-valued and unsigned 8-bit integers as the source and output data type. The input signal must be a 2-D matrix.

Parameters

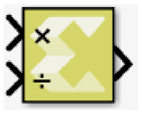
The Dilation block has no parameters to set.

Divide

Element-wise division

Library

Math Functions / Math Operations



Description

The Divide block has two input ports and one output port. The output signal (quotient) is the result of element-wise division of the first input (dividend) by the second input (divisor).

Data Type Support

The data types of the dividend and divisor can be any integer, fixed point or floating point data type. Boolean dividends or divisors are not supported. The input signals can be real or complex. The input signals can be scalars, vectors or matrices. If neither input signal is scalar, they must either be vectors of the same length or matrices with the same number of rows and columns. When the output is integer or fixed point data type, the result is truncated to zero.

The output data type is chosen for maximal alignment with Vivado® HLS. If the dividend and divisor are fixed point types and the fixed point parameters of the dividend are W1 (word length), FW1 (fractional length), and S1 (signedness), and the fixed point parameters of the divisor are W2, FW2, and S2, then the fixed point parameters of the quotient are as follows:

- S = S1 || S2
- FW2 ≥ 0
    - W = W1 + FW2 + S2
    - FW = FW1

- FW2 < 0
  - W = W1 +S2
  - FW = FW1 + FW2

Parameters

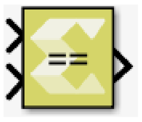
The Divide block has no parameters.

Equals

Perform element-wise equal to relational operation on the inputs. The top input corresponds to the first operand.

Library

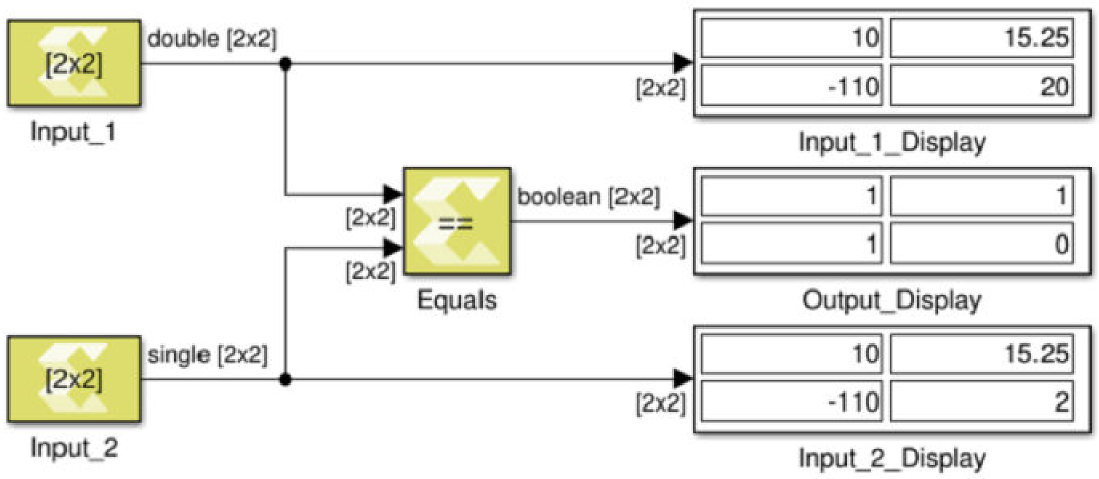
Relational Operations



Description

The Equals block performs element-wise comparison of inputs for equality. The block has two input ports and one output port. The output is true if the first input is equal to the second input.

Figure 46: **Equals Block**



Data Type Support

Data types accepted at the inputs of the block are:

- The block supports all data types supported by Xilinx® Model Composer.
- The block supports inputs having different data types. The output data type is always Boolean.
- The block supports mixed dimensions for inputs, where one input is a scalar, and the other input is a vector/matrix. The scalar value is compared with each element of the multi-dimensional input value for equality. The output has the same dimension as the multi-dimension input.
- If both inputs are non-scalar then their dimensions must match.

The output data type is always Boolean.

Parameters

The Equals block has no parameters to set.

Erosion

This block erodes away high intensity pixels from the regions where high intensity and low intensity pixels form the boundaries in the input image.

Library

Computer Vision

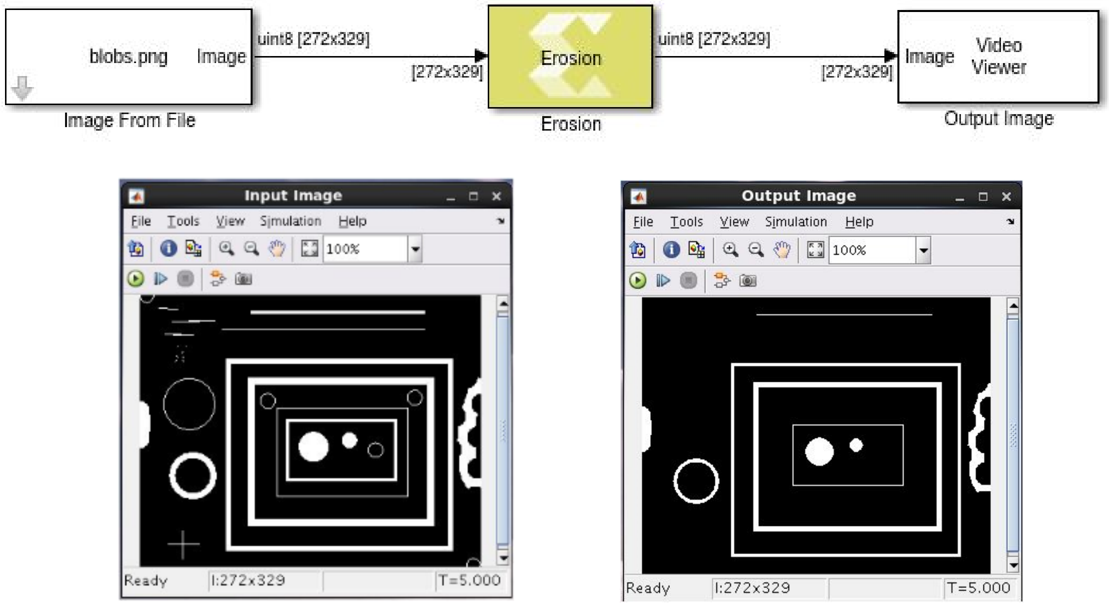
Description

The Erosion block supports the erosion algorithm of mathematical morphology. The algorithm creates a 3x3 pixels window around a pixel in the input image and replaces the pixel intensity with the minimum intensity found in the window.

This process is repeated for all the pixels in the input image. The resulting output image has high intensity pixels replaced with the low intensity pixels in the regions transitioning from lighter to darker shades.

The example below shows the effect of the erosion operation. Notice in the output image that some of the redundant high intensity (white) pixels are replaced by the low intensity pixels.

Figure 47: **Erosion Block**



Data Type Support

The block supports a 2-D grayscale or monochrone image in uint8 data type.

Parameters

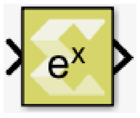
The Erosion block has no parameters to set. Only a fixed scanning window (structuring element) size of 3x3 is supported.

Exp

Perform an element-wise exponential value of the input

Library

Math Functions / Math Operations



Description

The Exp block returns the exponential e^x for each element in array `x`. The block supports all data types except Boolean. The input can be scalar, vector or matrix.

Data Type Support

Data types accepted at the inputs of the block are:

- Dimension : Input can be scalar, vector, or matrix.
- Data Types: Input supports signal of integer, fixed point, and floating point data type. It does not support Boolean inputs.
- Complex numbers are not supported.

The output has the same dimension and type as the input.

Parameters

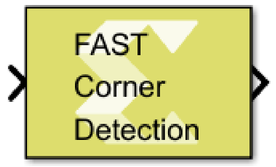
The Exponential block has no parameters to set.

FAST Corner Detection

Apply the FAST corner detection algorithm to a grayscale image (uint8) and mark the corner locations. The output matrix is the same size as the input matrix with corner locations marked with 255 and the rest of the locations marked with 0.

Library

Computer Vision



Description

The Features from Accelerated Segment Test (FAST) algorithm is an efficient corner detection algorithm, faster than most other corner feature detectors. It compares the intensity of each pixel in the image to that of 16 equidistant pixels in its Bresenham circle neighborhood. If the intensity of a continuous set of 9 or more of these pixels are found to be either above or below the intensity of the candidate pixel by a given threshold, then the candidate pixel is added to a list of detected corners. If the **Non-maximum suppression** option is enabled, the weaker corner candidates are pruned from the identified corners.

The function can be used for both still images and videos. You specify the intensity **Threshold** to help detect the corner candidates. If a pixel is marked as a corner, it is assigned intensity 255, otherwise it is assigned intensity 0.

Data Type Support

The input to the block is a grayscale image in the form of a real-valued 2D matrix of data type uint8.

The output is a grayscale image of data type uint8.

Parameters

Threshold

Threshold is the intensity difference between the candidate pixel and its neighbors. Usually it is taken at approximately 20. The **Threshold** parameter can be specified as an unsigned integer value between 0 and 255.

Non-maximum suppression

If the **Non-maximum suppression** box is checked, non-maximum suppression is applied to weed out weaker corners from the already identified corners.

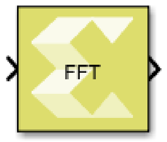
If the **Non-maximum suppression** box is unchecked, non-maximum suppression is disabled.

FFT

Perform Fast Fourier Transform (FFT) on the input data to produce full precision output data arranged in natural order. The block is configurable to scale the output data to avoid bit growth and to produce the output in a bit-reversed order. The FFT implementation can be optimized either for performance or for resources. The FFT transform length must be a power of 2 and between 2^3 and 2^16.

Library

Digital Signal Processing



Description

The FFT block computes an N-point forward Discrete Fourier Transform (DFT) across the first dimension of input array. The FFT transform length, N, is the same as the number of elements in the input array. The value of N must be a power of two and between 8 and 65536 (inclusive). The input data is a vector of N complex values, where each value is represented as dual (real and imaginary) two's-complement numbers. The block supports inputs only in fixed-point data type.

The FFT block output data can be produced in natural order or in bit-reversed order. The natural ordering of the output is achieved at the cost of either more memory or more run-time.

This block uses the Radix-2 decomposition with decimation in frequency (DIF) method for computing the DFT. An N-point FFT using Radix-2 decomposition has log2 (N) stages, with each stage containing N/2 Radix-2 butterflies. The output vector of N elements is represented by dual (real and imaginary) two's-complement numbers. The input data is always presented in natural order, whereas the output data can be either in bit-reversed order or in natural order.

Algorithm

The FFT is a computationally efficient algorithm for computing a Discrete Fourier Transform (DFT) on input size N.

The descrete Fourier transform (DFT) is defined by the formula:

$$X_k = \sum_{n=0}^{N-1} x_n e - \frac{2xi}{N} nk$$

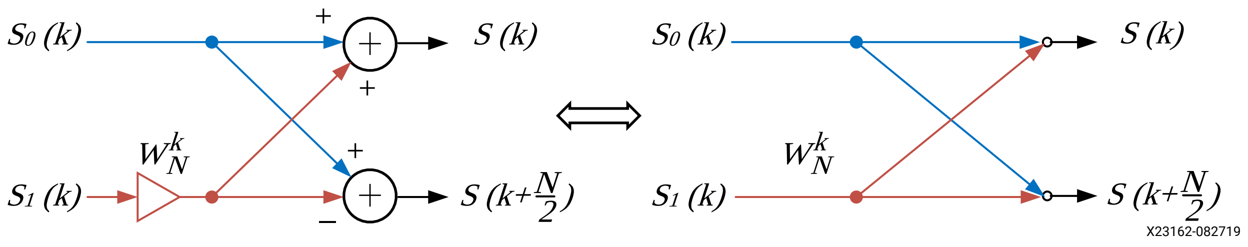
where *k* is an integer ranging from *0* to *N-1*.

The Radix-2 FFT algorithm proceeds by dividing the DFT into two DFTs of length N/2 each, and iterating as shown below.

$$S(k) = S_0(k) + W_N^k \, . \, S_1(k), \;\; k = 0 \dots \frac{N}{2} - 1;$$

$$S = \left(k + \frac{N}{k}\right) - W_N^k \, . \, S_1(k), \;\; k = 0 \dots \frac{N}{2} - 1.$$

$$W_N^k = \exp\left(-j \, . \, \frac{2\pi}{N} \, . \, k\right), \;\; k = 0 \dots N - 1,$$



Data Type Support

- **Inputs :**

  The input can be a vector (N) or column matrix [1 x N]. The number of elements in the input must be a power of 2 and must be between 8 and 65536 (inclusive).

The input is assumed to be in natural order.

FFT block only supports complex type input data represented using signed fixed-point data type. The requirements to have input data as complex numbers is due to intrinsic FFT algorithm and not due to its implementation.

- **Outputs:**

  The output is a complex type numbers with the same dimensions and the same data type as the input.

  The block produces output data that is unscaled, and with full precision or scaled to avoid bit growth. The bit growth is avoided by selecting **Divide output by (2 * transform length)** parameter.

  The output data can be arranged in either natural order or a bit-reversed order.

Parameters

- **Output in bit-reversed order:** Selection of this option produces the output data in bit-reversed order instead of the natural order. The natural ordering of the data results in more memory usage in Pipelined Streaming I/O architecture or longer run-time in Radix-2 Burst I/O architecture.
- **Divide output by (2 * transform length):** Selection of this option ensures that the output data is scaled to avoid bit growth compared to the input data. The output scaling is achieved by dividing the data by (2 * transform length).
- **Optimize for Performance or Resources:** The FFT algorithm is implemented, by default, using Pipelined Streaming I/O architecture to improve performance. This higher performance is achieved at the cost of more resources. The FFT can be implemented using Radix-2 Burst I/O architecture to optimize for the resources.

For more information about RTL co-simulation for this block, see RTL IP-Based Blocks.

FIR

Independently filters each input channels using a FIR Filter.

Library

Digital Signal Processing

Description

The Discrete FIR Filter block treats each element of the input as an independent channel (sample-based processing), and filters each channel independently. The block implements a static filter with fixed coefficients. The output dimensions equal the input dimensions, and the output data type is based on the input data type and the filter coefficient.

FIR block computes the following mathematical equation:

$$\mathrm{y(k)} = \sum_{n=0}^{N-1} a(n)x(k-n) \quad k = 0,1, \quad \ldots$$

Where `N` is the number of filter coefficients, `a(n)` are filter coefficients, `x` is the input, and `y` is the output.

Data Type Support

- **Dimension:**

  Input can be either a scalar, a matrix, or a vector.

- **Data Types:**

  The input can only be fixed-point data type.

- **Complex Number Support:** No.

The output has the same dimension as that of the input with bit growth based on filter coefficients data width.

Parameters

- **Coefficients parameter :**

  Specifies the filter coefficients.

- **Coefficients precision :**

  - **Use optimal value:** If this option is checked, then filter coefficients are represented with full precision.
  - **Word Length and Fraction Length:** Values for word length and fractional length parameters are used to represent filter coefficients when Use optimal value option is unchecked.

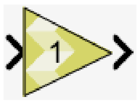
For more information about RTL co-simulation for this block, see RTL IP-Based Blocks.

Gain

Element-wise multiplication of the input by a constant gain factor

Library

Math Functions / Math Operations



Description

The Gain block multiplies the input signal by a constant gain factor.

You can specify the data type of the gain constant and built-in type promotion rules apply to determine the output data type. Alternatively, the output data type can be made the same as the input type. In case of integer overflow the block supports the option to saturate output values at the output type limits.

The block warns or errors out when an integer output overflows during simulation. To configure, select **Simulation → Model Configuration Parameters → Diagnostics → Data Validity** for your model in the Simulink Editor, then set the **Wrap on overflow** or **Saturate on overflow** parameter.

Data Type Support

The block supports all data types except Boolean.

Data type support for the block is:

- The input can be a scalar, vector or matrix.
- If the input is a vector or matrix and the **Gain** is a scalar, the scalar value will apply to all the elements of the input.
- If neither the input nor the gain constant are scalar the dimensions of the input and the gain constant must match.

The output is complex if either the **Gain** constant or the input is complex.

Parameters

Gain

Specifies the constant gain factor. The **Gain** can be any valid MATLAB expression that evaluates to a real or complex scalar, vector, or matrix.

Gain data type

This parameter specifies the type of conversion to be applied to the **Gain** factor constant before multiplication. If **fixed** is specified more parameters are available.

Settings for the **Gain data type** parameter are:

Table 46: **Gain Data Type Parameter**

Setting	Description
double	double precision floating-point
single	single precision floating-point
int8	8-bit signed integer
uint8	8-bit unsigned integer
int16	16-bit signed integer
uint16	16-bit unsigned integer
int32	32-bit signed integer
uint32	32-bit unsigned integer
fixed	fixed-point
half	half precision floating-point
data type expression	

Unless the **Output data type same as input** parameter is enabled, the output data type will be a function of the input type and the specified **Gain data type**.

- If either input or gain types are floating-point (double, single, or half), the output type will be floating-point. If both are floating-point, the output type will be the larger of both. The smaller type will be promoted to the larger before the operation.
- Otherwise, if either of input or **Gain data type** are fixed-point, the output type will be fixed-point with a bit width sufficient to hold the full output result. The other type (input or **Gain data type**) will be promoted to its equivalent fixed-point type.
- Otherwise, the input and **Gain data type** are integers. The output type will be the larger of either input or Gain data type, and will be a signed integer if either one is signed.

Output data type same as input

This parameter specifies the way the output data type is determined.

If disabled (unchecked) the output type is computed via built-in type promotion rules. If enabled (checked), the output data type is the same as the input type.

Saturate on integer overflow

This parameter specifies the behavior in case of integer overflow. By default the option is disabled and overflow would result in value wrap. With the option enabled, integer overflow gets mitigated by saturation at the limits of the output data type.

Settings for the **Saturate on integer overflow** parameter are:

Table 47: **Saturate On Integer Overflow Parameter**

Setting	Description
Unchecked	Wrap around
Checked	Saturation

When overflow is detected, the Diagnostic Viewer displays messages that depend on the diagnostic action you specify in the Simulink Editor. To configure, select **Simulation → Model Configuration Parameters → Diagnostics → Data Validity** for your model in the Simulink Editor, then set the **Wrap on overflow** or **Saturate on overflow** parameter.

Gaussian Blur

Apply a Gaussian blur function to an unsigned 8-bit integer grey scale image

Library

Computer Vision

Description

The Gaussian Blur block applies a spatial low-pass filter of size 3x3, 5x5, or 7x7 to the input image. The filter kernel coefficients are computed based on the parameter **Sigma**, denoting the variance of the Gaussian kernel according to this formula:

$$G_0(x, y) = e^{\frac{-(x-\mu_x)^2}{2\sigma_x^2} + \frac{-(y-\mu_y)^2}{2\sigma_y^2}}$$

In the case of this function the x- and y-variances are the same as the **Sigma** parameter. The mean values μ_x and μ_y are both zero.

Data Type Support

The Gaussian Blur block accepts grayscale images (real-valued matrices) of type uint8 as input. The output has the same type and dimensions as the input.

Parameters

Filter width

The **Filter width** parameter determines the kernel size.

Settings for the **Filter width** parameter are:

Table 48: **Filter Width Parameter**

Setting	Description
3x3	Sets the filter kernel size to 3x3 coefficients.
5x5	Sets the filter kernel size to 5x5 coefficients.
7x7	Sets the filter kernel size to 7x7 coefficients.

Sigma

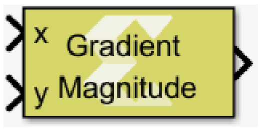
The **Sigma** parameter is a positive scalar value of either single or double precision floating point type.

Gradient Magnitude

Compute the element-wise magnitudes of an array of vectors given their orthonormal basis components (x,y), based on either Manhattan distance (L1-Norm), or Euclidean distance (L2-Norm).

Library

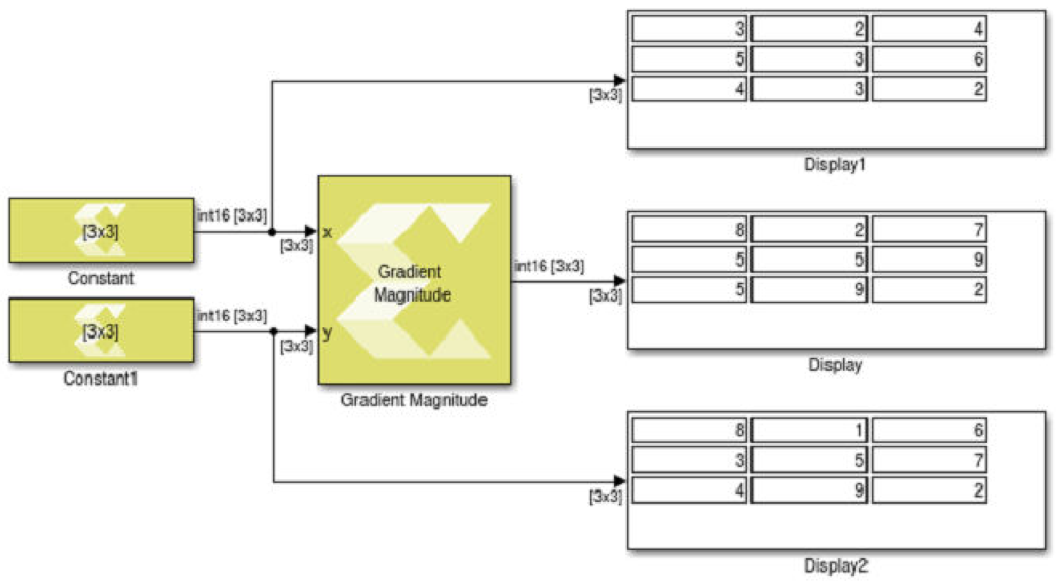
Computer Vision

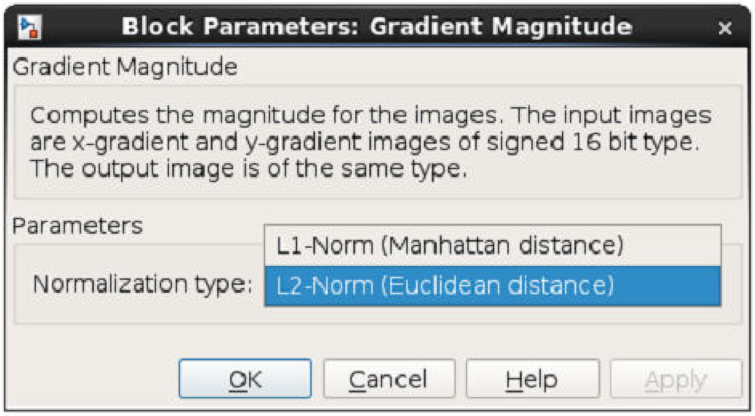


Description

This block takes as input two equally-sized matrices x and y, where each element represents one component of a 2D vector in an orthonormal basis vector space. For each array element at row j and column k, the block computes the magnitude of the vector (X_{jk},Y_{jk}), based on either Manhattan distance (L1-Norm), or Euclidean distance (L2-Norm).

Figure 48: **Gradient Magnitude Block**





Block Parameters: Gradient Magnitude

Gradient Magnitude

Computes the magnitude for the images. The input images are x-gradient and y-gradient images of signed 16 bit type. The output image is of the same type.

Parameters

Normalization type: L1-Norm (Manhattan distance) [✓ L2-Norm (Euclidean distance)]

Data Type Support

The block accepts two real-valued 2-D matrices of the same dimension, each of signed 16-bit integer type.

The output is of type int16 as well, and has the same dimensions as both inputs.

Parameters

Normalization type

The **Normalization type** parameter determines the distance metric used to compute the magnitude.

Settings for the **Normalization type** parameter are:

Table 49: **Normalization Type Parameter**

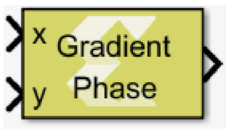
Setting	Description
L1-Norm (Manhattan distance)	L1-Norm uses the Manhattan distance metric and computes the magnitude as the sum of the component's absolute value: d = |x| + |y|.
L2-Norm (Euclidean distance)	L2-Norm uses the Euclidean distance metric and computes the magnitude as the square root of the sum of squares of the components: d = sqrt(x^2+y^2).

Gradient Phase

Compute the element-wise phase angle of an array of vectors given their orthonormal basis components (x,y)

Library

Computer Vision



Description

This block takes as input two equal sized matrices x and y, where each element represents one component of a 2D vector in an orthonormal basis vector space. For each array element at row j and column k the block computes the phase angle of the vector (x_{jk},y_{jk}), in radians or degrees.

Data Type Support

The block accepts two real-valued 2-D matrices of the same dimension, each of signed 16-bit integer type.

The output is either of type x_sfix16_En12 (for phase angles in radians), or x_sfix16_En6 (for phase angles in degrees). The output has the same dimensions as the two inputs.

Parameters

Phase angle unit

This parameter determines the output format for the phase angle.

Settings for the **Phase angle unit** parameter are:

Table 50: **Phase Angle Unit Parameter**

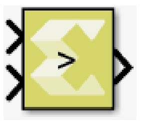
Setting	Description
Radians	The phase angles will be returned in **Output type** fixed-point format x_sfix16_En12 (a.k.a. Q4.12).
Degrees	The phase angles will be returned in **Output type** fixed-point format x_sfix16_En6 (a.k.a. Q10.6).

Greater

Performs element-wise greater than relational operation on the inputs. The top input corresponds to the first operand.

Library

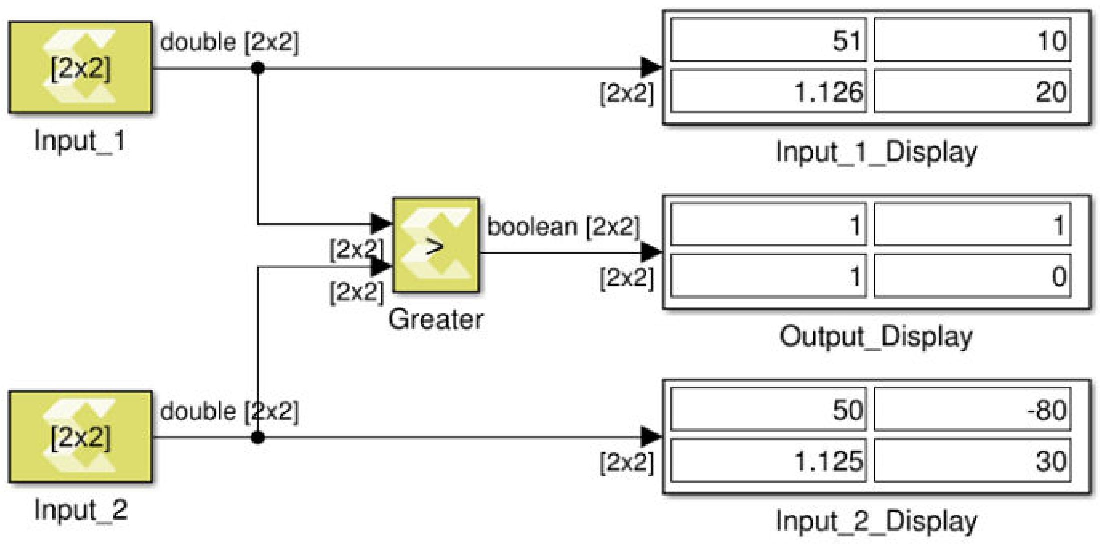
Relational Operations



Description

The Greater block has two input signals and one output signal. The block compares the two inputs using element-wise greater than relational operation. The first input corresponds to the top input port and the second input to the bottom input port. The dimension of the output signal matches the dimensions of the input signals. An element of the output signal is true if the corresponding element of the first input signal is greater than the corresponding element of the second signal. Otherwise the element is false.

Figure 49: **Greater Block**



Data Type Support

Data type support for the Greater block is:

- The data types of the input signals can be integer, fixed-point, Boolean, or floating-point data type.
- The input signals can be a scalars, vectors, or matrices. If both inputs are not scalar, their dimensions must match.
- The input signals must be real.
- The output signal is Boolean.
- The dimension of the output signal is scalar if both inputs are scalar. Otherwise it matches the dimensions of the non-scalar input.

Parameters

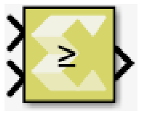
The Greater block has no parameters to set.

Greater Equals

Perform element-wise greater than or equal relational operation on the inputs. The top input corresponds to the first operand.

Library

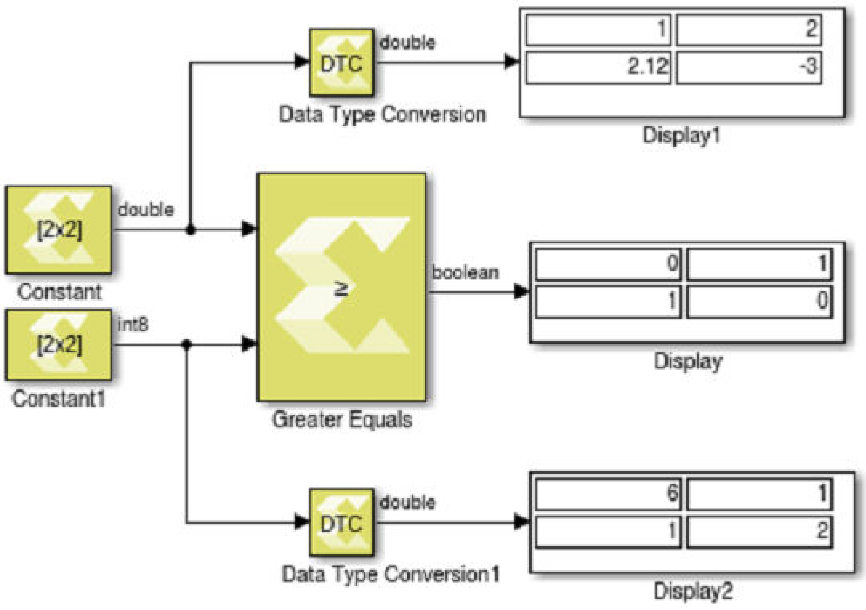
Relational Operations



Description

The Greater Equals block performs element-wise greater than or equal relational operation on the inputs. The upper input is the first input and the lower input is the second input. The block returns true if the first input is greater than or equal to the second input. The output equals 1 for true and 0 for false.

Figure 50: **Greater Equals Block**



Data Type Support

Data types accepted at the inputs of the block are:

- Data Types: Greater Equal block supports all data types supported by Model Composer (integer, floating-point, fixed-point, and Boolean).
- Dimension: The inputs can be scalar, vector, or matrix, or a combination of scalar, and matrix, or vector. If both the inputs are matrix or vector, they should have same dimension.
- Complex Number Support: No

The output is always Boolean.

Outputs for the different input types are:

Table 51: **Input/Output**

Inputs	Output
Both are scalar	Scalar
Both are vector	Vector of same dimension
Both are matrix	Matrix of same dimension
One is scalar and the other is vector or matrix	Dimension is that of the vector or matrix

Parameters

The Greater Equals block has no parameters to set.

Hermitian

Perform element-wise conjugate transpose operation on the input signal

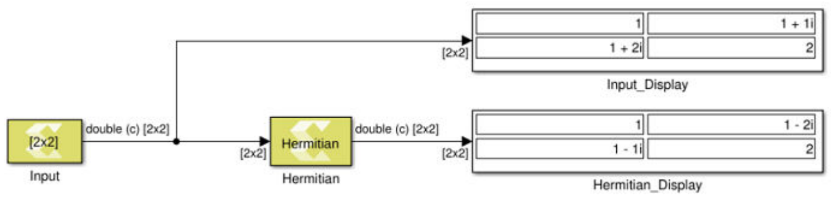
Library

Math Functions / Matrices and Linear Algebra

Description

The Hermitian block performs a conjugate transpose operation on the input signal.

Figure 51: **Hermitian Block**



Data Type Support

This block supports all data types supported by Xilinx® Model Composer. The input signal can be real or a complex number of scalar, vector, or matrix type.

Parameters

The Hermitian block has no parameters to set.

Histogram

Compute the histogram for 8-bit greyscale images

Library

Computer Vision

Description

The Histogram block takes a real-valued unsigned 8-bit greyscale image as input and computes the histogram with a fixed set of 256 bins. The output will be a vector of type uint32 and of size 256, containing the raw pixel count of each intensity value.

Data Type Support

The only supported input data type is uint8. Complex numeric type is not supported.

Parameters

The Histogram block has no parameters to set.

Histogram Equalization

Improve the contrast of an image (uint8 data type) supplied on the input ports by adjusting intensity values such that the resulting image has a more uniform distribution of the intensity values.

Library

Computer Vision

Histogram
Equalization

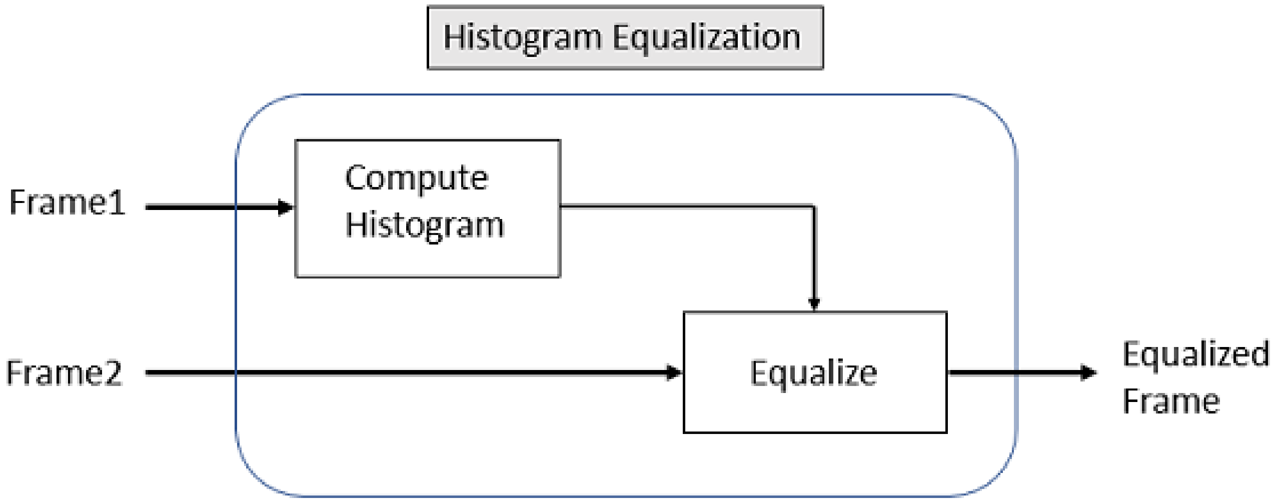
Description

A histogram operation is performed on an image to show the occurrences of each intensity value in the image. When the intensity values are not well distributed it results in a too dark or too bright image.

Histogram equalization is a technique used to enhance contrast for the input image (uint8 data type) by adjusting intensity values. The technique results in intensity values spread over the whole range of intensities. The output image has the same size and type as the inputs.

This process is completed in two stages. First, a histogram is computed for the image passed through the first input port. The second stage equalizes the image passed through the second input port using the histogram computed on the earlier frame. This approach minimizes memory utilization in the hardware.

Figure 52: **Histogram Equalization**



Data Type Support

The block supports a 2-D grayscale or monochrome image in uint8 data type.

Parameters

The Histogram Equalization block has no parameters to set.

HOG Descriptor

Outputs a vector of oriented gradients for the input image (uint8 data type). The input image can be a gray scale or a color image in RGB format. The output is a vector of N feature descriptors in uint32 data type.

Library

Computer Vision

Description

A feature descriptor is a representation of an image that simplifies the image by extracting useful information, and throwing away extraneous information. In the Histogram of Oriented Gradients (HOG) feature descriptor technique, the histograms of oriented gradients (the magnitude and directions of gradients) are used as features for object detection. This technique is widely used for the pedestrian detection in computer vision.

The technique counts occurrences of gradient orientations in a localized patch of an image. It starts by dividing the image into overlapping patches (windows). Each window is subdivided into consecutive regions called cells. The oriented gradients are computed for the pixels within each cell and stored in the form of a histogram. The resultant descriptor vector is the concatenation of these histograms.

Following are some of the details for this block.

- The patch is an overlapping window of 64x128 pixels, where window height is 128 pixels, width is 64 pixels, and the window overlap (stride) is 8 pixels.
- The input image height and width must be divisible by 8. The input matrix must not be less than the window size.
- Each window is subdivided into connected 8x8 pixel cells.
- A 9-bin histogram for each cell is stored in an array of 9 numbers to obtain a compact representation.
- For gray scale image, the block takes one input.
- The output is a feature descriptor vector of length N.
- For color image in RGB format, the block takes three inputs, one each for R, G, and B channels, where the R channel data is passed at the first input port.

Data Type Support

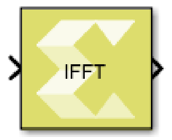
- Input matrix data type: uint8
- Output vector data type: uint32

IFFT

Performs the Inverse Fast Fourier Transform (IFFT) on the input data to produce full precision output data arranged in natural order. You can configure this block to scale the output data and avoid bit growth. You can also produce the output in a bit-reversed order. You can optimize the IFFT implementation either for performance or for resources. The IFFT transform length must be a power of 2 and between 2^3 and 2^{16}.

Library

Digital Signal Processing



Description

The IFFT block computes an N-point inverse Discrete Fourier Transform (DFT) across the first dimension of an input array. The IFFT transform length, N, is the same as the number of elements in the input array. The value of N must be a power of two and between 8 and 65536 (inclusive). The input data is a vector of N complex values, where each value is represented as dual (real and imaginary) two's-complement numbers. This block supports inputs only of fixed-point data type.

The IFFT block output data can be produced in natural order or in bit-reversed order. The natural ordering of the output is achieved at the cost of either more memory or more run-time.

The IFFT block uses the Radix-2 decomposition with decimation in frequency (DIF) method for computing the inverse DFT. An N-point IFFT using Radix-2 decomposition has log2 (N) stages, with each stage containing N/2 Radix-2 butterflies. The output vector of N elements is represented by dual (real and imaginary) twos-complement numbers. Input data is always presented in natural order, whereas, the output data can be either in bit-reversed order or in natural order.

Algorithm

The IFFT is a computationally efficient algorithm for computing an inverse Discrete Fourier Transform (DFT) of sample size N.

The inverse Discrete Fourier Transform (DFT) is defined by the formula:

$$X_n = \sum_{k=0}^{N-1} x_k e^{-\frac{2xi}{N}nk},$$

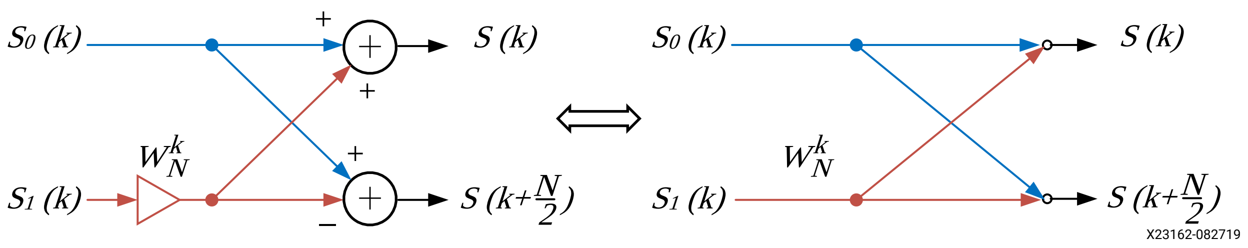
where n is an integer ranging from 0 to *N* - 1.

The Radix-2 N-point IFFT algorithms proceed by dividing the inverse DFT into two inverse DFTs of length N/2 each, and iterating as shown below.

$$S(k) = S_0(k) + W_N^k \,.\, S_1(k),\ k = 0 \ldots \frac{N}{2} - 1;$$

$$S\left(k + \frac{N}{2}\right) = S_0(k) - W_N^k \cdot S_1(k),\ k = 0 \ldots \frac{N}{2} - 1.$$

$$W_N^k = \exp\left(-j \,.\, \frac{2\pi}{N} \,.\, k\right),\ k = 0 \ldots N - 1,$$



Data Type Support

- **Inputs:**

  The input can be a vector (N) or row matrix [1 x N]. The number of elements in the input must be a power of 2 and must be between 8 and 65536 (inclusive).

  The input is assumed to be in natural order.

  IFFT block only supports complex type input data represented using signed fixed-point data type. The requirement to have input data as complex numbers is due to the intrinsic IFFT algorithm and not due to its implementation.

- **Outputs:**

  The output is a complex type number with the same dimensions and the same data type as the input.

  The block produces output data that is unscaled and with full precision, or scaled to avoid bit growth. To avoid bit growth select the **Divide output by (2 * transform length)** parameter.

The output data can be arranged either in natural order or in a bit-reversed order.

Parameters

- **Output in bit-reversed order:** Selection of this option produces the output data in bit-reversed order instead of the natural order. The natural ordering of the data results in more memory usage in Pipelined Streaming I/O architecture or longer run-time in Radix-2 Burst I/O architecture.
- **Divide output by (2 * transform length):** Selection of this option ensures that the output data is scaled to avoid bit growth compared to the input data. The output scaling is achieved by dividing the data by (2 * transform length).
- **Optimize for Performance or Resources:** The IFFT algorithm is implemented, by default, using Pipelined Streaming I/O architecture to improve performance. This higher performance is achieved at the cost of more resources. You can implement the IFFT block using Radix-2 Burst I/O architecture to optimize for the resources.

Integral Image

Compute the integral image of the input. Each output pixel is the sum of the corresponding pixel in the input image and the pixels above and to the left of it.

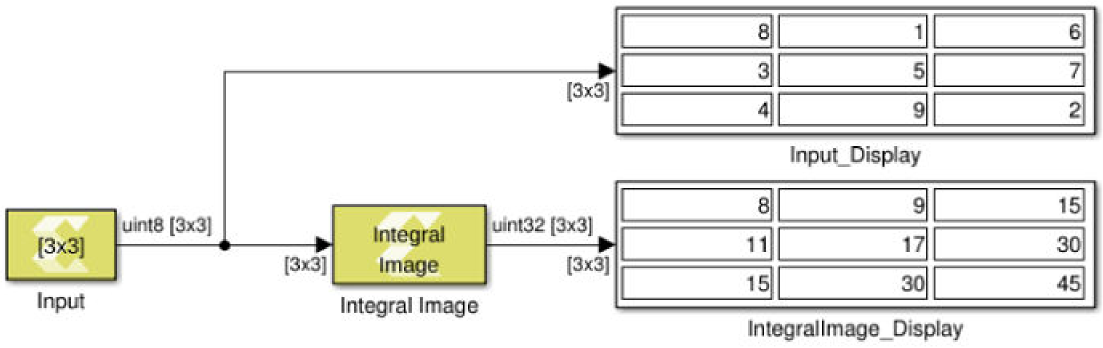
Library

Computer Vision

Description

The Integral Image block computes the integral image of an input intensity image (represented by an M x N matrix in uint8 grayscale). This helps to rapidly calculate summations over image sub regions in constant time regardless of the region size. Each pixel in an integral image is the summation of the corresponding pixel in the input image and the pixels above and to the left of it. As an example of the usage of Integral image, refer to the paper, "Robust Real-Time Face Detection" by Paul Viola and Micheal J. Jones, where integral image is used for rapid computations of rectangle features for a face recognition application.

Figure 53: **Integral Image Block**



In this design, consider the center pixel 5 in the input is test pixel, we can see the pixels which are left and top to the test pixel are 3, 8, and 1. The Integral Image block replaces the pixel 5 of the input with pixel value 17 (which is a sum of 5, 3, 8, and 1) in the output.

Data Type Support

The input must be real valued and 2-dimensional data of type uint8. The output data type is uint32 and its dimension is same as that of the input.

Parameters

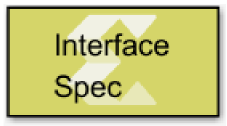
The Integral Image block has no parameters to set.

Interface Spec

Specify the RTL interfaces for a subsystem

Library

Tools



Description

The Interface Spec block allows you to control what RTL interfaces should be synthesized for the ports of the subsystem in which the Interface Spec block is instantiated. This affects only code generation and synthesis, when an RTL model (an IP) is synthesized by Vivado® HLS from the C++ model produced by Model Composer. The block has no effect on Simulink® simulation of your design. If your design does not have an Interface Spec block, Model Composer selects default interfaces for you. Interface synthesis is supported only to the subsystem for which you are generating C++ code. Therefore any Interface Spec blocks instantiated in subsystems nested within the subsystem for which you are generating C++ code are ignored.

The Interface Spec block is used as follows:

1. Instantiate the Interface Spec block in the subsystem for which you want to generate C++ code. The **Input ports** tab will be populated with one row for each input port of the parent subsystem. Similarly, the **Output ports** tab has one row for each output port of the parent subsystem.
2. Fill out the **Function Protocol**, **Input ports**. and **Output ports** tabs.

The information gathered by the Interface Specification block consists of three parts:

- The block-level Interface Protocol. This protocol is used to tell the IP when to start processing data. It is also used by the IP to indicate whether it accepts new data, or whether it has completed an operation, or whether it is idle.
- The port-level Interface Protocol for each input port of the parent subsystem.
- The port-level interface protocol for each output port of the parent subsystem.

The choice of port-level interface protocol should take into account the following considerations:

- Large array or matrix ports should use a streaming protocol such as AXI4-Stream, FIFO, or AXI4-Stream (video).
- Scalar ports can be implemented using any of the following protocols: Default, AXI4-Lite Slave, Constant, Valid Port, No protocol
- Video signals can be transported over an AXI4-Stream (video) interface. In this case you also need to specify the video format YUV 4:2:2, YUV 4:4:4, RGB or Mono. For video formats that have more than 1 color component, you also need to specify which port carries which color component and you need to assign the same name for the 'bundle' attribute for these (3) ports. All of the ports (either 3 or 1) that make up the video signal are implemented by a single AXI4-Stream interface that include start-of frame and end-of-line sideband signals. This follows the specifications described in the *AXI4-Stream Video IP and System Design Guide* (UG934).
- An AXI4-Lite Slave interface allows you to implement one or more ports.
- For further details refer to Interface Synthesis in the *Vivado Design Suite User Guide: High-Level Synthesis* (UG902).

The interface specification block currently supports subsystems with at most 8 input ports and 8 output ports.

Data Type Support

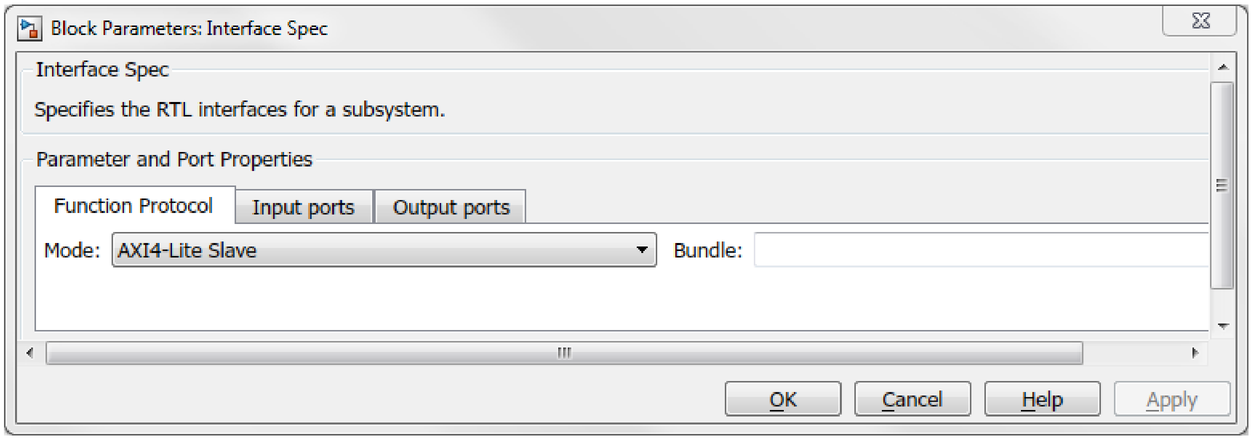
Data type support is not applicable to the Interface Spec block.

Parameters

The parameters for the Interface Specification block fall into the following groups.

- The parameters that apply to the function protocol. These are Mode, and Bundle. In the GUI dialog, these parameters appear in the 'Functional Protocol' tab.
- The parameters that apply to the Input ports. For each input port, there is 1 set of parameters Mode, Bundle, Offset, Video Format, and Video Component. In the Block Parameter dialog, these parameters appear in the 'Input ports' tab.
- The parameters that apply to the Output ports. For each output port, there is 1 set of parameters Mode, Bundle, Offset, Video Format, and Video Component. In the GUI dialog, these parameters appear in the 'Output ports' tab.

Figure 54: **Function Protocol Parameters**


Block Parameters: Interface Spec

Interface Spec

Specifies the RTL interfaces for a subsystem.

Parameter and Port Properties

Mode: AXI4-Lite Slave

Bundle:

Parameters on the **Function Protocol** tab are as follows:

- **Mode:**

  The **Mode** parameter specifies the block-level I/O protocol.

  Following are the settings for the **Mode** parameter.

Table 52: **Mode Parameter**

Setting	Description
AXI4-Lite Slave	Specifies AXI4-Lite Slave as the block-level I/O protocol.

Table 52: **Mode Parameter** *(cont'd)*

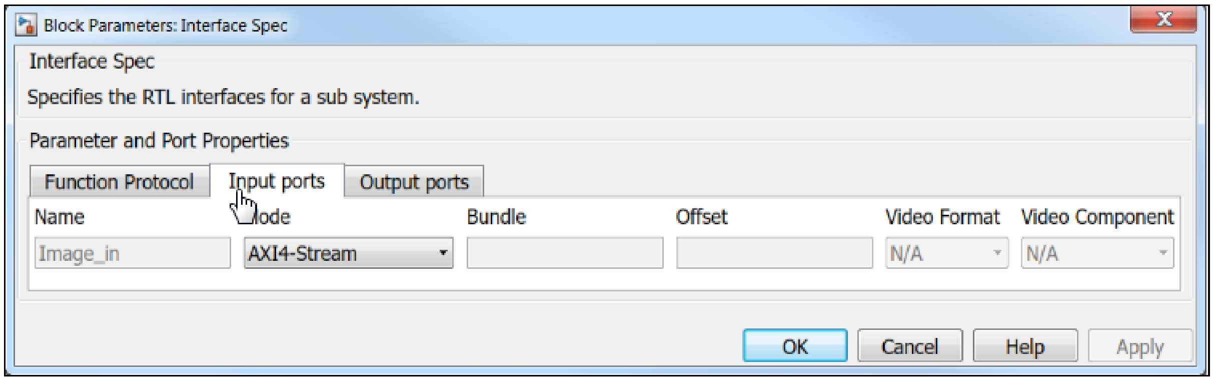
Setting	Description
Handshake	Specifies a handshake protocol as the block-level I/O protocol.
No block-level I/O Protocol	Specifies that there is no block-level I/O protocol.

The default choice for the function protocol is 'AXI4-Lite Slave'. However, if the DUT does not have any scalar ports then Handshake is selected as default function protocol.

- **Bundle:**

  The **Bundle** parameter is used in conjunction with the AXI4-Lite Slave interface to indicate that multiple ports should be grouped into the same interface. Enter a legal identifier in the C language (cannot contain spaces or special characters) for **Bundle**.

Figure 55: **Input Ports**


Block Parameters: Interface Spec

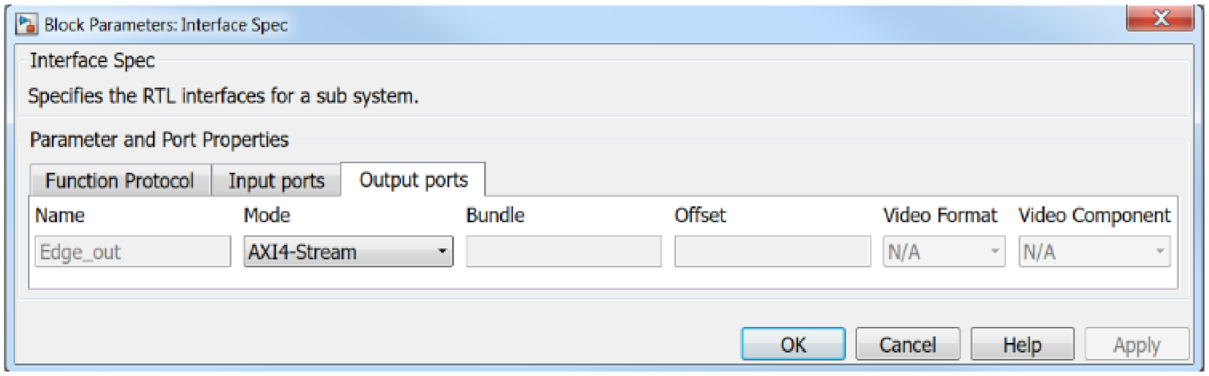
Interface Spec

Specifies the RTL interfaces for a sub system.

Parameter and Port Properties

Name	Mode	Bundle	Offset	Video Format	Video Component
Image_in	AXI4-Stream			N/A	N/A

Figure 56: **Output Ports**


Block Parameters: Interface Spec

Interface Spec

Specifies the RTL interfaces for a sub system.

Parameter and Port Properties

Name	Mode	Bundle	Offset	Video Format	Video Component
Edge_out	AXI4-Stream			N/A	N/A

Parameters on the **Input ports** and **Output ports** tabs are as follows.

- **Mode:**

  The **Mode** parameter specifies the I/O protocol for the input port or the output port.

Settings for the **Mode** parameter are:

Table 53: **Mode Parameter**

Setting	Description
Default	Specifies to use **AXI4-Lite Slave** if port is scalar, and use **AXI4-Stream** if the port is non-scalar.
AXI4-Stream	Specifies **AXI4-Stream** protocol.
AXI4-Stream (video)	Specifies **AXI4-Stream (video)** protocol. Allows you to specify **Bundle**, **Video Format**, and **Video Component** parameters.
AXI4-Lite Slave	Specifies **AXI4-Lite Slave** protocol. Allows you to specify **Bundle** and **Offset** parameters.
FIFO	Specifies a protocol for arrays whose elements are accessed in a sequential manner.
Valid port	Specifies a handshake protocol that only has a valid port.
Constant	Specifies a mode in which no I/O protocol is added to the port. The mode is intended for configuration inputs which only change when the device is in reset mode. This mode only applies to **Input ports**.
No protocol	Specifies that no I/O protocol is added to the port.
Block RAM	Specifies **Block RAM** interface protocol.

- **Bundle:**

  The **Bundle** parameter applies to the input ports or output ports and it is used in conjunction with the **AXI4-Stream (video)** interfaces that have more than one color component. In this case there should be one port for each color component and these ports should specify the same name for the **Bundle** attribute so that they will be grouped into the same **AXI4-Stream (video)** interface.

  The parameter is also used in conjunction with **AXI4-Lite Slave** interfaces to specify that ports with the same name for the **Bundle** attribute will be grouped into the same AXI4-Lite Slave interface.

  Enter a legal identifier in the C language (cannot contain spaces or special characters) for **Bundle**.

- **Offset:**

  The **Offset** parameter applies to the input ports or output ports and it is used in conjunction with the **AXI4-Lite Slave** interface. The parameter allows you to specify the address offset for a port within the **AXI4-Lite Slave** address map.

- **Video Format:**

  The **Video Format** parameter applies to the input ports or output ports and it specifies the color format for a video signal. It applies only to **AXI4-Stream (video)** interfaces. Options are **Mono**, **YUV 4:2:2**, **YUV 4:4:4**, and **RGB**.

- **Video Component:**

  The **VideoComponent** parameter applies to the input ports or output ports and it specifies the color component for a video signal. It applies only to **AXI4-Stream (video)** interfaces that use a **Video Format** with more than one color component. Options are **Mono**, **YUV 4:2:2**, **YUV 4:4:4**, and **RGB**.

  The **Video Component** selections for the different **Video Format** options are as follows.

Table 54: **Video Component Option**

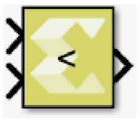
Video Format	Video Component Options
Mono	N/A
YUV 4:2:2	**Y**, **U**, **V**
YUV 4:4:4	**Y**, **U**, **V**
RGB	**R**, **G**, **B**

Lesser

Performs element-wise less than relational operation on the inputs. The top input corresponds to the first operand.

Library

Relational Operations



Description

The Lesser block has two input signals and one output signal. The block compares two inputs using element-wise lesser relational operation. The first input corresponds to the top input port and the second input to the bottom input port. The dimension of the output signal matches the dimensions of the input signals. An element of the output signal is true if the corresponding element of the first input signal is less than the corresponding element of the second signal; otherwise the element is false.

Data Type Support

Data types support for the Lesser block is:

- The data types of the input signals can be integer, fixed-point, boolean, or floating point data type.

- The input signals can be scalars, vectors, or matrices. If both inputs are not scalar, their dimensions must match.
- The input signals must be real.
- The output signal is boolean.
- The dimension of the output signal is scalar if both inputs are scalar. Otherwise it matches the dimensions of the non-scalar input.

Parameters

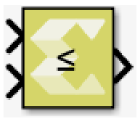
The Lesser block has no parameters to set.

Lesser Equals

Perform element-wise less than or equal relational operation on the inputs. The top input corresponds to the first operand.

Library

Relational Operations



Description

Performs element-wise less than or equal relational operation on the inputs. The block returns true (1) if the first input is less than or equal to the second input; returns false (0) otherwise.

Data Type Support

The Lesser Equals block supports inputs of native data types of MATLAB® and Ap_Fixed data type supported by Vivado® HLS. The output type is always Boolean. It does not support complex data types. The block supports inputs of scalar, vector, and matrix dimensions. When both inputs have non-scalar dimensions then the dimensions of the inputs must match each other.

Parameters

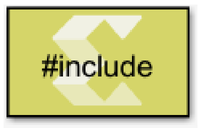
The Lesser Equals block has no parameters to set.

Library Function

Import user created C function as a block

Library

Library can be specified after you create the Library Function block.



Description

The Library Function block allows you to bring C or C++ models into Model Composer for block simulation and code generation. The block I/O interface is determined by the function declaration, which is auto-discovered by the tool. The Library function block is created when you run the `xmcImportFunction` script to specify the library function sources files and header search paths.

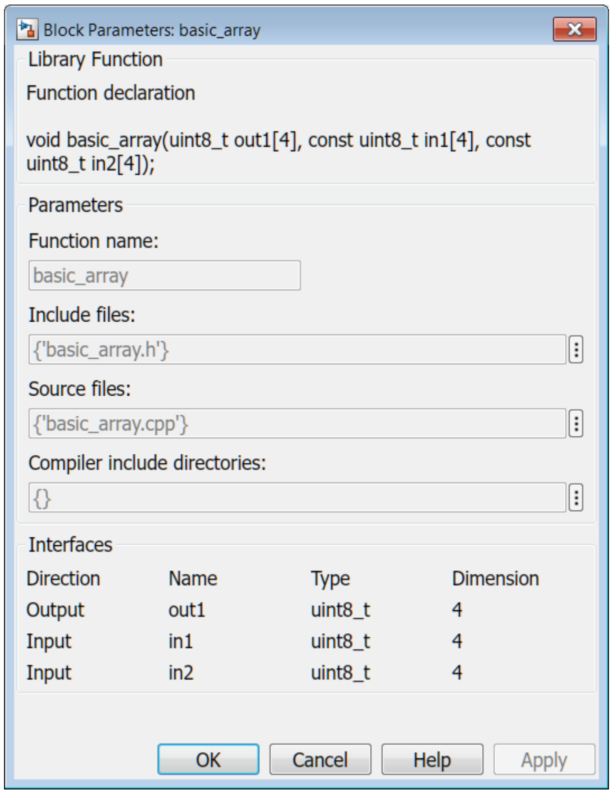
Data Type Support

You can import functions that have scalar, vectors, or matrices as function parameters. All data types, including fixed-point are supported. Complex values, real, and imaginary components, and phase angles are not supported.

Parameters

The block parameters dialog box for the Library Function block is shown below:

Figure 57: **Block Parameters**



Block Parameters: basic_array

Library Function

Function declaration

void basic_array(uint8_t out1[4], const uint8_t in1[4], const uint8_t in2[4]);

Parameters

Function name: basic_array

Include files: {'basic_array.h'}

Source files: {'basic_array.cpp'}

Compiler include directories: {}

Interfaces

Direction	Name	Type	Dimension
Output	out1	uint8_t	4
Input	in1	uint8_t	4
Input	in2	uint8_t	4

The dialog box indicates the settings for these parameters which were specified when the block was created:

- **Function name**
- **Include files**
- **Source files**
- **Compiler include directories**

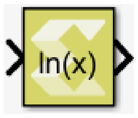
You cannot change these settings from the dialog box. To change these settings, you will have to recreate the block using the `xmcImportFunction` command.

Log

Compute element-wise natural logarithm of input

Library

Math Functions / Math Operations



Description

The Log block returns the logarithm value for the provided input. The block supports input of all data types except Boolean. The input can be scalar, vector, or matrix.

Data Type Support

Data types accepted at the inputs of the block are:

- Data Types: Input supports signals of integer, fixed-point, and floating-point data type. The block does not support Boolean inputs.
- Complex Number Support: No

  The output has the same dimension and type as the input.

Parameters

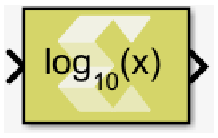
The Log block has no parameters to set.

Log10

Compute element-wise base 10 logarithm of input

Library

Math Functions / Math Operations



Description

The Log10 block returns the base 10 logarithm value for the input. The block supports input of all data types except Boolean. The input can be scalar, vector, or matrix.

Data Type Support

Data types accepted at the inputs of the block are:

- Data Types: Input supports signals of integer, fixed-point, and floating-point data type. The block does not support Boolean inputs.

- Complex Number Support: No

The output has the same dimension and type as the input.

Parameters

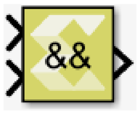
The Log10 block has no parameters to set.

Logical AND

Performs element-wise logical AND operation on inputs

Library

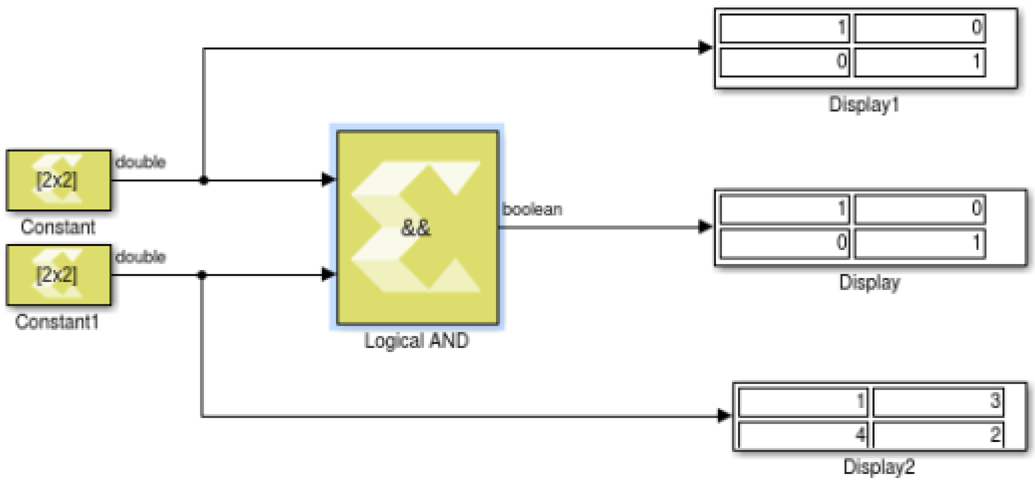
Logic and Bit Operations



Description

The Logical AND block has two input ports and one output port. The block performs an element-wise logical AND operation on the inputs and produces a Boolean result on the output.

Figure 58: **Logical AND Block**



Data Type Support

- The block supports inputs of different data types. The output data type is always Boolean.

- The block only supports real inputs.
- If one input is non-scalar type then the other input can be scalar type.
- If both inputs are non-scalar type then their dimensions must match. In case of non-scalar inputs, the output has the same dimension as the inputs.

Parameters

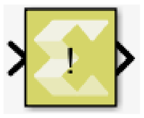
The Logical AND block has no parameters to set.

Logical NOT

Performs element-wise logical NOT operation on the input

Library

Logic and Bit Operations



Description

The Logical NOT block has one input port and one output port. The output is false if the input is a non-zero (true) value.

Data Type Support

- The Logical NOT block supports all data types supported by Model Composer.
- The block only supports real inputs.

Parameters

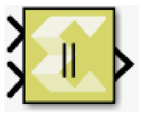
The Logical NOT block has no parameters to set.

Logical OR

Performs element-wise logical OR operation on inputs

Library

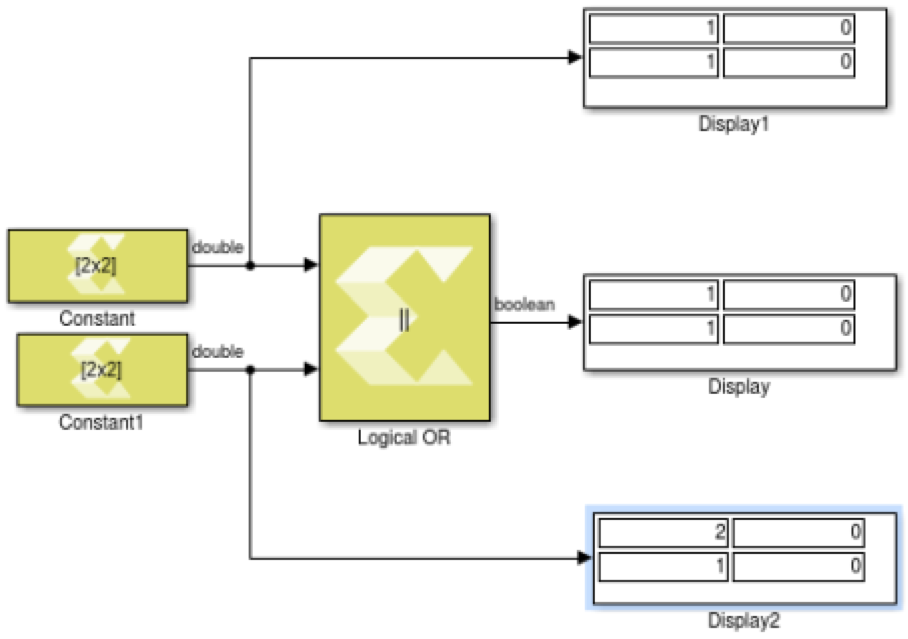
Logic and Bit Operations



Description

The Logical OR block has two input ports and one output port. The block performs an element-wise logical OR operation on the inputs and produces a Boolean result on the output.

Figure 59: **Logical OR Block**



Data Type Support

- The Logical OR block supports all data types supported by Model Composer.
- The block supports inputs of different data types. The output data type is always Boolean.
- The block only supports real inputs.
- The block supports scalar and non-scalar type inputs. If one input is non-scalar type then the other input can be scalar type. When both inputs are non-scalar type then their dimensions must match. In that case, the output has the same dimension as the inputs.

Parameters

The Logical OR block has no parameters to set.

Lookup Table

Perform one-dimensional lookup operation with an input index.

Library

Lookup Tables

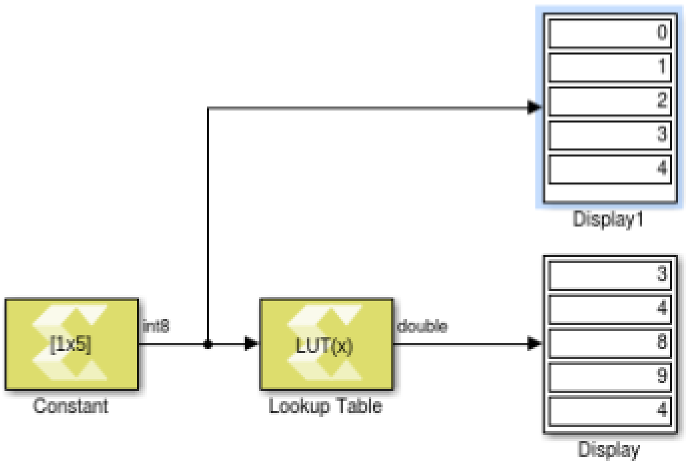
Description

The Lookup Table block implements a simple read-only memory block with an input index. The block maps input to an output value by looking up a table of values you define with a **Table data** parameter.

The input value is used as a zero-based index into the table data. The **Input bias** parameter is an offset to the index value (to support negative indexing). A **When input is out of range** parameter lets you specify the behavior of the block if the index value exceeds the valid table size range.

***Note*:** If the table size is not an exact power of 2, the block incurs additional hardware cost when it is implemented in a Xilinx device, due to a remainder calculation on the index.

Figure 60: **Lookup Table Block**



In the example above, the **Table data** setting for the Lookup Table block is [7 3 4 8 9 4 1 5] with an input bias of 1.

Data Type Support

The Lookup Table block accepts the following data types to represent scalar index value: int8, uint8, int16, uint16, int32, uint32, and fixed-point type. Fixed-point inputs is shifted appropriately to generate an integer index.

The output data type is same as the **Table data** parameter type. Inputs for indexing must be real, but table data can be complex.

Parameters

Table Data

This parameter accepts a 1-D vector of table values. The size of the vector determines the valid index range for the input index. The data will be explicitly converted into the type specified in the **Output data type** parameter. If the input index exceeds table size and **Saturate at table ends** is specified for the **When input is out of range** parameter, then index value is saturated to either top or bottom of table size range. If **Wrap around** is specified for the **When input is out of range** parameter, the index is wrapped into the valid table size range.

***Note*:** Large tables should be defined via a Simulink workspace variable due to space limitations in the block dialog box.

Input bias

This parameter is an offset into the table data that will be added to the index input. This makes it possible to use negative indices and perform look up operation.

When input is out of range

This parameter will guard the index value if it exceeds the valid table size range.

Following are the settings for the **When input is out of range** parameter.

Table 55: **When Input Is Out of Range Parameter**

Setting	Description
Saturate at table ends	If index value exceeds the valid table size range, then index value is saturated to either top or bottom of table size range, depending on the overflow direction.
Wrap around	If index value exceeds the valid table size range, then index value is wrapped into the valid table size range.

Output data type

Specifies the output data type.

Following are settings for the **Output data type** parameter.

Table 56: **Output Data Type Parameter**

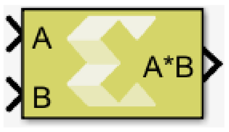
Setting	Description
double	double precision floating-point
single	single precision floating-point
int8	8-bit signed integer
uint8	8-bit unsigned integer
int16	16-bit signed integer
uint16	16-bit unsigned integer
int32	32-bit signed integer
uint32	32-bit unsigned integer
boolean	boolean
fixed	fixed-point
half	half precision floating-point
data type expression	

Matrix Multiply

Compute matrix product of two input signals. The first operand is the top input on the block.

Library

Math Functions / Matrices and Linear Algebra



Description

The Matrix Multiply block has two input ports and one output port. The output signal is the matrix product of the input signals where the first operand corresponds to the top input.

Data Type Support

The data type of the input signals can be any floating-point, fixed-point, integer, or Boolean. The input signals can be real or complex. The input signals can be scalar, vector, or matrix, but they do need to be such that mathematically, the matrix product is defined. The table below shows valid combinations. Combinations that do not match any row in the table result in an error.

Table 57: **Data Type Combinations**

Dimensions of First Operand	Dimensions of Second Operand	Dimensions of Matrix Product	Conditions
K x L	L x M	K x M	K >= 1, L >= 1, M >= 1
K x L	L	K	K >= 1, L > 1
K x 1	1	K x 1	K >= 1
K	1	K	K >= 1
K	1 x M	K x M	K >= 1, M >= 1

The output data type is determined according to the following rules, in the order listed. T1 is a variable representing the type of the first operand; T2 is a variable representing the type of the second operand. These rules were chosen for maximum alignment with Vivado® HLS, which may not correspond to the output data type computed via the internal rule of the Simulink® Matrix Product block.

Table 58: **Output Data Type**

Data Type of First Operand	Data Type of Second Operand	Data Type of Matrix Product
T1: floating-point	T2	The widest floating-point type between T1 and T2 if T2 is a floating-point type; otherwise T1
T1	T2: floating-point	The widest floating-point type between T1 and T2 if T1 is a floating-point type; otherwise T2
fixed-point	fixed-point	The smallest fixed-point type capable of representing the product without loss of precision
fixed-point	integer	The smallest fixed-point type capable of representing the product without loss of precision
integer	fixed-point	The smallest fixed-point type capable of representing the product without loss of precision
T1: integer	T2: integer	Let W1 be the bit width of T1 and W2 be the bit width of T2. The product is the integer type with bit width max(W1,W2) and it is signed if either T1 or T2 are signed.
boolean	T2	T2
T1	boolean	T1

Parameters

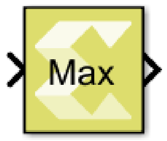
The Matrix Multiply block has no parameters to set.

Max

Outputs the maximum value of an input or element-wise maximum value of multiple inputs.

Library

Math Functions/Math Operations



Description

The Max block with a single scalar or vector input, outputs the maximum value of the input.

If the block has more than one input, the non-scalar inputs must have the same dimensions. Any scalar input is expanded to the dimensions of the non-scalar inputs, and the block outputs element-wise maximum value of the inputs.

Data Type Support

- The block supports floating-point, integer, and fixed-point data types.
- The block supports real valued inputs.
- For Number of Inputs = 1.
  - The block supports a scalar or vector (1-D or 2-D) input when it has only one input port. The output is a scalar.
- For Number of Inputs > 1.
  - The block supports scalar, vector, or matrix inputs when it has more than one input port.
  - The output has the same dimensions as those of the inputs when all the inputs have the same dimensions.
  - The block supports mixed dimensions for inputs when it has more than one input port, provided all the non-scalar inputs have the same dimensions. Any scalar input is expanded to the dimensions of non-scalar inputs, and the block outputs element-wise maximum value of the inputs. The output has the same dimensions as those of the non-scalar inputs.
- The block supports inputs having different data types. The output data type in this case is defined by the following set of rules.
  - If the data type of one of the inputs is a floating-point type, the data type of the output is the floating-point type among the data types of the inputs with the most precision.
  - If the data type of one of the inputs is a fixed-point type, the data type of the output is the smallest fixed-point type capable of representing the result without any loss of precision.
  - If the inputs are integral, the output is integral. If any input is signed, the output is signed. The bit width of the output is the largest among the bit widths of the inputs.

Parameters

- **Number of inputs:**

This parameter determines the number of inputs.

Table 59: **Number of Inputs**

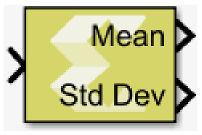
Choices	Description
1	Initially, the block icon has a single input.
N	A positive integer value. The block icon is redrawn with the specified number of input ports.

Mean & Std Deviation

Compute the mean and standard deviation of the input data

Library

Computer Vision



Description

The Mean & Std Deviation block computes the mean and standard deviation of the input data. Mean and standard deviation are calculated in this way:

$$\mu = \frac{\sum_{y=0}^{height} \sum_{x=0}^{width} src(x,y)}{(width * height)}$$

$$\sigma = \sqrt{\frac{\sum_{y=0}^{height} \sum_{x=0}^{width} (\mu - src(x,y))^2}{(width * height)}}$$

Data Type Support

The Mean and standard deviation block accepts an input image (represented by an M-by-N matrix) of type uint8.

The output data type is scalar of x_ufix16_En8 for mean and scalar of x_ufix16_En8 for standard deviation.

Data type support is:

- Input Dimension: 2-D array
- Input Data Type: uint8
- Output Dimension: Scalar
- Output Data type: x_ufix16_En8

Parameters

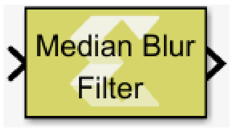
The Mean & Std Deviation block does not have any parameters to set.

Median Blur Filter

Apply a 3x3 median filter to the input

Library

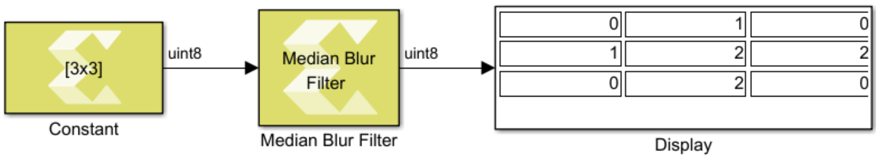
Computer Vision



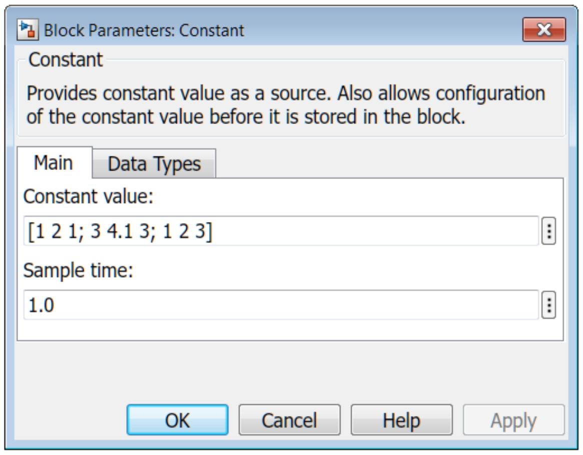
Description

The median blur filter operates on grayscale images of type uint8, and acts as a non-linear digital filter which improves noise reduction. It uses a filter size of 3 and the output is computed as the median value of the 3x3 neighborhood of pixel values. The output type is the same as the input type, uint8.

Figure 61: **Median Blur Filter Block and Parameters**



0	1	0
1	2	2
0	2	0



Block Parameters: Constant

Constant

Provides constant value as a source. Also allows configuration of the constant value before it is stored in the block.

Constant value:

[1 2 1; 3 4.1 3; 1 2 3]

Sample time:

1.0

Data Type Support

The Median Filter block accepts as input real-valued 2-D matrices of type uint8. The generated output type is also of type uint8.

Parameters

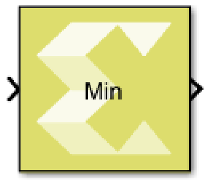
The Median Blur Filter block has no parameters to set.

Min

Outputs the minimum value of an input or element-wise minimum value of multiple inputs.

Library

Math Functions/Math Operation



Description

The Min block with a single scalar or vector input outputs the minimum value of the input.

If the block has more than one input, the non-scalar inputs must have the same dimensions. Any scalar input is expanded to the dimensions of non-scalar inputs and the block outputs element-wise minimum value of the inputs.

Data Type Support

- The block supports floating-point, integer, and fixed-point data types.
- The block supports real valued inputs.
- Number of inputs = 1.
  - The block supports a scalar or vector (1-D or 2-D) input when it has only one input port. The output is a scalar.
- Number of inputs > 1.
  - The block supports scalar, vector, or matrix inputs when it has more than one input port.
  - The output has the same dimensions as those of the inputs when all the inputs have the same dimensions.
  - The block supports mixed dimensions for inputs when it has more than one input port, provided all the non-scalar inputs have the same dimensions. Any scalar input is expanded to the dimensions of non-scalar inputs and the block outputs element-wise minimum value of the inputs. The output has the same dimensions as those of the non-scalar inputs.
- The block supports inputs that have different data types. The output data type in this case is defined by the following set of rules.
  - If the data type of one of the inputs is a floating point type, the data type of the output is the floating point type among the data types of the inputs with the most precision.
  - Otherwise, if the data type of one of the inputs is a fixed-point type, the data type of the output is the smallest fixed-point type capable of representing the result without any loss of precision.
  - Otherwise, if the inputs are integral, the output is integral. If any input is signed, the output is signed. The bit width of the output is the largest among the bit widths of the inputs.

Parameters

- **Number of inputs:**

  This parameter determines the number of inputs.

Table 60: **Number of Inputs**

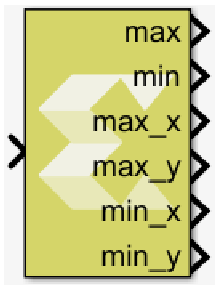
Choices	Description
1	Initially, the block icon is created with a single input.
N	A positive integer value. The block icon is redrawn with the specified number of input ports.

MinMax Location

Determines the value and the location of the minimum and maximum of the input image

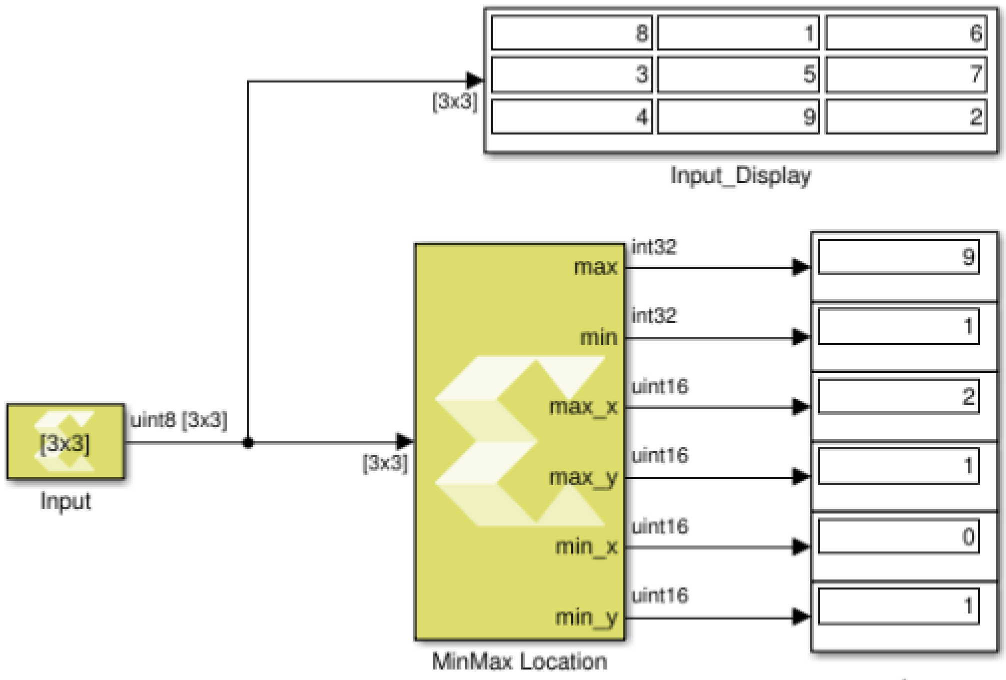
Library

Computer Vision



Description

The MinMax Location block accepts a real-valued 2D matrix image as its input and returns the location and value of the minimum and maximum. The block looks up all the pixels of an image row-wise and returns the location of first occurrence if min/max value is present more than once in an image.



In the above design, the MinMax Location block returns a maximum value of 9, a minimum value of 1, and the locations of these values: (2,1) and (0,1) respectively.

Data Type Support

Data types accepted at the inputs of the block:

- Unsigned 8-bit integer
- Signed 16-bit integer
- Unsigned 16-bit integer
- Signed 32-bit integer

The min/max value outputs are of type int32, and the coordinates for locating min and max are returned as uint16.

Parameters

The MinMax Location block has no parameters to set.

Model Composer Hub

Control implementation of the model

Description

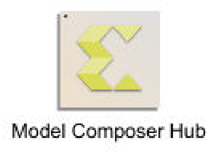
The Model Composer Hub block controls the behavior of the Xilinx® Model Composer tool.

You can specify the targeted design flow for the generated output, the directory path for the output, and the desired device and design clock frequency using following tabs.

- The **Compilation** tab gives options to select the output flow through Export type, the code generation directory, and test bench generation.
- The **Hardware** tab helps with device or board selection. You can specify clock frequency and throughput factor for the model.
- The **User Connect** tab requests feedback for the tool.

Library

Tools

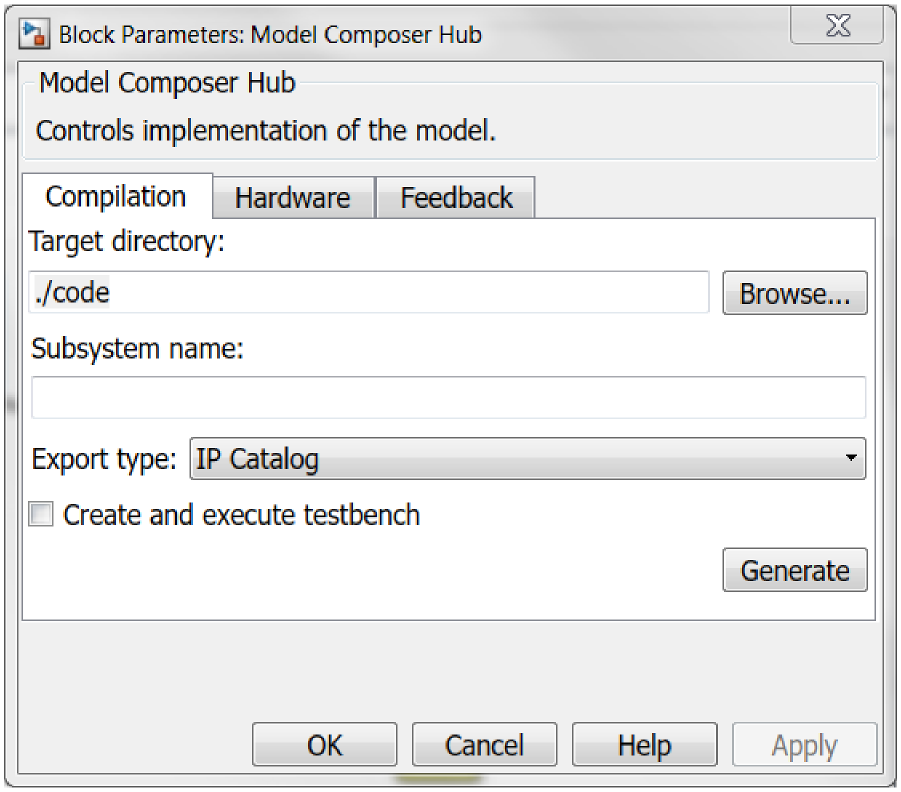


Data Type Support

Data type support is not applicable to the Model Composer Hub block.

Parameters

Figure 62: **Model Composer Block Parameters**


Block Parameters: Model Composer Hub

Model Composer Hub

Controls implementation of the model.

Target directory: ./code

Subsystem name:

Export type: IP Catalog

☐ Create and execute testbench

- **Compilation:**
  - **Target directory:** Enter the output directory name with the complete path, or use the **Browse** button to provide a path.
  - **Subsystem name:** Enter a subsystem name that contains only Model Composer blocks.
  - **Export type:**

    Settings for the **Export type** are:

    Table 61: **Export Type Settings**

Setting	Description
IP Catalog	Select **IP Catalog** to export the design to the Vivado IP Catalog. After C/C++ code generation,Vivado High-Level Synthesis (HLS) is invoked to synthesize the C code and create a project that can be exported as an IP to the Vivado IP Catalog.
System Generator	Select **System Generator** to export the design to System Generator. After C/C++ code generation, Vivado High-Level Synthesis (HLS) is invoked to synthesize the C code and create an RTL solution that can be used as a Vivado HLS block in a System Generator model.
C++ code	Select **C++ code** to compile the design model into C++ code.

- **Create and execute testbench:**

  If selected, Model Composer runs simulation and generates test vectors while generating code.

  - **Testbench stack size:**

    This parameter prompts you to enter a larger stack size.

    When **Create and execute testbench** is enabled, the **Testbench stack size** option specifies the size of the testbench stack frame during C simulation (CSIM). Occasionally, the default stack frame size of 10 MB allocated for execution of the testbench may be insufficient to run the test, due to large arrays allocated on the stack and/or deep nesting of sub-systems. Typically when this happens, the test would fail with a segmentation fault and an associated error message. In such a case you may opt to increase the size of the stack frame and rerun the test.

- **Hardware:**
  - **Project device:**

    Specifies the current target part or board platform for the Model Composer model.

    Clicking the browse button (...) next to **Project device** displays the Device Chooser dialog box, which allows you to select the board or part to which your design is targeted. Xilinx Model Composer obtains board and device data from the Vivado Design Suite database.

  - **FPGA clock frequency:**

    Specifies the clock frequency in MHz for the Xilinx device. This frequency is passed to the downstream tool flow.

  - **Throughput factor:**

    Specifies the number of samples processed per clock to increase the throughput. A larger factor increases hardware resource usage. The throughput factor must be between 1 and 16.

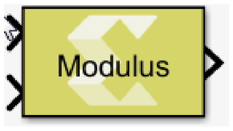
- **Feedback:** This tab requests feedback to the tool developers, and suggestions to improve the tool. It points to a weblink that opens a survey that can be completed in less than 3 minutes.

Modulus

Performs element-wise modulus operation on the input signals

Library

Math Functions / Math Operations



Description

The Modulus block takes two inputs. The first input is taken as dividend and the second input is considered as divisor. The output is the remainder after division. For each element of the dividend A, compute the modulus operation (remainder after division) with regard to the corresponding element of the divisor B, as follows:

```
M = mod(A, B) = A - B .* floor(A ./ B)
```

The block can handle division by 0 by returning the NaN output for floating-point data types and 0 for the rest of the data types.

Note: For signed data types the remainder will have the same sign as the divisor B. If instead it should have the same sign as the dividend, use the Remainder block instead of the Modulus block.

Data Type Support

Data types accepted at the inputs of the block are:

- The block supports all native Simulink® data types, as well as half precision floating-point, and fixed-point data types.
- For inputs of bool data type, the output value is always false.
- The block supports scalars, vectors, and 2D matrices.
- The input dimensions must match unless one input is a scalar, in which case, it gets broadcast to be used with each of the other input's elements. The output dimension is the larger of the two input dimensions.
- The block supports mismatched input data types. The output data type is decided by Xilinx® Model Composer data type propagation rules.
- The block operates on inputs of real numeric type only. For input of complex numeric type it issues an error.

Parameters

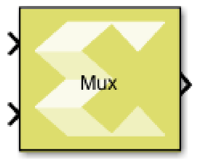
The Modulus block has no parameters to set.

Mux

Combines scalar and vector inputs into a larger vector output.

Library

Signal Routing

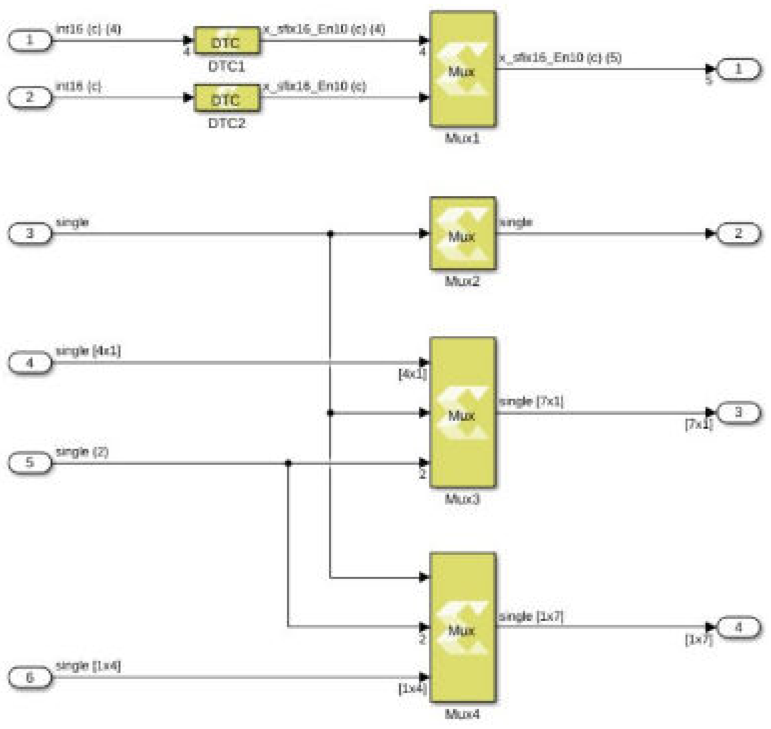


Description

The Mux block combines scalar and vector inputs into a larger vector. The elements of the inputs are concatenated starting from the first input at the top left. The block takes an input signal that is a scalar, a vector, a row matrix, or a column matrix with the limitation that it cannot support a row matrix signal, and a column matrix signal at the same time. If an input is a row vector or a column vector, then the output also takes that form. The output is a non-virtual vector meaning that its elements are stored in contiguous memory.

The number of inputs to the block is configurable using the Number of inputs block parameter. When the value of the block parameter is changed, the input ports are added or removed starting from the last port at the bottom left.

Figure 63: **Mux Diagram**



Note: This figure shows how the Mux block computes output port dimensions.

Data Type Support

- **Inputs:**
  - The number of inputs is decided by value of the Number of inputs parameter.
  - The input signal can be a scalar, a vector (M), a row matrix (1xM), or column matrix (Mx1).
  - The block cannot have a row matrix and a column matrix as inputs at the same time.
  - All inputs must have the same data type, and the same numeric type either real or complex.
  - The Mux block supports all native data types (double, single, uint8, int8, uint16, int16, uint32, int32, and boolean), and Model Composer supported half and fixed-point data types.
- **Outputs:**
  - The block has one output port.
  - The output data type and numeric type are the same as the inputs.
  - The output signal dimension depends upon the dimensions of the input signals.

Parameters

- **Number of inputs:**

  The value for the parameter must be a finite positive integer. When the value of the parameter changes, the input ports are either added or removed starting from the last port at the bottom left.

Table 62: **Number of Inputs**

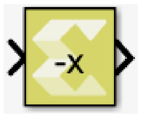
Choices	Description
2	The block icon is initially created with two input ports.
N	A finite positive integer value. The block icon is redrawn with the specified number of input ports.

Negate

Perform element-wise unary minus operation on the input data

Library

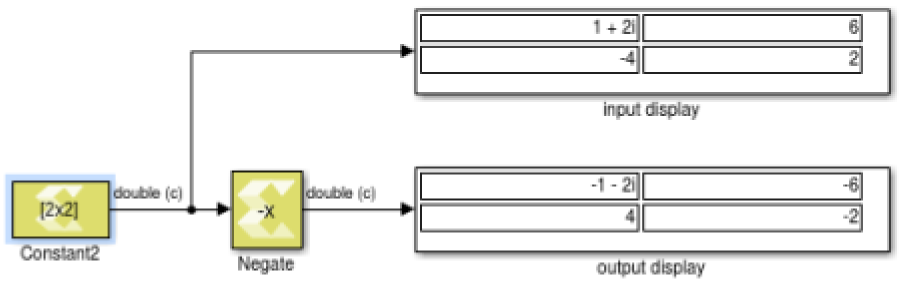
Math Functions / Math Operations



Description

The Negate block computes element-wise minus operation on the input data. The block handles signedness of real and imaginary parts separately in case of complex data input.

Figure 64: **Negate Block**



Data Type Support

Data type support is:

- Input Data Type: All data types are supported except unsigned integer and Boolean values.
- Output: The data type, dimension, and complexity of the output are the same as those of the input signal.

The block supports scalar, vector, and two-dimensional matrix data.

Parameters

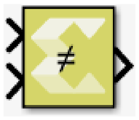
The Negate block has no parameters to set.

Not Equals

Perform element-wise not equal to relational operation on the inputs. The top input corresponds to the first operand.

Library

Relational Operations



Description

The Not Equals block has two input signals and one output signal. The block compares two inputs using element-wise not equals relational operation. The first input corresponds to the top input port and the second input to the bottom input port. The dimension of the output signal matches the dimensions of the input signals. An element of the output signal is true if the corresponding element of the first input signal is not equal to the corresponding element of the second signal; otherwise the element is false.

Data Type Support

Data type support for the block is:

- The data types of the input signals can be of any integer, fixed-point, boolean, or floating-point data type.
- The input signals can be a scalars, vectors, or matrices. If both inputs are not scalar, their dimension must match.
- The input signals can be complex.
- The output signal is boolean.
- The dimension of the output signal is scalar if both inputs are scalar. Otherwise, it matches the dimensions of the non-scalar input.

Parameters

The Not Equals block has no parameters to set.

Otsu Thresholding

Find threshold intensity value for the input greyscale image (uint8 data type) such that pixel intensity levels on each side of the threshold have the minimum spread.

Library

Computer Vision

Description

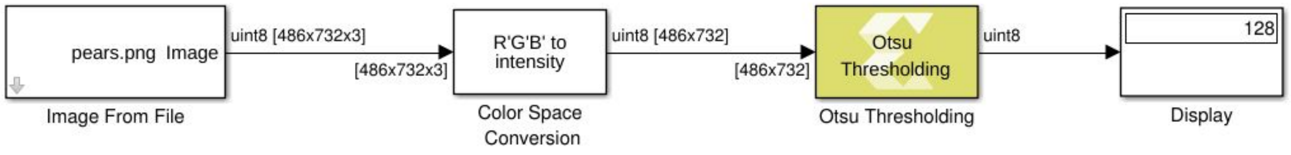
Thresholding techniques are used in image processing to convert a grayscale image into a binary image.

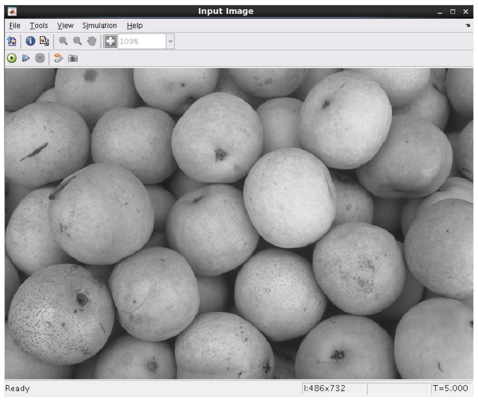
Otsu Thresholding is one such technique that computes an optimum intensity threshold to divide pixels into two groups such that sum of intensity spread within a group has minimum variance.

The optimum threshold value can be used to convert grayscale image into a monochrome image by replacing all lower intensity pixels with 0 intensity (black) and by replacing all higher intensity pixels with 1 (white).

The following design model uses the Otsu Thresholding block to compute an optimum threshold value for the input image.

Figure 65: **Otsu Thresholding Block**





Data Type Support

Data type support is:

- Input: A 2-D grayscale image in uint8 data type.
- Output: A uint8 scalar value.

Parameters

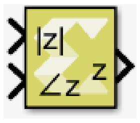
The Otsu Thresholding block has no parameters to set.

Polar to Complex

Element-wise conversion of real magnitude and angle representation signals into a complex signal

Library

Math Functions / Math Operations



Description

The Polar to Complex block accepts a real signal of floating point data type such as double, single, or half. The first and second inputs represent magnitude and angle respectively. The angle is in radians. The outputs are complex values of the same data type as the block input for a given real magnitude and angle. The input can be scalar, vector or matrix of real signals, in which case the output signals are also scalar, vector, or matrix. The elements of a certain magnitude input map to the magnitudes of the corresponding complex output elements. Similarly, the elements of a certain angle input map to the angles of the corresponding complex output elements.

Data Type Support

Data types accepted at the inputs of the block are:

- Dimension : The inputs can be scalar, array, or combination of scalar and an array. If both the inputs are arrays, they must have the same dimensions.
- Data Types: Supports signal of floating point data type such as double, single, and half. Both inputs must have the same data type.
- Complex Number Support: No

Outputs for the different input types are:

Table 63: **Input/Output**

Inputs	Output
Both are scalar	Scalar
Both are vector	Vector of same dimension
Both are matrix	Matrix of same dimension
One is scalar and the other is vector or matrix	Dimension is that of the vector or matrix

Parameters

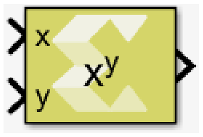
The Polar to Complex block has no parameters to set.

Pow

Compute the element-wise power function

Library

Math Functions / Math Operations



Description

The Pow block computes the value of a base raised to the power exponent. Z = X^y raises each element of X to the corresponding power in y. If one of inputs is a matrix and the other is a scalar, the scalar input is expanded to match the dimension of the non-scalar input to perform the operation. If both inputs are non-scalar, they must agree in dimension.

Data Type Support

Data type support for the block is:

- Dimension: Inputs can be scalar, vector, or matrix. If one of the inputs is scalar and the other is a vector or matrix then the scalar input is expanded to match the other input dimension and the operation will be performed element-wise. If both inputs are non-scalar, then they must match in dimension.
- Data Types: Input supports signals of integer type, floating point data type (double, single, and half), and signed and unsigned fixed-point type. Both inputs must be of the same data type.
- Complex Numbers: Complex numbers are not supported.

A negative base value raised to a fractional power will result in a Not A Number (NAN) value.

Parameters

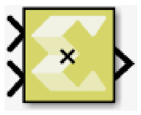
The Pow block has no parameters to set.

Product

Compute element-wise product of the input signals

Library

Math Functions / Math Operations



Description

The Product block computes the element-wise product of its input signals.

The block warns or errors out when an integer output overflows during simulation. To configure, select **Simulation → Model Configuration Parameters → Diagnostics → Data Validity** for your model in the Simulink® Editor, then set the **Wrap on overflow**, or **Saturate on overflow** parameter.

Data Type Support

The Product block outputs the result of multiplying two inputs. Inputs can be two scalars, a scalar and a non-scalar, and two non-scalars of the same dimension. If one of the inputs is scalar and other input is vector/matrix then the scalar input is expanded to match the dimension of the other input to perform the operation. If input signals are both integer data types, such as int16, the datatype of the output is also an integer datatype int16. Hence an overflow of wrap around (no saturation) is likely if the output exceeds the range that is represented with int16. If the operation is to be applied without loss of precision or range, use fixed-point data types. If you want to narrow the bit width of the output signal, you can run it through a Data Type Convert block and select a fixed-point data type that saturates.

The output dimension is the same as that of the inputs if both inputs are either scalar or non scalar. If one input is scalar and the other is a vector/matrix then the output dimension matches the dimension of the vector/matrix input. The output type is same as that of the inputs if both inputs are of the same type. Otherwise, the output type is defined as follows.

Table 64: **Input/Output**

Input Type	Output Type
(double, single)	double
(double, int)	double
(double, half)	double
(double, fixed)	double
(double, Boolean)	double
(single, int)	single
(single, half)	single
(single, fixed)	single
(single, Boolean)	single

Table 64: **Input/Output** *(cont'd)*

Input Type	Output Type
(half, int)	half
(half, fixed)	half
(half, Boolean)	half
(fixed, int)	fixed
(fixed, Boolean)	fixed
(int, Boolean)	int

Parameters

Saturate on integer overflow

This parameter specifies whether integer overflow is handled by wrapping (default) or by saturating. This parameter is relevant only if the output is integral (int8, int16, int32, uint8, uint16, uint32).

Settings for the **Saturate on integer overflow** parameter are:

Table 65: **Saturate on Integer Overflow Parameter**

Setting	Description
Not selected	Integer overflow is handled by wrapping.
Selected	Integer overflow is handled by saturation.

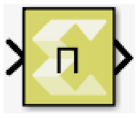
When overflow is detected, the Diagnostic Viewer displays messages that depend on the diagnostic action you specify in the Simulink Editor. To configure, select **Simulation → Model Configuration Parameters → Diagnostics → Data Validity** for your model in the Simulink Editor, then set the **Wrap on overflow** or **Saturate on overflow** parameter.

Product of Elements

Multiply the elements of the input signal

Library

Math Functions / Math Operations



Description

The Product of Elements block computes the product of the elements of the input signal. The block can be configured in the following ways.

- By default, the output is scalar and equal to the product of all (matrix) elements of the input signal.
- If the dimension to multiply over is specified to be **1**, the output is a row matrix (1xN), where N is the number of columns of the input, and element (1,k) is the product of the elements of column k of the input.
- If the dimension to multiply over is specified to be **2**, the output is a column matrix (Mx1), where M is the number of rows of the input, and element (k,1) is the product of the elements of row k of the input.

Data Type Support

The input signal can be real or complex. The input data type can be any Boolean, integer, floating-point, or fixed-point data type. The block can perform element-wise multiplication on real or complex number data.

Parameter

Multiply over

The **Multiply over** parameter value is used to decide whether elements will be multiplied in all dimensions or in one of the dimensions.

Following are the settings for the **Multiply over** parameter.

Table 66: **Multiply Over Parameter**

Setting	Description
All dimensions	Multiply all elements of the input signal (output is scalar)
Specified dimension	This option shows an edit box, **Dimension**, where the specific dimension value can be entered.

Dimension

The **Dimension** parameter is displayed only if the **Multiply over** parameter value is set to **Specified dimension**.

Following are the settings for the **Dimension** parameter.

Table 67: **Dimension Parameter**

Setting	Description
1	Multiply input over row dimension. Output is a row matrix.

Table 67: **Dimension Parameter** *(cont'd)*

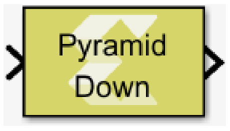
Setting	Description
2	Multiply input over column dimension. Output is a column matrix.

Pyramid Down

Downsample an unsigned 8-bit integer grayscale image to half its size

Library

Computer Vision



Description

The Pyramid Down block implements an image down-sampling algorithm to resize an unsigned 8-bit integer grayscale input image (2-D matrix) to half its height and half its width, i.e., to height = floor((rows+1)/2), and width = floor((cols+1)/2). It does so by first smoothing the image with a 5x5 Gaussian filter (filter kernel shown below), followed by dropping every other row and every other column.

The filter kernel used is given as follows:

$$\frac{1}{256}\begin{bmatrix} 1 & 4 & 6 & 4 & 1 \\ 4 & 16 & 24 & 16 & 4 \\ 6 & 24 & 36 & 24 & 6 \\ 4 & 16 & 24 & 16 & 4 \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix}$$

Data Type Support

The block supports only 2-D matrix inputs with an 8-bit unsigned integer pixel format.

Complex valued inputs are not supported.

Parameters

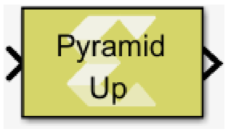
The Pyramid Down block does not have any parameters to set.

Pyramid Up

Upsamples an unsigned 8-bit integer grayscale image to twice its size

Library

Computer Vision



Description

The Pyramid Up block implements an image up-sampling algorithm to resize an unsigned 8-bit integer grayscale input image (2-D matrix) to twice its height and twice its width. The block inserts rows and columns of zeros between the original image rows and columns and applies a 5x5 Gaussian filter kernel to smoothen the resulting image. The filter kernel used is given as follows:

$$\frac{1}{256}\begin{bmatrix} 1 & 4 & 6 & 4 & 1 \\ 4 & 16 & 24 & 16 & 4 \\ 6 & 24 & 36 & 24 & 6 \\ 4 & 16 & 24 & 16 & 4 \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix}$$

Data Type Support

The block only supports 2-D matrix inputs with an 8-bit unsigned integer pixel format. Complex value inputs are not supported.

Parameters

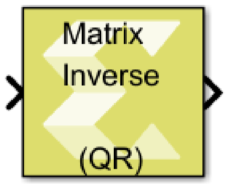
The Pyramid Up block has no parameters to set.

QR Inverse

Compute the inverse of a matrix using QR factorization

Library

Math Functions / Matrices and Linear Algebra



Description

The QR Inverse block provides the inverse of the input matrix A by performing QR factorization.

$$A^{-1} = Q^T R^{-1}$$

Q is an orthogonal matrix and R is an upper triangular square matrix. For singular matrix input the output would contain NaN or +inf/-inf.

Data Type Support

Data type support is:

- Dimension: Input has to be a square matrix. Scalar and vector inputs are not supported.
- Data Types: Input supports signals of floating point data types (double, single, and half). It does not support integer, boolean, and fixed-point data types.
- Complex Numbers: Complex numbers are not supported.

The output has the same dimension and data type as the input.

Parameters

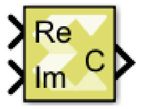
The QR Inverse block has no parameters to set.

Real-Imag to Complex

Computes the complex output from real and imaginary input.

Library

Math Functions / Math Operations



Description

The Real-Imag to Complex block converts the real and imaginary inputs to a complex-valued output signal. The input signal can be of any data type except boolean.. The complex output has the same data type as that of the block input. The input can be a scalar, 1-D vector, or matrix of real signals. It is possible to specify the constant real or imaginary part from the block dialog.

Data Type Support

Data type support for the input port is as follows.

- The block supports all data types except boolean. If the 'Input' is 'Real and Imag', both the inputs must have the same data type. Otherwise, 'Real part' or 'Imag part' parameter is converted to the same data type as that of the block input.
- Real and imaginary parts specified using inputs or block dialog must be real.
- The input can be a scalar, 1-D vector, or matrix.
- The output is always complex, and has the same data type as that of the input.
- Dimensions:
    - The output has the same dimensions as that of the input when both real and imaginary parts have the same dimensions.
    - The block supports mixed dimensions for real and imaginary inputs (specified as inputs or using block dialog). Any scalar input is expanded to the dimensions of non-scalar input, and the block has the same dimensions as those of the non-scalar input. If both the inputs are non-scalar, they must agree on dimensions.

Parameters

- **Input:**

  Input is a drop down menu parameter which specifies whether real, imaginary, or both of the parts of the output signal are specified as inputs.

  - **Settings:**

    Following are settings for the **Input** parameter.

    Table 68: **Input Parameter**

Setting	Description
Real and imag	Real and imaginary parts of the output signal are specified using Re and Im inputs of the block, respectively.
Real	The block has only Re input in this case. Real part of the output signal is specified using the Re input of the block, while its imaginary part is specified using the Imag part parameter.

Table 68: **Input Parameter** *(cont'd)*

Setting	Description
Imag	The block has only Im input in this case. Imaginary part of the output signal is specified using the Im input of the block, while its real part is specified using the Real part parameter.

- **Real part:** Specify the constant real part of the output signal when Input is set to Imag. This parameter is visible only when you set Input to Imag.

Table 69: **Real Part Parameter**

Choices	Description
0	The value of the Real part parameter must be a numeric, real-valued scalar, vector, or matrix.

- **Imag part:** Specify the constant imaginary part of the output signal when Input is set to Real. This parameter is visible only when you set Input to Real.

Table 70: **Imag Part Parameter**

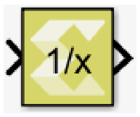
Choices	Description
0	The value of the Imag part parameter must be a numeric, real-valued scalar, vector, or matrix.

Reciprocal

Element-wise computation of the reciprocal for a given argument

Library

Math Functions / Math Operations



Description

The Reciprocal block returns the output of the function inv(`x`) for each element in array `x`. The block supports input of all data types except Boolean. The input can be scalar, vector or matrix.

Data Type Support

Data types accepted at the inputs of the block:

- Dimension: Input can be scalar, vector, or matrix.

- Data Types: Input supports signals of integer, fixed-point, and floating-point data type. The block does not support Boolean inputs.
- Complex Number Support: No

  The output has the same dimension and type as the input.

Parameters

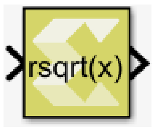
The Reciprocal block has no parameters to set.

Reciprocal Sqrt

Element-wise computation of the reciprocal square root for a given argument

Library

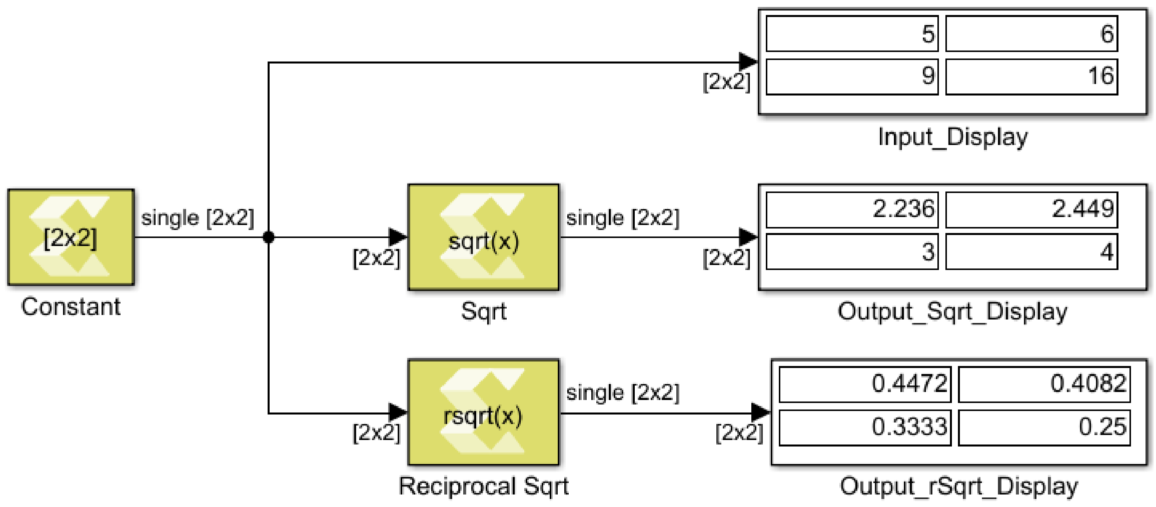
Math Functions / Math Operations



Description

The Reciprocal Sqrt block returns the reciprocal square root for each element in an array. The block supports input of all data types except Boolean. The input can be a scalar, vector or a matrix.

Figure 66: **Reciprocal Sqrt Block**



Data Type Support

Data type support is:

- Dimension: Input can be scalar, vector or matrix.
- Data Types: Input supports signals of integer, fixed-point, and floating point data type. It does not support Boolean inputs.
- Complex Numbers: Complex numbers are not supported.

The output has the same dimension and data type as the input.

Parameters

The Reciprocal Square root block has no parameters to set.

Reduction AND

Compute bitwise AND of the elements of the input over all dimensions or over a specified dimension

Library

Logic and Bit Operations

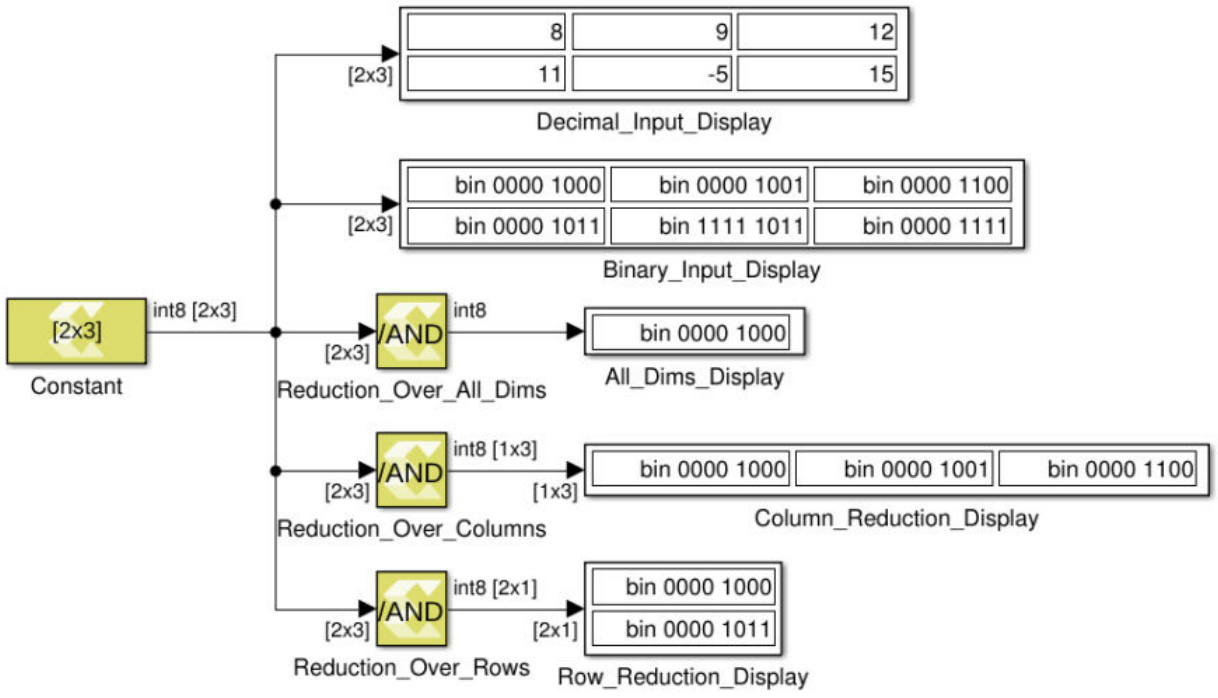
Description

The Reduction AND block has one input signal and one output signal. It computes the bitwise AND of the elements of the input signal over all of the dimensions or over a specified dimension. The data type of the output signal is the same as that of the input signal. The dimension of the output signals depends on whether the reduction takes place over all dimensions or over a specified dimension.

- **Reduce over all dimensions**: The output is a scalar and it is the bitwise AND of the elements of the input signal.
- **Reduce over dimension 1**: The output is a row vector (2-D) with as many elements as the number of columns of the input. Each element in the output is the bitwise AND reduction of the elements of the corresponding column of the input.
- **Reduce over dimension 2**: The output is a column vector (2-D) containing as many elements as the number of rows of the input. Each element in the output is the bitwise AND reduction of the elements of the corresponding row of the input.

In the example below a 2x3 input signal of type int8 feeds into three different configurations of the Reduction AND block.

Figure 67: **Reduction AND Block**



Data Type Support

- The input signal can be of any data type except for floating point types.
- The input signal must be real.
- The input signal must be a matrix if reduction is along dimension 2.
- The input signal can be a matrix, vector or scalar if reduction is along all dimensions or along dimension 1.

Parameters

Reduce over

This parameter specifies whether reduction takes place over all dimensions or over a specified dimension. If reduction is specified over all dimensions, the output signal is a scalar.

Following are the settings for the **Reduce over** parameter.

Setting	Description
All dimensions	The Reduce AND operator will be applied to all elements, producing a scalar output.
Specified dimension	The Reduce AND operator will be applied along the specified dimension, producing a vector output along the opposite dimension. When **Dimension** 1 is specified, the Reduction AND is applied along columns, producing a row vector as output. When **Dimension** 2 is specified, the Reduction AND is applied along rows, producing a column vector as output.

***Note*:** The dimension specified will be the one that gets reduced to size 1. For example, a 2-D M x N input matrix specifying **Dimension** 1 (number of rows M) will result in a 1 x N row vector.

Dimension

If the **Reduce over** parameter is set to **Specified dimension**, the **Dimension** parameter specifies over which dimension reduction takes place.

- If the input signal has dimensions M x N and the reduction **Dimension** is 1, the output has dimensions 1 x N.
- If the input signal has dimensions M x N and the reduction **Dimension** is 2, the output has dimensions M x 1.
- If the input signal is scalar or 1 x 1, the output dimension is 1 x 1.

Following are the settings for the **Dimension** parameter.

Table 71: **Dimension Parameter**

Setting	Description
1	Reduce over row dimension.
2	Reduce over column dimension.

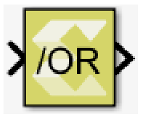
***Note*:** If the reduce **Dimension** is specified to be **2** the input signal must be two-dimensional.

Reduction OR

Compute bitwise OR of the elements of the input over all dimensions or over a specified dimension

Library

Logic and Bit Operations



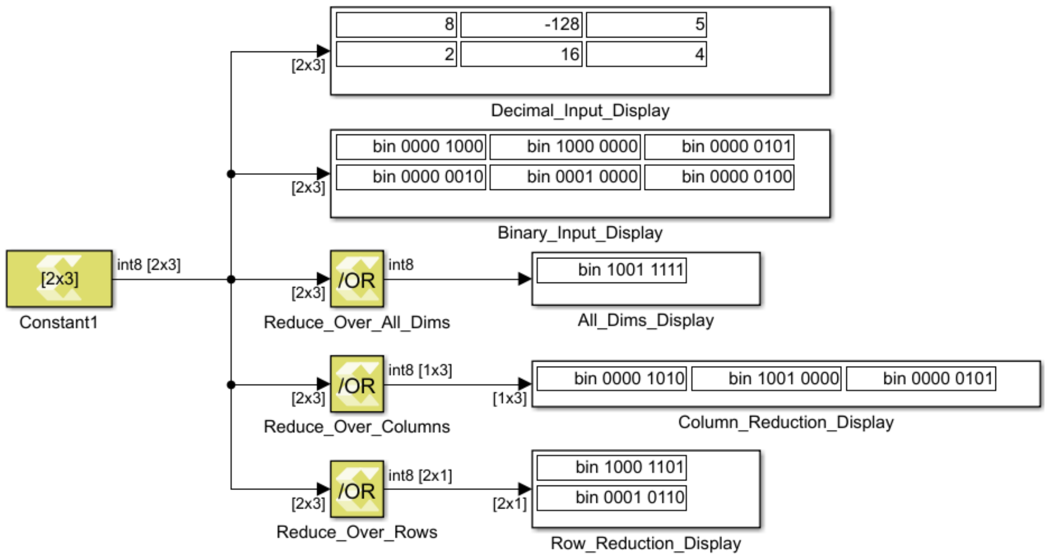
Description

The Reduction OR block has one input signal and one output signal. It computes the bitwise OR of the elements of the input signal over all of the dimensions or over a specified dimension. The data type of the output signal is the same as that of the input signal. The dimension of the output signals depends on whether the reduction takes place over all dimensions or over a specified dimension.

- **Reduce over all dimensions**: The output is a scalar and it is the bitwise OR of all of the elements of the input signal.
- **Reduce over dimension 1**: The output is a row vector (2-D) with as many elements as the number of columns of the input. Each element in the output is the bitwise OR reduction of the elements of the corresponding column of the input.
- **Reduce over dimension 2**: The output is a column vector (2-D) containing as many elements as the number of rows of the input. Each element in the output is the bitwise OR reduction of the elements of the corresponding row of the input.

In the example below a 2x3 input signal of type int8 feeds into three different configurations of the Reduction OR block.

Figure 68: **Reduction OR Block**



Data Type Support

- The input signal can be of any data type except for floating point types.
- The input signal must be real.
- The input signal must be a matrix if reduction is along dimension 2.
- The input signal can be a matrix, vector, or scalar if reduction is along all dimensions or along dimension 1.

Parameters

Reduce over

This parameter specifies whether reduction takes place over all dimensions or over a specified dimension. If reduction is specified over all dimensions, the output signal is a scalar.

Following are settings for the **Reduce over** parameter.

Table 72: **Reduce Over Parameter**

Setting	Description
All dimensions	Reduction takes place over all dimensions.
Specified dimension	Reduction takes place over the dimension specified by the **Dimension** parameter.

Dimension

If the **Reduce over** parameter is set to **Specified dimension**, the **Dimension** parameter specifies over which dimension reduction takes place.

- If the input signal has dimensions M x N and the reduction **Dimension** is 1, the output has dimensions 1 x N.
- If the input signal has dimensions M x N and the reduction **Dimension** is 2, the output has dimensions M x 1.
- If the input signal is scalar or 1 x 1, the output dimension is 1 x 1.

Following are settings for the **Dimension** parameter.

Table 73: **Dimension Parameter**

Setting	Description
1	Reduce over row dimension.
2	Reduce over column dimension.

***Note*:** If the reduce **Dimension** is specified to be **2** the input signal must be two-dimensional.

Reduction XOR

Compute bitwise XOR of the elements of the input over all dimensions or over a specified dimension

Library

Logic and Bit Operations

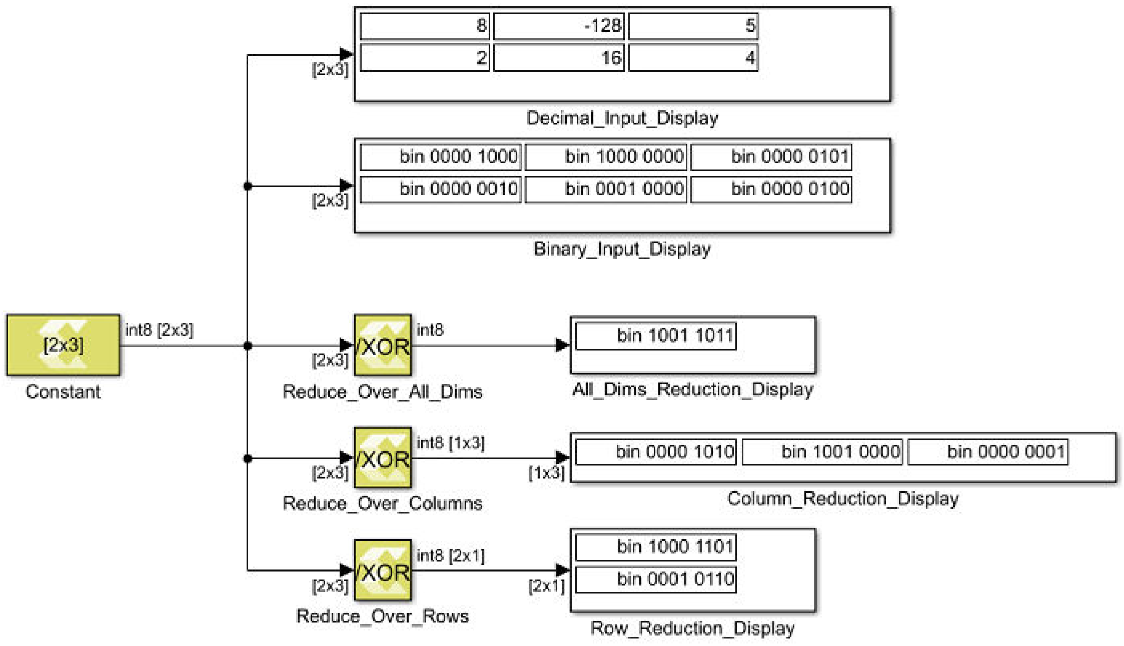
Description

The Reduction XOR block has one input signal and one output signal. It computes the bitwise exclusive OR (XOR) of the elements of the input signal over all of the dimensions or over a specified dimension. The data type of the output signal is the same as that of the input signal. The dimension of the output signals depends on whether the reduction takes place over all dimensions or over a specified dimension.

- **Reduce over all dimensions**: The output is a scalar and it is the bitwise XOR of all of the elements of the input signal.
- **Reduce over dimension 1**: The output is a row vector (2-D) with as many elements as the number of columns of the input. Each element in the output is the bitwise XOR reduction of the elements of the corresponding column of the input.
- **Reduce over dimension 2**: The output is a column vector (2-D) containing as many elements as the number of rows of the input. Each element in the output is the bitwise XOR reduction of the elements of the corresponding row of the input.

In the example below a 2x3 input signal of type int8 feeds into three different configurations of the Reduction XOR block.

Figure 69: **Reduction XOR Block**



Data Type Support

- The input signal can be of any data type except for floating point types.
- The input signal must be real.
- The input signal must be a matrix if reduction is along dimension 2.
- The input signal can be a matrix, vector or scalar if reduction is along all dimensions or along dimension 1.

Parameters

Reduce over

This parameter specifies whether reduction takes place over all dimensions or over a specified dimension. If reduction is specified over all dimensions, the output signal is a scalar.

Settings for the **Reduce over** parameter are:

Table 74: **Reduce Over Parameter**

Setting	Description
All dimensions	Reduction takes place over all dimensions.
Specified dimension	Reduction takes place over the dimension specified by the **Dimension** parameter.

Dimension

If the **Reduce over** parameter is set to **Specified dimension**, the **Dimension** parameter specifies over which dimension reduction takes place.

- If the input signal has dimensions M x N and the reduction **Dimension** is 1, the output has dimensions 1 x N.
- If the input signal has dimensions M x N and the reduction **Dimension** is 2, the output has dimensions M x 1.
- If the input signal is scalar or 1 x 1, the output dimension is 1 x 1.

Settings for the **Dimension** parameter are:

Table 75: **Dimension Parameter**

Setting	Description
1	Reduce over row dimension.
2	Reduce over column dimension.

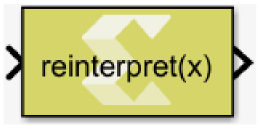
***Note*:** If the reduce **Dimension** is specified to be **2** the input signal must be two-dimensional.

Reinterpret

Element-wise reinterpretation of the input type into a compatible output type with the same bit width

Library

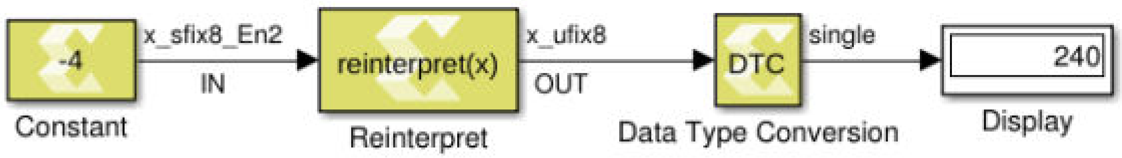
Signal Attributes



Description

The Reinterpret block provides a mechanism for interpreting a value from a different data type. You can specify the output data type with the restriction that the bit widths of input and output data types must match.

Figure 70: **Reinterpret Block**



In the above example, the input is of fixed-point signed number (x_sfix8_En2) represented with 8 bits, in which 2 bits are used for the fractional part (i.e., -4 = 1111 00.00). The Reinterpret block interprets the input type to unsigned fixed-point number (x_ufix8), represented with 8 bits as that of input, and no bits are used for the fractional part. The output becomes 240 (1111 0000).

Data Type Support

The data types of the input can be any integer, Boolean, fixed-point, or floating-point data type. The input can be any real or complex valued signal. If the input is real, the output is real. If the input is complex, the output is complex. The block supports scalar, vector, or matrix data.

Following are the supported output data types.

Table 76: **Input/Output**

Input Data Type	Supported Output Data Type
double	double, 64 bit fixed-point data
single	single, int32, uint32, 32 bit fixed-point data
int8	int8, uint8, 8 bit fixed-point data

Table 76: **Input/Output** *(cont'd)*

Input Data Type	Supported Output Data Type
uint8	uint8, int8, 8 bit fixed-point data
int16	int16, uint16, half, 16 bit fixed-point data
int16	uint16, int16, half, 16 bit fixed-point data
uint16	int32, uint32, single, 32 bit fixed-point data
int32	int32, uint32, single, 32 bit fixed-point data
uint32	uint32, int32, single, 32 bit fixed-point data
bool	bool, 1 bit fixed-point data
fixed	Same bit width fixed-point data with different fractional widths, all native data types if the bit width matches
half	half, int16, uint16, 16 bit fixed-point data type

Parameters

Output data type

This parameter specifies the output data type for reinterpreting the input data. If **fixed** is specified more parameters are available.

Following are the settings for the **Output data type** parameter.

Table 77: **Output Data Type Parameter**

Setting	Description
double	Double precision floating point
single	Single precision floating point
int8	8-bit signed integer
uint8	8-bit unsigned integer
int16	16-bit signed integer
uint16	16-bit unsigned integer
int32	32-bit signed integer
uint32	32-bit unsigned integer
boolean	Boolean
fixed	Fixed-point
half	Half precision floating-point
data type expression	

Remainder

Perform element-wise division on the input signal. The output is the remainder after the division.

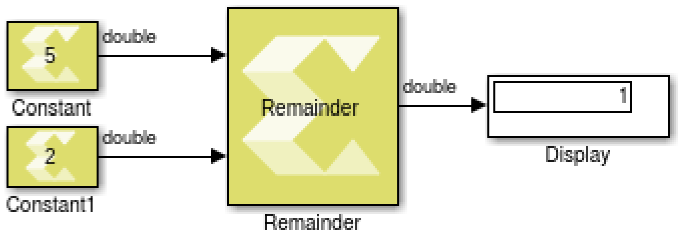
Library

Math Functions / Math Operations

Description

The remainder block takes two inputs. The dividend is the top input and the divisor is the bottom input. The block supports scalar, vector and matrix dimensions. The dimensions of the inputs must match unless one input is a scalar. The output has the larger dimension of the two inputs. The block can handle division by 0 and produces NaN as the output only for floating-point data types double and single.

Figure 71: **Remainder Block**



Data Type Support

The Remainder block supports all MATLAB native data types, half precision floating-point data type, and fixed-point. The block operates on real type inputs where it requires both inputs to have the same type of data.

Parameters

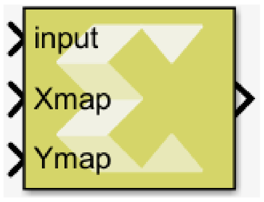
The Remainder block has no parameters to set.

Remap

Take pixels from one position in the input image and relocate them to another position in the output image.

Library

Computer Vision



Description

Renders an output image by means of looking up pixel data from a source image at locations given by the Xmap and Ymap arrays. For example:

$$\mathrm{dest}(x,y) = \mathrm{src}(\mathrm{Xmap}(x,y),\ \mathrm{Ymap}(x,y))$$

The Xmap and Ymap inputs must have data type single. For nearest-neighbor interpolation, the map values are converted to integer values via truncation. When bilinear interpolation is enabled, the fractional parts of map values control the ratios used for blending of neighboring source pixels.

The Remap block accepts an input intensity image (represented by an `M`-by-`N` matrix in uint8 grayscale), and two single precision float `M`-by-`N` matrices for mapping x- and y-coordinates to the output. The output is a remapped image of the same size as the input image. It uses either nearest neighbor or bilinear interpolation for mapping from source to destination.

Data Type Support

Data type support is as follows:

- **Input Port 1**: The **input** port data must be a 2-D matrix of real-valued unsigned 8-bit integer type.
- **Input Port 2**: **XMap** input port must be single, real valued, and of the same dimensions as the input.
- **Input Port 3**: **YMap** input port must be single, real valued, and of the same dimensions as the input.
- **Output port**: The output port is uint8, real, and of the same dimensions as the input.

Parameters

Prefetch buffer size (Rows)

This parameter is used to set the size of the line buffer in the synthesized implementation of this block, in terms of multiples of the width of the input image. For example, given an input image in Full HD resolution (width = 1920 pixels) a setting of 540 lines (rows) will create a line buffer of 1920 x 540 = 1,036,800 bytes.

***Note*:** this parameter does not affect simulation behavior. For simulation purposes the entire input image is buffered before the transformation takes place.

The **Prefetch buffer size (Rows)** edit box allows you to enter arbitrary integer values. Decimal values will be truncated to an integer value. You will receive an error if the value is zero or negative. If you select **Bilinear** for the **Interpolation method** parameter, the **Prefetch buffer size (Rows)** parameter requires even values, and odd values will be incremented by one automatically.

The amount of line-buffer needed is dependent upon the maximum vertical displacement within the y-coordinate mapping table for any given frame. For static transformations (i.e., if the **Ymap** input does not change) the required buffer size can be determined analytically and set appropriately. Moderate transforms such as correction of lens distortion, rotation by a small angle (leveling), translation by a small amount, etc. would require only a minimum of buffering, while flipping images upside down or free rotation require a full size line buffer.

The maximum relative y-displacement can be computed as:

$$\max_{x,y}(map_y(y,x) - (y-1))$$

over the entire y-displacement map table.

Interpolation method

This parameter specifies the interpolation method used to remap the image.

Following are the settings for the **Interpolation method** parameter.

Table 78: **Interpolation Method Parameter**

Setting	Description
Nearest neighbor	**Nearest neighbor** interpolation picks the value of the pixel that is closest to the computed coordinates.
Bilinear	**Bilinear** interpolation computes the linear interpolation between the four neighboring pixels closest to the computed coordinates.

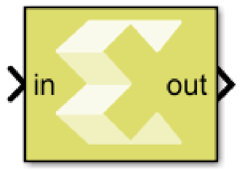
***Note*:** **Nearest neighbor** interpolation will be more resource and latency efficient than **Bilinear** interpolation, at the cost of reduced image quality.

Reshape Row-Major

Changes the input dimensions in row-major order.

Library

Math Functions/Math Operations



Description

The Reshape Row-Major block changes the input dimensions based on the specified "Output dimensionality" parameter. The output contains elements of the input in row-major order, that is the first row of the input matrix followed by the second row, and so on.

Data Type Support

- The block supports floating point, integer, boolean, and fixed-point data types.
- The block supports real and complex valued inputs.
- The input can be a scalar, 1-D vector, or matrix.
- The output has the same data type as the input.

Parameters

- **Output dimensionality:**

  This parameter specifies how the input should be reshaped. The settings for **Output dimensionality** are as follows:

Table 79: **Output Dimensionality**

Choices	Description
1-D array	Converts the input to a 1-D vector. For a matrix input, the output consists of input matrix elements in row-major order.
Column vector	Converts the input to a M x 1 matrix, where M is the number of elements in the input signal. For a matrix input, the output consists of input matrix elements in row-major order.
Row vector	Converts the input to a 1 x N matrix, where N is the number of elements in the input signal. For a matrix input, the output consists of input matrix elements in row-major order.
Custom	Converts the input to an output which has dimensions specified by the user using the **Output dimensions** parameter. The conversion is done in row-major order. The value of the **Output dimensions** parameter must be a two-element vector. For example, a value of `[M N]` outputs an M x N matrix. The number of elements of the input must match the number of elements specified by the **Output dimensions** parameter.

Table 79: **Output Dimensionality** *(cont'd)*

Choices	Description
Derive from reference input port	Creates a second input port on the block and derives the dimensions of the output from the dimensions of the second input port. Selecting this option disables the **Output dimensions** parameter. Both the inputs to the block must have the same number of elements.

- **Output dimensions:**

  Specify Output dimensions when **Output dimensionality** is set to **Custom**. The settings for **Output dimensions** are as follows:

Table 80: **Output Dimensions**

Choices	Description
[1, 1]	The value of the **Output dimensions** parameter must be a two-element vector.

Resize

Resize an image to specified dimensions

Library

Computer Vision

Description

The Resize block resizes grayscale images to a specified output dimension. The block supports up-scaling as well as down-scaling.

Data Type Support

The block only accepts real-valued 2D grayscale images of type uint8.

Parameters

Output dimensions

This parameter determines the size of the output image. The dimensions are given as a 2-element row-vector, e.g. [160 200], with the number of rows as the first element and the number of columns as the second element.

Interpolation method

This parameter specifies the interpolation method used to resize the image.

Following are settings for the **Interpolation method** parameter.

Table 81: **Interpolation Method**

Setting	Description
Area	The scaling algorithm will take the average of all the pixel values that will get mapped into the same output pixel location. This may be a preferred method for image decimation.
Bilinear	The algorithm will first interpolate linearly between the neighboring integer pixel locations along the x-axis, followed by another linear interpolation along the y-axis.
Nearest Neighbor	For non-integer pixel coordinates, the algorithm chooses the pixel closest to the target point.

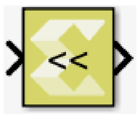
For up-scaling (scaling factors greater than 1) **Area** interpolation is equivalent to **Nearest Neighbor**. For down-scaling images, **Area** interpolation gives superior results that are free of moire effects. The fastest and most resource efficient interpolation method is **Nearest Neighbor**.

Shift Left

Perform logical shift left of input over a constant number of bit positions specified by a non-negative integer parameter

Library

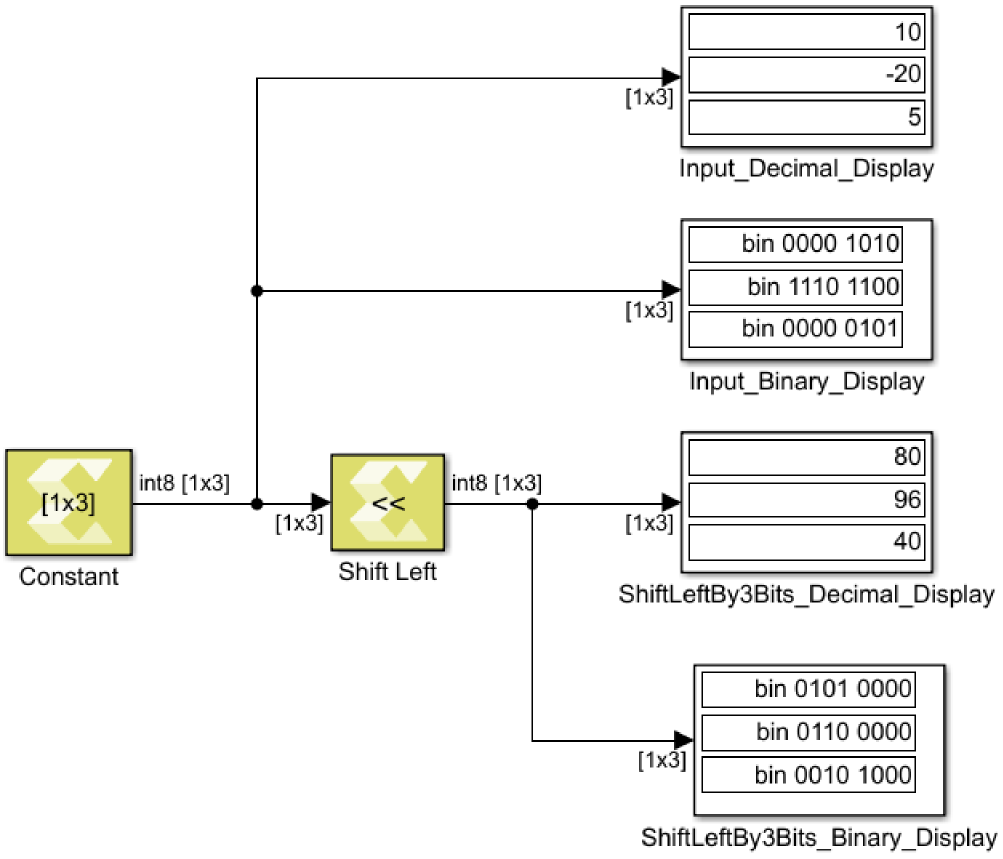
Logic and Bit Operations



Description

The shift left block computes element-wise left shift of the input by a constant amount specified via a parameter. This is also known as a logical shift left. The shift amount specifies over how many bit positions bits are shifted. This shift amount must be a non-negative integer. The default value is 0. The output is of the same type, dimension, and numeric type (real or complex) as the input. The input type must be integral or fixed-point.

Figure 72: **Shift Left Block**



Data Type Support

- Input: integer, fixed-point, but not logical or floating point.
- Output: output data type is the same as input data type.

Parameters

Shift by

This parameter specifies the number of bit positions over which the shift takes place.

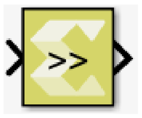
Enter a scalar real non-negative integer for the **Shift by** parameter.

Shift Right

Performs arithmetic shift right of input over a constant number of bit positions specified by a non-negative integer parameter

Library

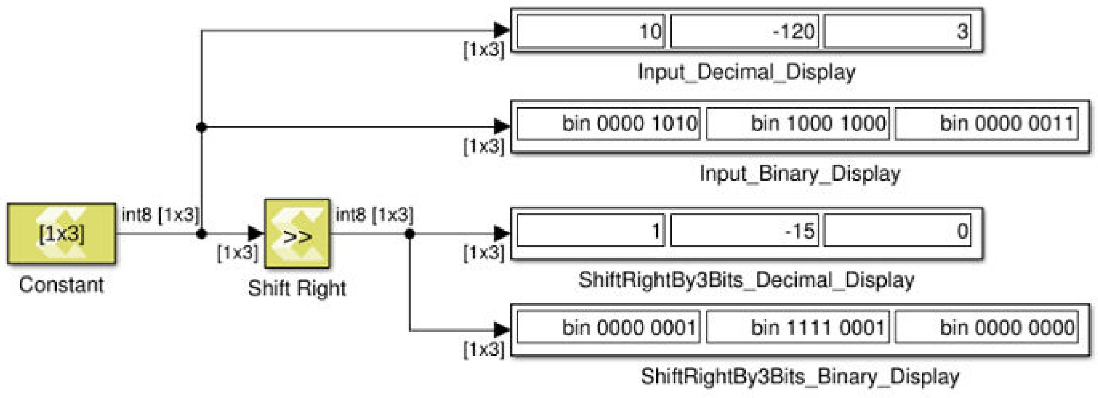
Logic and Bit Operations



Description

The shift right block computes an element-wise right shift of the input by a constant amount specified via a **Shift by** parameter. This is also known as arithmetic shift right. The shift amount specifies over how many bit positions bits are shifted. This shift amount must be a non-negative integer. The default value is 0. The output is of the same type, dimension, and numeric type (real or complex) as the input. The input data must be integer or fixed-point type.

Figure 73: **Shift Right Block**



Data Type Support

The Shift Right block supports integer and fixed-point input data, but not Boolean or floating type. Input data can be real or complex. Output data type and dimension are the same as that of input data.

Parameters

Shift by

This parameter specifies the number of bit positions over which the shift takes place.

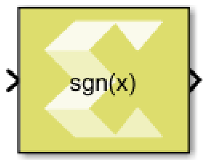
Enter a scalar real non-negative integer for the **Shift by** parameter.

Signum

Performs signum function (sign extraction) on the input.

Library

Math Functions/Math Operations



Description

The Signum block returns the sign for each element of the real input. The block returns 1, 0, or -1 if the number is positive, equal to 0, or negative, respectively.

When the input s is complex, the block output is calculated as:

```
sign(s) = s./ abs(s)
```

Where sign is the MATLAB® signum function, and `./` indicates element-wise division.

Data Type Support

The Signum block supports signals of integer type, floating-point type (double, single and half), and fixed-point type for real inputs. The complex inputs are supported only for floating point data types.

The output data type, and dimension are the same as those of the input.

Parameters

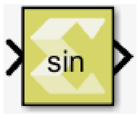
The Signum block has no parameters to set.

Sine

Element-wise computation of the sine function for the given input

Library

Math Functions / Math Operations



Description

The Sine block returns the output of the function sin(`x`) for each element in array `x`.

Data Type Support

Data type support is:

- Dimension: Input can be scalar, vector or matrix.
- Data Types: Input supports signals of floating point data types (double, single, and half) and signed fixed-point type. It does not support integer, Boolean, and unsigned fixed-point data types.
- Complex Numbers: Complex numbers are not supported.

The output has the same dimension and data type as the input. However, If the data type of the input is a fixed-point type, the data type of the output is fixed-point type with integer width fixed as 2.

Parameters

The Sine block has no parameters to set.

sinh

Element-wise computation of the hyperbolic sine for a given argument

Library

Math Functions / Math Operations

Description

The sinh block returns the output of the function sinh(x), which is the hyperbolic sine, for each element in array x.

The hyperbolic sine of `x` is:

$$\sinh(x) = \frac{e^{x} - e^{-x}}{2}$$

Data Type Support

Data type support is:

- Dimension: Input can be scalar, vector or matrix.
- Data Types: Input supports signals of integer type, floating-point data types (double, single, and half) and fixed-point type.
- Complex Numbers: Complex numbers are not supported.

The output has the same dimension and data type as the input.

Parameters

The sinh block has no parameters to set.

Sobel Filter

Compute the gradients of an input image in both x and y directions.

Library

Computer Vision

Sobel
Filter
x
y

Description

The Sobel Filter is a discrete differentiation operator. It computes an approximation of the gradient of the image intensity function by convolving the input image with the filter kernel in horizontal and vertical directions. The gradient approximation is used to find out the high-frequency variations in the image. Hence, the Sobel Filter is used mostly in edge detection algorithms.

Data Type Support

The Sobel Filter block accepts grayscale images (real-valued matrices) of type uint8 as input. The two outputs are real-valued grayscale images of type uint8 or int16, depending upon the user selection. The first output represents horizontal gradient, and the second output represents vertical gradient of the image. Both outputs have the same size as the input.

While detecting intensity gradients of an image, the Black-to-White transition is taken as a positive slope (represented with a positive value), whereas White-to-Black transition is taken as a negative slope (represented with a negative value).

- **Limitations:** Selection of uint8 as output data type results in the loss of information about the negative transition edge. Because the uint8 data type cannot support negative values, the values are saturated to zero. Use int16 as output data type to capture both positive and negative gradients of the image intensity.

Parameters

- **Filter Width:** This parameter selects filter kernel size of 3x3, 5x5, or 7x7. For any of the filter kernels, the output data type can be set to uint8 or int16.

  Settings for the **Filter Width** parameter are as follows.

Table 82: **Filter Width Parameter**

Setting	Description
3x3	Sets the filter kernel size to 3x3 coefficients. $Gx = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \; G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$
5x5	Sets the filter kernel size to 5x5 coefficients. $G_x = \begin{bmatrix} -1 & -2 & 0 & 2 & 1 \\ -4 & -8 & 0 & 8 & 4 \\ -6 & -12 & 0 & 12 & 6 \\ -4 & -8 & 0 & 8 & 4 \\ -1 & -2 & 0 & 2 & 1 \end{bmatrix} \; G_y = \begin{bmatrix} -1 & -4 & -6 & -4 & -1 \\ -2 & -8 & -12 & -8 & -2 \\ 0 & 0 & 0 & 0 & 0 \\ 2 & 8 & 12 & 8 & 2 \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix}$

Table 82: **Filter Width Parameter** *(cont'd)*

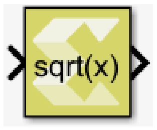
Setting	Description
7x7	Sets the filter kernel size to 7x7 coefficients. $$G_x = \begin{bmatrix} -1 & -4 & -5 & 0 & 5 & 4 & 1 \\ -6 & -24 & -30 & 0 & 30 & 24 & 6 \\ -15 & -60 & 75 & 0 & 75 & 60 & 15 \\ -20 & -80 & -100 & 0 & 75 & 60 & 15 \\ -15 & -60 & -75 & 0 & 75 & 60 & 15 \\ -6 & -24 & -30 & 0 & 30 & 24 & 6 \\ -1 & -4 & -5 & 0 & 5 & 4 & 1 \end{bmatrix}$$ $$G_y = \begin{bmatrix} -1 & -6 & -15 & -20 & -15 & -6 & -1 \\ -4 & -24 & -60 & -80 & -60 & -24 & -4 \\ -5 & -30 & -75 & -100 & -75 & -30 & -5 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 5 & 30 & 75 & 100 & 75 & 30 & 5 \\ 4 & 24 & 60 & 80 & 60 & 24 & 4 \\ 1 & 6 & 15 & 20 & 15 & 6 & 1 \end{bmatrix}$$

Sqrt

Element-wise computation of the square root for a given argument

Library

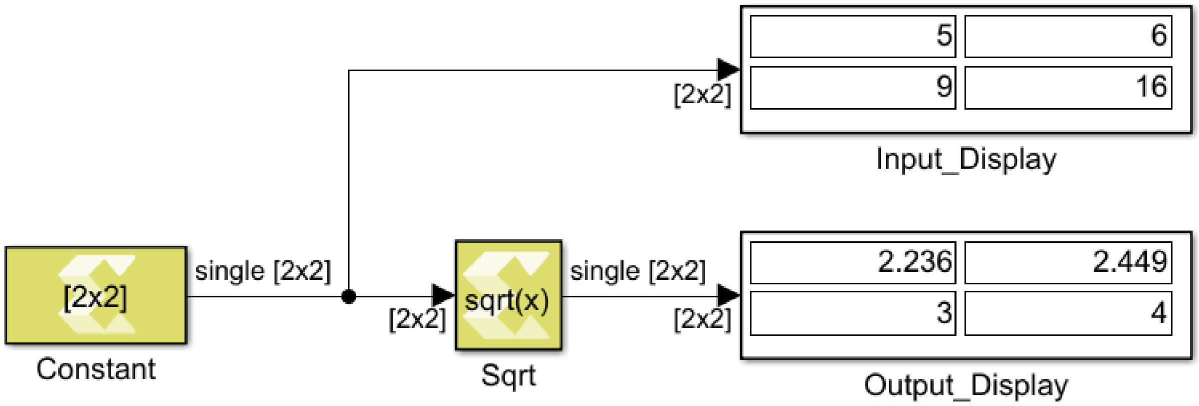
Math Functions / Math Operations



Description

The Sqrt block returns the square root for each element in array x. The block supports input of all data types except boolean. The input can be a scalar, vector or a matrix.

Figure 74: **Sqrt Block**



Data Type Support

Data type support is:

- Dimension: Input can be scalar, vector or matrix.
- Data Types: Input supports signals of integer, fixed-point and floating-point data type. It does not support Boolean inputs.
- Complex Numbers: Complex numbers are not supported.

The output has the same dimension and data type as the input.

Parameters

The Sqrt block has no parameters to set.

Stereo Block Matching

Estimates the motion or depth of objects in an image by locally computing intensity disparities for the objects between two consecutive frames (uint8 data type), or a stereo pair. It relies on the fact that the foreground objects have higher disparities than the objects in the background.

The output is a grayscale image (x_ufix16_EN4 data type) that represents disparities between the stereo pair.

Library

Computer Vision

L
R
Stereo Local
Block Matching
Stereo Block Matching

Description

Stereo block matching is a method of estimating the motion of the blocks between the consecutive frames, called a stereo pair. The idea behind this is that, considering a stereo pair, the foreground objects will have disparities higher than the background. Local block matching uses the information in the neighboring patch, based on the window size, for identifying the conjugate point in its stereo pair. The techniques under global method use the information from the whole image for computing the matching pixel, providing much better accuracy than local methods. But, the efficiency in the global methods are obtained with the cost of resources, which is where local methods stands out.

A local block matching algorithm consists of pre-processing and disparity estimation stages. The pre-processing consists of Sobel gradient computation followed by image clipping. And the disparity estimation consists of SAD (Sum of Absolute Differences) computation and obtaining the disparity using a winner takes all method (least SAD will be the disparity). Invalidity of the pixel relies upon its uniqueness from the other possible disparities. And the invalid pixels are indicated with the disparity value of zero.

Data Type Support

- The inputs are 2-D grayscale images of uint8 data type.
- The output is 2-D grayscale image of the same size as the input frames and of x_ufix16_En4 data type.

Parameters

- **Window width:** This parameter is the width of the small window to be formed around the pixel of interest. The default value is 9.

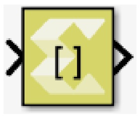
  The **Window width** must be a positive odd integer, greater than or equal to 5, and must be less than the minimum of the image width and height.
- **Number of disparities:** This parameter takes number of disparities to be computed for the given stereo pair. The default value is 32. The value must be greater than 1 and less than the width of the input image.
- **Number of disparities to be computed in parallel:** The value for this parameter decides how many disparities can be computed in parallel. The default value is 4. The ratio of "Number of disparities" and "Number of disparities to be computed in parallel" must be a non-fractional number.

Submatrix

Select a subset of elements (submatrix) from matrix input

Library

Math Functions / Matrices and Linear Algebra



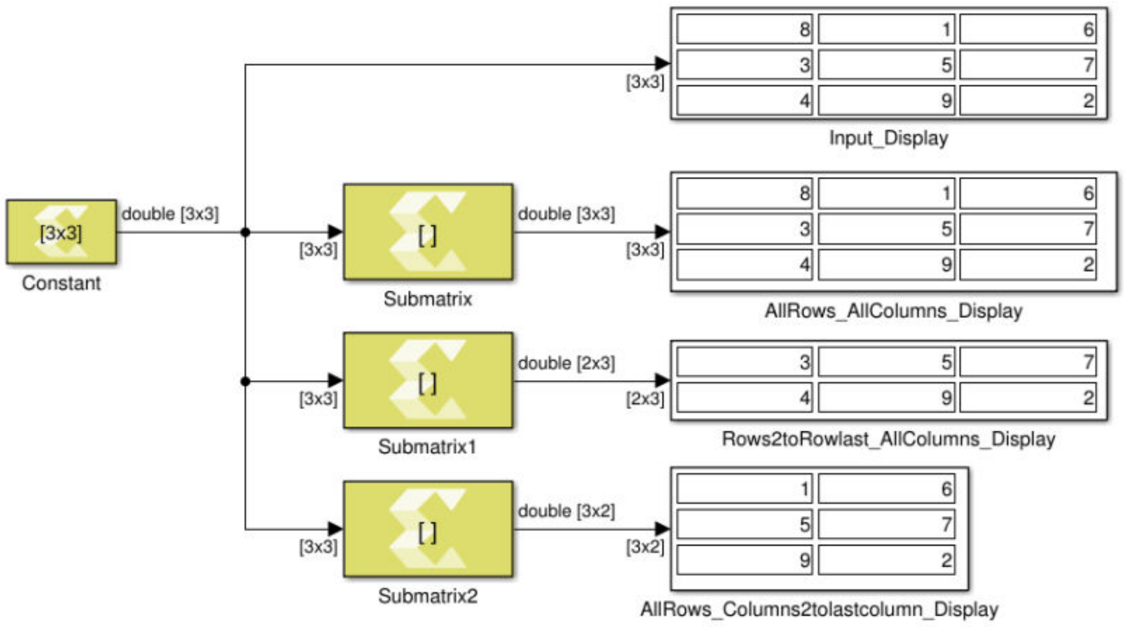
Description

The Submatrix block extracts a contiguous submatrix from the `M`-by-`N` input matrix `u`. The **Row span** parameter provides three options for specifying the range of rows in `u` to be retained in submatrix output `y`:

- **All rows**: Specifies that `y` contains all `M` rows of `u`.
- **One row**: Specifies that `y` contains only one row from `u`. The Row parameter (described below) is enabled to allow selection of the desired row.
- **Range of rows**: Specifies that `y` contains one or more rows from `u`. The **Starting row** and **Ending row parameters** are enabled to allow selection of the desired range of rows.

The Column span parameter contains a corresponding set of three options for specifying the range of columns in `u` to be retained in the submatrix `y`: **All columns**, **One column**, or **Range of columns**. The **One column** option enables the Column parameter, and Range of columns options enable the **Starting column** and **Ending column** parameters.

Figure 75: **Submatrix Block**



Data Type Support

All data types are supported. The output type is the same as the input type.

Parameters

- **Row span:** The range of input rows to be retained in the output. Options are **All rows**, **One row**, or **Range of rows**.
- **Row:** The input row to be used as the row of the output. **Row** is enabled when you select **One row** for **Row span**.
- **Row index:** The index of the input row to be used as the first row of the output. **Row index** is enabled when you select **Index** for **Row**.
- **Row offset:** The offset of the input row to be used as the first row of the output. **Row offset** is enabled when you select **Offset from middle** or **Offset from last** for **Row**.
- **Starting row:** The input row to be used as the first row of the output. **Starting row** is enabled when you select **Range of rows** for **Row span**.
- **Starting row index:** The index of the input row to be used as the first row of the output. **Starting row index** is enabled when you select **Index** for **Starting row**.
- **Starting row offset:** The offset of the input row to be used as the first row of the output. **Starting row offset** is enabled when you select **Offset from middle** or **Offset from last** for **Starting row**.
- **Ending row:** The input row to be used as the last row of the output. **Ending row** is enabled when you select **Range of rows** for **Row span** and you select any option but **Last** for **Starting row**.
- **Ending row index:** The index of the input row to be used as the last row of the output. **Ending row index** is enabled when you select **Index** for **Ending row**.
- **Ending row offset:** The offset of the input row to be used as the last row of the output. **Ending row offset** is enabled when you select **Offset from middle** or **Offset from last** for **Ending row**.
- **Column span:** The range of input columns to be retained in the output. Options are **All columns**, **One column**, or **Range of columns**.
- **Column:** The input column to be used as the column of the output. **Column** is enabled when you select **One column** for **Column span**.
- **Column index:** The index of the input column to be used as the first column of the output. **Column index** is enabled when you select **Index** for **Column**.
- **Column offset:** The offset of the input column to be used as the first column of the output. **Column offset** is enabled when you select **Offset from middle** or **Offset from last** for **Column**.
- **Starting column:** The input column to be used as the first column of the output. **Starting column** is enabled when you select **Range of columns** for **Column span**.

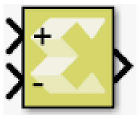
- **Starting column index:** The index of the input column to be used as the first column of the output. **Starting column index** is enabled when you select **Index** for **Starting column**.
- **Starting column offset:** The offset of the input column to be used as the first column of the output. **Starting column offset** is enabled when you select **Offset from middle** or **Offset from last** for **Starting column**.
- **Ending column:** The input column to be used as the last column of the output. **Ending column** is enabled when you select **Range of columns** for **Column span** and you select any option but **Last** for **Starting column**.
- **Ending column index:** The index of the input column to be used as the last column of the output. **Ending column index** is enabled when you select **Index** for **Ending column**.
- **Ending column offset:** The offset of the input column to be used as the last column of the output. **Ending column offset** is enabled when you select **Offset from middle** or **Offset from last** for **Ending column**.

Subtract

Perform element-wise subtraction

Library

Math Functions / Math Operations



Description

The Subtract block performs element-wise subtraction of two input signals.

The block warns or errors out when an integer output overflows during simulation. To configure, select **Simulation → Model Configuration Parameters → Diagnostics → Data Validity** for your model in the Simulink Editor, then set the **Wrap on overflow** or **Saturate on overflow** parameter.

Data Type Support

The Subtract block supports any floating point, fixed-point, integer or Boolean data type. The block can perform element-wise subtraction on real and complex inputs. The input signals can be scalars, vectors or matrices. When both inputs have non-scalar dimensions, the dimensions must match each other.

Parameters

Saturate on integer overflow

This parameter specifies whether integer overflow is handled by wrapping (default) or by saturating. This parameter is relevant only if the output is integral (int8, int16, int32, uint8, uint16, uint32).

Settings for the **Saturate on integer overflow** parameter are:

Table 83: **Saturate On Integer Overflow Parameter**

Setting	Description
Not selected	Integer overflow is handled by wrapping.
Selected	Integer overflow is handled by saturation.

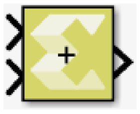
When overflow is detected, the Diagnostic Viewer displays messages that depend on the diagnostic action you specify in the Simulink Editor. To configure, select **Simulation → Model Configuration Parameters → Diagnostics → Data Validity** for your model in the Simulink Editor, then set the **Wrap on overflow** or **Saturate on overflow** parameter.

Sum

Perform element-wise addition of two input signals

Library

Math Functions / Math Operations



Description

The Sum block performs element-wise addition of two input signals.

The block warns or errors out when an integer output overflows during simulation. To configure, select **Simulation → Model Configuration Parameters → Diagnostics → Data Validity** for your model in the Simulink Editor, then set the **Wrap on overflow** or **Saturate on overflow** parameter.

Data Type Support

Data types accepted at the inputs of the block are as follows.

- This block supports all data types supported by Xilinx Model Composer. The block supports real and complex numbers.
- The input signals can be real or complex numbers of scalar, vector or matrix type. When both inputs are non-scalar then their dimensions must match.

Output data types are as follows.

- If the data type of one input is a floating point type, the data type of the output is the floating point type among the data types of the inputs with the most precision.
- Otherwise, if the data type of one input is a fixed-point type, the data type of the output is the smallest fixed-point type capable of representing the result without any loss of precision.
- Otherwise, if the data type of both inputs is Boolean the output is Boolean as well.
- Finally, if one input is integral and the other is also integral or Boolean, the output is integral. If both inputs are unsigned the output is unsigned, otherwise it is signed. The bit width of the output is largest among the bit widths of the inputs.

Parameters

Saturate on integer overflow

This parameter specifies whether integer overflow is handled by wrapping (default) or by saturating. This parameter is relevant only if the output is integral (int8, int16, int32, uint8, uint16, uint32).

Settings for the **Saturate on integer overflow** parameter are:

Table 84: **Saturate On Integer Overflow Parameter**

Setting	Description
Not selected	Integer overflow is handled by wrapping.
Selected	Integer overflow is handled by saturation.

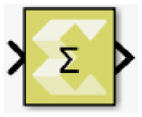
When overflow is detected, the Diagnostic Viewer displays messages that depend on the diagnostic action you specify in the Simulink Editor. To configure, select **Simulation → Model Configuration Parameters → Diagnostics → Data Validity** for your model in the Simulink Editor, then set the **Wrap on overflow** or **Saturate on overflow** parameter.

Sum of Elements

Perform element-wise addition on the input, column-wise, row-wise, or in all dimensions

Library

Math Functions / Math Operations



Description

This block performs element-wise addition on a vector or matrix type input. If input is a scalar then this block operates as a pass-through. The output is a scalar type if input is a vector or a matrix and the **Sum over** parameter is set as **All dimensions**.

Data Type Support

Data type support for the block is:

- The Sum of Elements block supports any floating-point, fixed-point, integer or Boolean data type.
- The output type is a scalar or a vector depending on the dimensions of the input and the selection of the **Sum over** parameter.
- The block can perform element-wise addition on real or complex number data.

Parameters

Sum over

The **Sum over** parameter value is used to decide whether elements will be added in all dimensions or in one of the dimensions.

Following are settings for the **Sum over** parameter.

Table 85: **Sum Over Parameter**

Setting	Description
All dimensions	Add all elements of the input signal (output is scalar)
Specified dimension	This option shows an edit box, **Dimension**, where the specific dimension value can be entered.

Dimension

The **Dimension** parameter is displayed only if the **Sum over** parameter value is set to **Specified dimension**.

Settings for the **Dimension** parameter are:

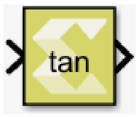
Setting	Description
1	Add input over row dimension. Output is a row matrix.
2	Add input over column dimension. Output is a column matrix.

Tangent

Perform an element-wise computation of the tangent function for the given argument

Library

Math Functions / Math Operations



Description

The block returns the output of the function tan(`x`) for each element in array `x`.

Data Type Support

Data types accepted at the input of the block are:

- Dimension : Input can be scalar, vector or matrix.
- Data Types: Input supports signals of integer type, floating point type (double, single and half) and fixed-point type.
- Complex Number Support: No

  Output has the same dimension and type as the input.

Parameters

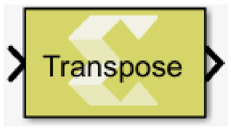
The Tangent block has no parameters to set.

Transpose

Perform an element-wise transpose operation on the input signal

Library

Math Functions / Matrices and Linear Algebra



Description

The Transpose block performs a transpose operation on the input signal.

Data Type Support

This block supports all data types supported by Xilinx Model Composer. The input signal can be real or a complex number of scalar, vector, or matrix type. The output type is always the same as that of the input.

Parameters

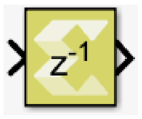
The Transpose block has no parameters to set.

Unit Delay

Provides a delay of one sample period

Library

Signal Operations



Description

The Unit Delay block provides a delay of one sample period. The output has the same sample time as the input. The output dimension is the same as the initial condition dimension if the input is scalar. Otherwise, the output dimension is the same as the input dimension and should match the dimension of the initial condition.

Data Type Support

Data type support is:

- All data types are supported.
- Input can be a vector or a matrix. If input is a vector or a matrix and the initial value is a scalar, the scalar value will apply to all the elements of the input during the first cycle.
- Output is complex if the input is complex.
- If the initial value is complex but the input signal is type real, the block gives an error indicating that the input type must be complex.

Parameters

Initial Condition

Specifies the initial value.

The **Initial Condition** can be scalar, vector, or matrix, of real or complex type.

Warp Transform

Perform either an Affine or a Perspective transform of an 8-bit grayscale image, according to the 2x3 or 3x3 transformation matrix provided to the input port T, respectively.

Library

Computer Vision

Warp
T Transform

Description

The Warp Transform block takes an image and a real-valued 2x3 or 3x3 transformation matrix as inputs and produces either an Affine or a Perspective transformation of the input image, respectively.

- **Affine Transforms:**

  An Affine transformation consists of scaling, rotation, and translation. For example, you can calculate a 2x3 rotation matrix as follows:

$$\begin{bmatrix} \alpha & \beta & (1-\alpha)\times center_x - \beta \times center_y \\ -\beta & \alpha & \beta \times center_x + (1-\alpha)\times center_y \end{bmatrix}$$

$$\alpha = scale \times \cos(angle)$$

$$\beta = scale \times \sin\,(angle)$$

where:

> center = Center of the rotation in the source image.
>
> angle = Rotation angle in radians. Positive values mean clockwise rotation. The coordinate origin is assumed to be the top left corner.
>
> scale = Scaling factor.

The following link contains information on further Affine transformations that can be applied, for example, shearing, reflection, and other transformations:

https://en.wikipedia.org/wiki/Affine_transformation#Image_transformation

You can use the following function to calculate the affine transform matrix:

```
function T = affine_transform_matrix(rotation_angle_radians, scale,
center_x,center_y)

a = scale*cos(rotation_angle_radians);
b = scale*sin(rotation_angle_radians);

T = [a b (1-a)*center_x - b*center_y; -b a b*center_x+(1-a)];

end
```

As an example, we can use the following command to create the transform matrix to rotate an input image of size 384x512 by 30 degrees around the center of the image:

```
T = affine_transform_matrix(pi/6,1,512/2,384/2)
```

Figure 76: **Warp Transform Block**

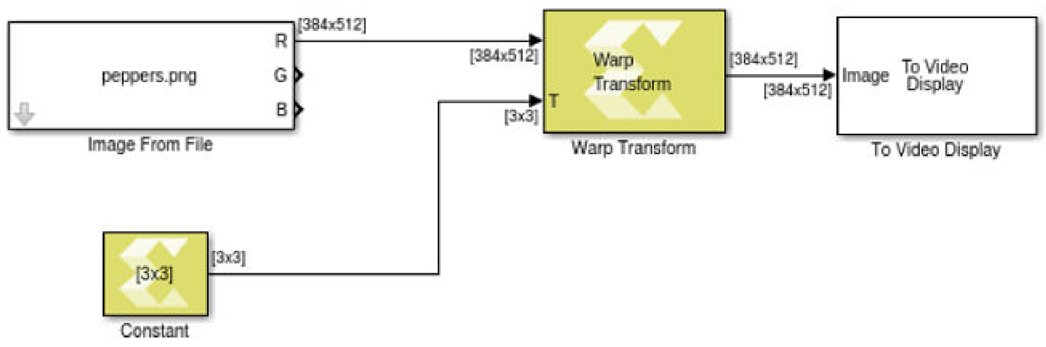
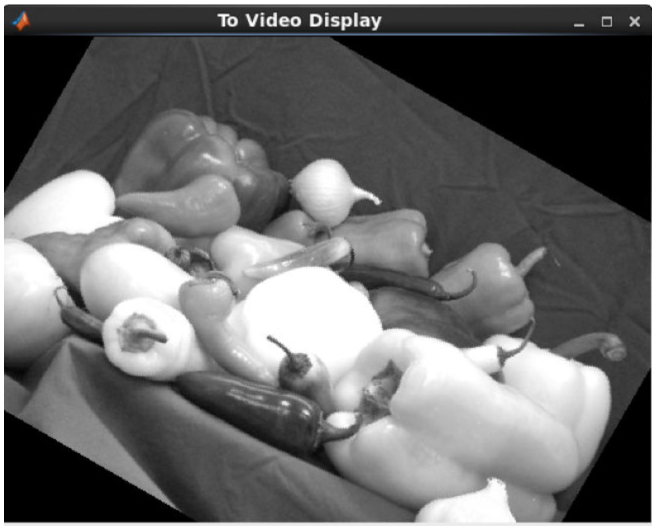


Figure 77: **Transform Example**



Perspective Transforms

A perspective transform is a 3-D projection mapping of the image data within an arbitrary convex quadrilateral area of the source image to the output image. The 3 x 3 transformation matrix required can be determined by solving two systems of linear equations defined by the four corners of the quadrilateral and the destination image, respectively and combining the results as detailed in the MATLAB® code below:

```
% Specify the quadrilateral source patch
src = {[x1 y1], [x2 y2], [x3 y3], [x4 y4]};
% Specify destination frame of size [width height]
dst = {[0 0], {width 0], [width height], [0 height]};

M = findTransformMat(src, dst);
```

with the `findTransformMat()` function defined as below:

```
% srcPts and dstPts are cell arrays with four 2-D points each
function C = findTransformMat(srcPts, dstPts)
  linEq1 = [];
  linEq2 = [];
  for i = 1:3
    linEq1 = [linEq1(:,1:end) [cell2mat(srcPts(i)) 1]'];
    linEq2 = [linEq2(:,1:end) [cell2mat(dstPts(i)) 1]'];
  end
  res1 = [cell2mat(srcPts(4)) 1]';
  res2 = [cell2mat(dstPts(4)) 1]';
  coeffs1 = inv(linEq1) * res1;
  coeffs2 = inv(linEq2) * res2;
  A = repmat(coeffs1',3,1) .* linEq1;
  B = repmat(coeffs2',3,1) .* linEq2
  C = B * inv(A);
end
```

Data Type Support

The block supports grayscale images of 8-bit pixel depth, that is, uint8 data type, for both input and output images. The image data has to be real-valued. Complex numeric types are not supported.

The transformation matrix has to be a real-valued matrix of single data type with dimensions of 2x3 for Affine transformation, and 3x3 for Perspective transformation.

Parameters

- **Transformation type:**

  The **Transformation type** parameter allows you to select between Affine and Perspective warp transformation.

  - **Settings:**

    Table 86: **Transformation Type Parameter**

Setting	Description
Affine transform	The block performs an Affine warp transformation using a 2x3 transformation matrix allowing various combinations of rotation, translation, scaling, shearing, and mirroring of the input image.
Perspective transform	The block performs a 3-D perspective projection of an area in the source image defined by a quadrilateral using a 3x3 transformation matrix..

- **Interpolation:**

  The **Interpolation** parameter allows you to select between **Nearest neighbor** and **Bilinear** interpolation options.

  - **Settings:**

    Table 87: **Interpolation Parameter**

Setting	Description
Nearest neighbor	Nearest neighbor interpolation picks the value of the pixel that is closest to the computed coordinates.
Bilinear	Bilinear interpolation computes the linear interpolation between the four neighboring pixels closest to the computed coordinates.

    ***Note*: Nearest neighbor** interpolation is more resource and latency efficient than **Bilinear** interpolation, at the cost of reduced image quality.

- **Percentage of input image size:**

  ***Note*:** This parameter does not affect simulation behavior. For simulation purposes the entire input image is buffered up before the transformation takes place.

The **Percentage of input image size** parameter sets the size of the streaming line buffer in the synthesized implementation of this block, in terms of percentage of the input image size. For example, a setting of 50% for an input image in Full-HD resolution (1920 column, 1080 rows) creates a line buffer holding 540 rows of the input image (1920 x 540 = 1,036,800 bytes).

Note: Buffer size is always a multiple of the full image width.

- **Settings:** The spin-box control allows you to set values from 20-100% in steps of 10%.
- **Tips:**

  The amount of line-buffer needed is dependent upon the actual transformation matrix at any given time. For static transforms the required buffer size can be determined analytically and set appropriately. Moderate transforms such as rotation by a small angle, or translation by a small amount require only a minimum of buffering, but flipping images upside down or free rotation require a full size line buffer.

- **Ratio of buffer used for look-ahead operations (%):**

  Depending on the nature of the transformation the block processing may need to "look ahead" into the line buffer compared to the current pixel position. The **Ratio of buffer used for look-ahead operations (%)** parameter allows you to specify the amount of the image that is to be buffered up before processing starts.

  - **Settings:**

    You can set this parameter to values from 0-100% in steps of 10%.

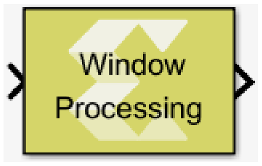
  - **Dependencies:** This parameter is given as a percentage of the line-buffer size, not to be mistaken for a percentage of the image height. As such, it is dependent on the previous parameter **Percentage of input image size**.
  - **Tips:** An example for a transformation that does not need pre-fetching is to shift the image down by a few lines, because the transformation would only require access to "past" samples of the input stream. Shifting the image up would require "look-ahead" access to "future" samples in the input stream, which is why processing needs to be delayed until those samples have been streamed into the line-buffer.

Window Processing

Assemble an output matrix by applying the kernel subsystem to submatrices (windows) of the input matrix in row-major order

Library

Computer Vision



Description

The Window Processing block is a masked subsystem for assembling an output matrix by applying a kernel subsystem to sub matrices of the input matrix in row-major order. You customize the Window Processing block by specifying its parameters and by adding blocks to the Kernel subsystem for computing a scalar element of the output in terms of the corresponding input window. The computation can be thought of to proceed in the following steps:

1. Compute the padded input matrix depending on the setting of the **Output size** parameter.

   Let the dimensions of the input be Min,Nin; let the dimensions of the window be Mwin, Nwin.

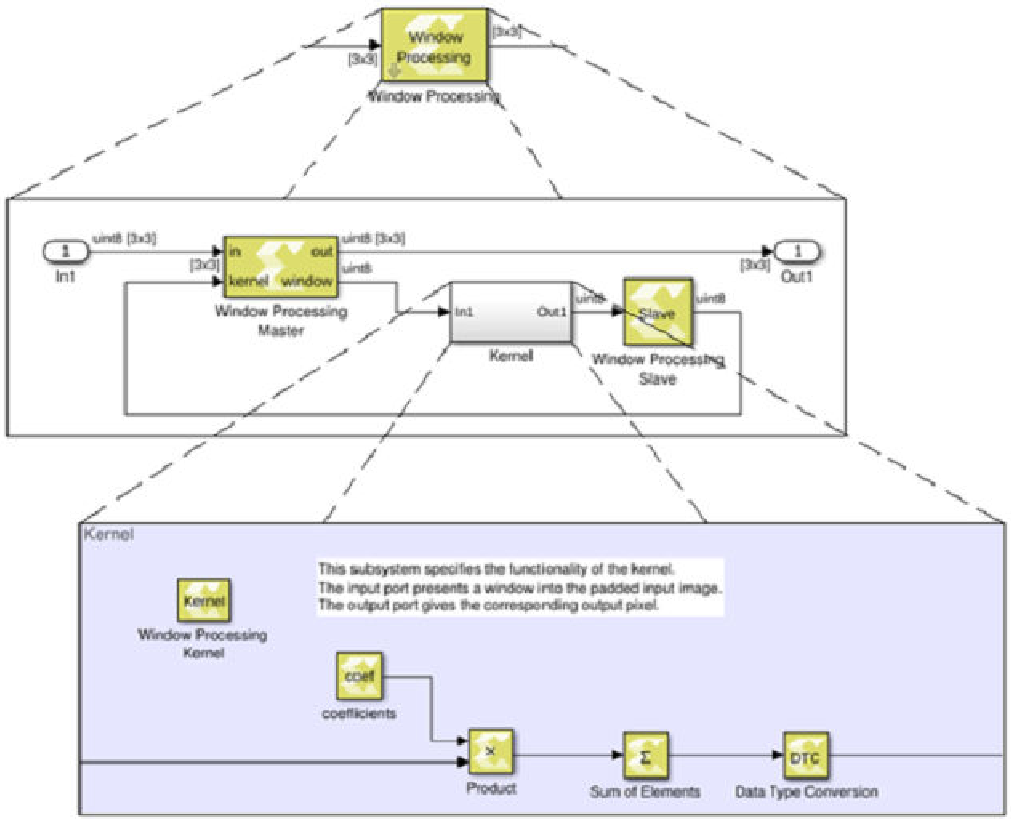
   - If the **Output size** is set to **Valid**, no padding is performed, and the dimensions of the output are Min-Mwin+1,Nin-Nwin+1.
   - If the **Output size** is set to **Same as input**, the input is padded with Mwin-1 rows and Nwin-1 columns of 0s. Half of rows added is put at the top and the remainder goes at the bottom. If Nwin is even then the bottom gets one more row of padding than the top. The same is done for adding the columns. The size of the output is the same as the size of the input.
   - If the **Output size** is set to **Full**, the input is padded with 2*(Mwin-1) rows and 2*(Nwin-1) columns of 0s. Half of the padding rows (columns) are added at the top (left) and the other half at the bottom (right).

2. For each element (i,j) of the output (iterate over output in row-major order):

   - Select the MxN sub matrix from the padded input matrix starting with element (i,j)
   - Apply the kernel subsystem to this sub matrix
   - Assign the scalar output of the kernel to element (i,j) of the output.

The hierarchy of the Window Processing block is shown below.

Figure 78: **Hierarchy of Window Processing Block**



Restrictions on the use of the Window Processing block are:

- The topology of the window processing subsystem shall not be modified. It must match the topology show in the figure above.
- The input signal must be a matrix.
- The kernel must have precisely one input port and one output port.
- The output of the kernel must be scalar.
- The input of the kernel must have the dimensions specified by the window size parameter.
- The dimensions of the input must be at least as large as the window size.
- The kernel must not contain any of the following blocks:
  - Blocks that have internal state, such as the Unit Delay block.
  - The following Computer Vision blocks: Bilateral Filter, Histogram Equalization, Otsu Threshold, and Warp Transform.
  - The following Digital Signal Processing blocks: FFT and IFFT.

    - Blocks created with the `xmcImportFunction` command.
- The kernel must not contain an if-action subsystem.

Data Type Support

There are no restrictions on the data types of the input or output signals.

Parameters

Window size

This parameter specifies the size of the window. Enter a 2-element vector of real positive integers, for example [5,3], for the **Window size**. Please note that **Window size** cannot specify more than a total of 255 elements.

Output size

This parameter specifies how the edges of the input image are treated.

Let Min, Nin be the dimensions of the input, and let Mwin, Nwin be the **Window size**. The dimensions of the output are as follows:

- Min+Mwin-1, Min+Nwin-1 if **Full** is selected for **Output size**.
- Min, Nin if **Same as Input** is selected for **Output size**.
- Min-Mwin+1, Min-Nwin+1 if **Valid** is selected for **Output size**.

Appendix C

Additional Resources and Legal Notices

Xilinx Resources

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

Solution Centers

See the Xilinx Solution Centers for support on devices, software tools, and intellectual property at all stages of the design cycle. Topics include design assistance, advisories, and troubleshooting tips.

Xilinx Resources

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

Documentation Navigator and Design Hubs

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

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

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

- In DocNav, click the **Design Hubs View** tab.

- On the Xilinx website, see the Design Hubs page.

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

References

These documents provide supplemental material useful with this guide:

1. *Introduction to FPGA Design with Vivado High-Level Synthesis* (UG998)
2. *Vivado Design Suite User Guide: High-Level Synthesis* (UG902)
3. *UltraFast High-Level Productivity Design Methodology Guide* (UG1197)
4. Vivado® Design Suite Documentation
5. *Vivado Design Suite User Guide: Designing with IP* (UG896)
6. *Vivado Design Suite User Guide: Creating and Packaging Custom IP* (UG1118)
7. *Vivado Design Suite User Guide: Model-Based DSP Design Using System Generator* (UG897)
8. MathWorks® Simulink® Documentation

Please Read: Important Legal Notices